

Programmer to Programmer™



Professional

VSTO 2005

Visual Studio® 2005 Tools for Office

Alvin Bruney



Updates, source code, and Wrox technical support at www.wrox.com

**Professional
VSTO 2005**

**Visual Studio® 2005
Tools for Office**

**Professional
VSTO 2005**

**Visual Studio® 2005
Tools for Office**

**Professional
VSTO 2005**

**Visual Studio® 2005
Tools for Office**

Alvin Bruney



WILEY

Wiley Publishing, Inc.

Professional VSTO 2005: Visual Studio® 2005 Tools for Office

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2006 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN-13: 978-0-471-78813-3
ISBN-10: 0-471-78813-9

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

1O/RX/QU/QW/IN

Library of Congress Cataloging-in-Publication Data:

Brunev, Alvin, 1971-
Professional Visual studio 2005 tools for Office / Alvin Brunev.
p. cm.

Includes index.
ISBN-13: 978-0-471-78813-3 (paper/website)
ISBN-10: 0-471-78813-9 (paper/website)
1. Microsoft Visual studio. 2. Web site development--Computer programs. 3. Microsoft Office. I. Title.
TK5105.8885.M57B786 2006
005.5--dc22

2006005385

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at <http://www.wiley.com/go/permissions>.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Visual Studio is a registered trademark of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

About the Author

Alvin Bruney is a senior software engineer for Indigo Books and Music. His previous development jobs included spearheading the .NET architecture for NetworkIP, a telecommunications provider, and as a programmer with Intuit. He self-published a book on Office Web Components, and frequently writes articles for *ASP.NET Professional* magazine, MSDN, and other online venues. He is also a Microsoft .NET MVP and is well known in the 10 newsgroups he monitors.

Credits

Senior Acquisitions Editor

Jim Minatel

Development Editor

Brian MacDonald

Technical Editors

Jay B. Harlow

Phred Menyhert

Production Editor

Pamela Hanley

Copy Editor

Foxxe Editorial Services

Editorial Manager

Mary Beth Wakefield

Production Manager

Tim Tate

Vice President and Executive Group Publisher

Richard Swadley

Vice President and Executive Publisher

Joseph B. Wikert

Quality Control Technicians

John Greenough

Charles Spencer

Brian H. Walls

Graphics and Production Specialists

Denny Hager

Barbara Moore

Alicia B. South

Proofreading and Indexing

Techbooks

Kathy, thanks for being there for me all the time.

Boobie tube, you aren't old enough to read this yet, but I love you.

Acknowledgments

Thanks to the staff of Wiley for believing in this project and having the discipline to see it through. Thanks to my family for putting up with the anxious moments and erratic behavior. It was worth it. Thanks to the One for the energy to pull this one off.

Contents

Acknowledgments	ix
Introduction	xvii
Chapter 1: Visual Studio Tools for Office	1
What's New in VSTO?	1
VSTO Architecture	2
User Interface	3
Client Interface	3
Server Component	3
The VSTO Package	4
About Microsoft Office PIAs	5
System Requirements	6
Alternatives to the VSTO Office Systems	7
VBA	7
Office Web Components	8
Excel COM Interop Libraries	8
Third Party Products	9
Disadvantages of VSTO	9
.NET Framework Required	9
Security	9
Performance	10
VSTO Automation	10
Office XML Schemas	11
Installation and deployment	11
Creating VSTO Projects	13
VSTO Installation Issues	16
Summary	17
Chapter 2: Excel Automation	19
Excel Data Manipulation	19
Design-Time Data Loads	20
Loading Files at Runtime	21
Application Object Manipulation	30
Workbook Manipulation	31
Worksheet Manipulation	32

Contents

Excel Range Manipulation	34
Working with Named Ranges	35
Working with Cells	37
Working with Unions	38
Working with Intersections	39
Working with Offsets	40
Data Formatting and Presentation	42
Considerations for Excel Automation	44
Excel Case Study — Time Sheet Software	45
Summary	52
Chapter 3: Advanced Excel Automation	55
VSTO Security	55
Workbook Password Protection	55
Worksheet Security	57
Protection through Hidden Worksheets	60
Protecting Ranges	61
VSTO Security through the .NET Framework	62
Working with Excel Formulas	66
Using Formulas at Design Time	66
Using Formulas at Runtime	68
Working with the WorksheetFunctions Method	69
Excel Spreadsheet Error Values	69
Responding to Events	71
Working with Workbook Controls	74
The Range Control	75
The List Control	76
The Actions Pane	78
Printing Workbook Data	80
Excel Toolbar Customization	82
Excel Menu Customization	85
VSTO and Web services	89
Excel Server Automation	95
Excel COM Add-Ins	101
Summary	104
Chapter 4: Word Automation	107
Key Application Objects in Word	107
ThisApplication Instance	108
Working with the Range Object	109
Working with the Bookmark Object	112

Working with the Selection Object	114
VSTO Table Manipulation	115
Working with Documents	119
Manipulating Office Documents	120
Working with Word Templates	123
Toolbar Customization	126
Menu Customization	131
Working with Other Office Applications	134
A PowerPoint Automation Example	134
An Office Automation Executable Example	137
A Smart Tag Example	139
Manipulating Office Controls	143
Word Event Model	147
Printing Documents from Word	149
Considerations for Word Development	149
PIAs and RCWs for Office Word	149
Assemblies and Deployment	150
Custom Actions Panes	150
Summary	151
Chapter 5: Outlook Automation	153
Configuring VSTO Outlook	154
Key Outlook Objects	157
What Is MAPI?	157
Application Object	158
Explorer Object	158
Inspector Object	159
Email Integration	161
Creating Emails	161
Manipulating Email Messages	163
Appointments and Meetings	165
Creating and Scheduling Appointments	165
Deleting Appointments	170
Creating Meetings	172
Creating and Scheduling Outlook Items	173
Folder Manipulation	176
Address Book Manipulation	177
Events	180
Data Manipulation	183
Searching for Data	184
Advanced Data Search	187
Object Model Guard	190

Contents

Working with Office Controls	191
Toolbar Customization	192
Menu Customization	195
Integrating Windows Applications with Outlook	198
Summary	206
Chapter 6: The Charting Tool	209
Design-Time Charting	209
Chart Wizard	210
Charting Nomenclature	211
Creating VSTO Charts	219
Stand-Alone Charts	219
Embedded Charts	220
Loading Charts with Data	221
Essential Chart Object Hierarchy	226
Series Manipulation	226
Axes and Scaling	226
Tick Marks	228
Titles and Captions	232
Chart Groups	234
Formatting Chart Data	236
Font Object Customizations	237
Number Format Customization	238
Chart Surface Customization	240
Chart Legend Manipulation	242
Analysing Data	245
Trending through Trend Lines	246
Internally Supported Trendlines	246
Custom Trendlines	248
Error Bar Analysis	250
Chart Customization through Events	251
Advanced Chart Manipulation	254
Charts with Multiple Series	254
Combination Charts	256
Perspectives and 3-D Drawing	258
Chart Point Customization	260
Adding Objects to the Chart Surface	265
Charts with Special Needs	268
Chart Limitations	271
Summary	272

Chapter 7: Pivot Table Automation	273
Design-Time Pivoting	274
VSTO PivotTable Wizard	275
Pivoting and Drilling	277
Pivot Table Terminology	278
PivotCache Object	279
PivotData Objects	279
Pivot Axis	279
Pivot Fields	280
Pivot Cell Object	280
Pivot Labels	281
Creating Pivot Tables through Code	281
Formatting Pivot Data	285
Font Object Customizations	285
Label Format Customization	286
Style Object Customization	287
Data Formatting with NumberFormats	289
Pivot Axis Iteration	290
Pivot Filtering	292
Adding Miscellaneous Columns	293
Pivot Table Events	294
Creating Pivot Charts	295
Pivot Table Limitations	297
Summary	298
Index	299

Introduction

Visual Studio Tools for Office (VSTO) grew out of a need to write enterprise software based on Office technology. The book chooses to satisfy this need from a practical perspective. Therefore, the book is focused on delivering practical solutions for those seeking to port existing functionality to VSTO. Where necessary, however, the book injects a healthy dose of theory so that developers can learn to think and feel in this new technology.

The book is especially written for enterprise developers, the VBA or COM Interop savvy—who need to leverage the power and productivity of VSTO today. .NET developers seeking to explore VSTO as a potential solution will also find ample material that suits their needs. Even if you are not at all familiar with VSTO, the book is designed to build your knowledge in this new technology, working first with simple concepts on up to more involved strategies and solutions for enterprise software.

Who This Book Is For

This book is written for developers who are considering adopting VSTO as an enterprise solution. The book assumes that you are familiar with object-oriented concepts. The book also assumes that you are familiar with Visual Studio .NET. However, knowledge of Visual Studio .NET is not necessarily a requirement.

For the most part, the technical language in the book is mild to moderate. Concepts are presented from a coding perspective targeting intermediate to advanced developers. The material is presented in such a way that it serves to encourage non-.NET programmers. The language and focus of the examples are not intimidating and have been specifically refined to encourage learning.

The book adopts a code-first approach to satisfy VBA and COM Interop developers. The code to implement common programming tasks possesses a premium above the theory behind the requirement. For the most part, developers seeking to port functionality from VBA to VSTO are less interested in the why and more interested in the how. This book caters wholly to this school of thought.

If you find yourself already comfortable with core concepts, then you may benefit from reading the latter chapters. Even if you are a seasoned developer, you will still find a lot of useful techniques and strategies for solving enterprise-level software problems from an Office perspective.

What This Book Covers

This book covers VSTO 2005 in detail. The approach focuses on the major components that form the suite. One exception is that InfoPath is not covered. All other components benefit from an exploration of the key objects that are most likely to be used in common programming scenarios. Once the basics are out of the way, the material covers common programming requirements from a practical perspective.

Introduction

This book is not a discourse in theory. The emphasis is on code. Code is presented in both Visual Basic and C# with no bias toward either. However, the code is always explained with a dose of theory when needed. Toward the end of each chapter, the book dives into wider strategies for common problems.

VSTO 2005 is a well-oiled machine, but too often, the focus of this new technology has been on Excel and Outlook. While this book offers complete coverage of these objects, it also presents the lesser-known parts of VSTO, including pivot tables and charts. For instance, there are many companies developing applications based on VSTO while using third-party charting products. VSTO contains a potent charting engine second to none. And this book explores every facet of charting.

While you may certainly apply techniques learned in this book to Visual Studio Tools for Office 2003, that is not the intent of this book. For the most part, Visual Studio Tools for Office 2005 has undergone drastic changes that really separate it from its predecessor.

How This Book Is Structured

This book is organized into chapters that present the building blocks of VSTO first. Microsoft Excel and its `Range` objects form the cornerstone of range manipulation across the VSTO suite. For that reason, the first few chapters explain these basic concepts. Subsequent chapters use these building blocks as a starting point, so it is important to be familiar with these concepts before skipping to the back of the book.

Once you gain a complete understanding of the Excel `Range` object in Chapters 2 and 3, you can apply that knowledge to the remaining chapters. In fact, Microsoft Word, charts, and pivot table manipulation are all based on the Excel `Range` object. The obvious benefit of this clever architecture is that it significantly reduces the learning curve for those who wish to adopt this relatively new technology. And this book is designed to show you how to exploit this architecture.

What You Need to Use This Book

To compile and run the examples in this book, you will need to install Visual Studio 2005 and Microsoft Office Professional Edition with Service Pack 1. The Professional Edition is an absolute requirement because VSTO cannot run without it. However, Visual Studio 2005 is not an absolute requirement. If you have purchased the stand-alone version of Visual Studio Tools for Office 2005, you do not need Visual Studio 2005 to compile and run VSTO-based applications.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.

Tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

As for styles in the text:

- ❑ We *highlight* new terms and important words when we introduce them.
- ❑ We show keyboard strokes like this: Ctrl+A.
- ❑ We show file names, URLs, and code within the text like this:
`persistence.properties.`
- ❑ We present code in two different ways:

In code examples we highlight new and important code with a gray background.

The gray highlighting is not used for code that's less important in the present context or has been shown before.

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at www.wrox.com. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 0-471-78813-9.

Once you download the code, just decompress it with your favorite decompression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, such as a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher-quality information.

To find the errata page for this book, go to www.wrox.com and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page, you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at www.wrox.com/misc-pages/booklist.shtml.

Introduction

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to email you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com> you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

- 1.** Go to p2p.wrox.com and click the Register link.
- 2.** Read the terms of use and click Agree.
- 3.** Complete the required information to join as well as any optional information you wish to provide, and click Submit.
- 4.** You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum emailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

Visual Studio Tools for Office

Visual Studio Tools for Office is the new kid on the block for harnessing the power and functionality of Microsoft Office technology. And Microsoft is investing heavily in getting the word out. The marketing push forms part of an overall strategy to secure the dominance of Office technology and Microsoft Office products on the desktop while also providing a solid platform for developing enterprise level applications based on Microsoft Office technology.

At its core, Visual Studio Tools for Office (VSTO, pronounced “visto”) is simply a platform that allows Microsoft Office documents to execute code wrapped in a .NET assembly. It is certainly not the only technology available for developing applications based on Microsoft Office. In fact, web developers have been using classic ASP and regular COM Interop to build Office based software for years. But VSTO has been designed to make development of such applications easier and more reliable than the current approaches available today.

What’s New in VSTO?

VSTO has been completely reworked for .NET 2005. The rework includes tighter integration with Excel as well as the promotion of native Excel and Word binaries to bona fide .NET objects. These new objects expose more functionality than what is currently available through the aging Microsoft Office Application Programming Interface (API). Also, these new .NET objects resolve many of the memory management issues that plagued the previous Microsoft Office Component Object Modeler (COM) components.

VSTO sports a new development model, a novel approach to building Office-based applications that offers significant advantages when compared to previous models. The Classic ASP model that predates VSTO was based on a platform where content and presentation were mangled together to construct applications.

One advantage is that there is more room to scale because business logic can be cleanly separated from the presentation of data. Another advantage is that Office automation can occur on the web application server in a thread-safe way without the need to write code to adjust the presentation of the data. A third advantage is that the new model allows executing code to exploit the rich feature set of the .NET Framework Class Library. The downside, and there always is one, is that these improvements come at the expense of an increased learning curve, and the need to write code to target a new model.

To help drive the acceptance of VSTO, Microsoft is using the .NET framework as the vehicle to move the new Office development platform into the midst of the .NET developer population. The marketing strategy is geared toward enticing .NET developers into building applications based on Microsoft Office technology. Hopefully, if .NET developers find it easy to write Office applications, then they will.

Unfortunately, the marketing picture may not always present the whole story. This book will peel back the marketing hype baked into Visual Studio Tools for Office to show you what works well and what does not. The book also intends to show you, in a very practical way, how to build and deliver industrial strength applications based on Microsoft Office technology. The material in the next few sections will compare and contrast VSTO with some current approaches for developing Microsoft Office based applications so you can decide whether or not VSTO is the right tool for the job.

VSTO Architecture

VSTO is not entirely new. The first .NET tool suite appeared with Visual Studio 2003. Before that, savvy developers simply used COM Interop and Visual Basic for Applications (VBA) to etch out Office based software. The effort required was considerable due in part to the limitations of the COM based VBA programming model. The chosen few who succeeded in building Office applications took on the crown of “Office Developers”.

Office developers used any means necessary — macros, hacks, Interop — to coax performance and stability out of these crude, unforgiving environments. Along the way, the approach to building Office applications got a bad rap because hacks and macros cultivated an environment for malicious software to flourish — even today, embedded macros in applications still cause all sorts of security concerns. Microsoft is hoping to turn a new page with Visual Studio Tools for Office. VSTO is safer and more secure than previous technologies used to build Microsoft Office applications. In fact, VSTO can build fully operational applications without macros.

VSTO.NET addresses the safety and security issues in a couple of ways. Firstly, the new document/view architecture allows code to be separated from presentation. The code that runs behind the Office document can have security restrictions applied to it so that it is guaranteed safe and its authenticity can be verified before being run. Contrast this to the macro approach where embedded macros are permitted to execute on the user system unbounded. The new VSTO approach provides developers with more freedom to build secure, scalable applications than what was possible before. In that sense, VSTO carves out new ground, and it does so in a way that eliminates the need to write trick code.

Office 2003 sports a new surface for embedding objects into the document — the actions pane. The actions pane reduces the effort that is required to host controls on the document surface. Controls can now be hosted cleanly and efficiently on the document. This new approach leads to well-behaved applications and side-steps the need to implement hacks to achieve the impossible.

VSTO also hosts the Excel and Word objects inside the Microsoft Visual Studio designer. When you open a new Excel project in Visual Studio 2005, the Excel spreadsheet appears in the Microsoft Visual Studio designer. The Excel object is able to respond to design-time settings and is fully customizable. By hosting Excel and Word on the document surface, debugging the application is a lot more manageable.

VSTO exposes bona fide .NET objects that seamlessly integrate with the .NET framework. These objects expose more functionality and are cleaner to incorporate in managed software development efforts. The approach also stems the tide of memory exhaustion issues and component instability that plagued the previous generation of Office software.

VSTO offers n-tier architecture for Microsoft Office document development. The model is tiered into the user interface, client interface, server components, and backend data.

User Interface

The user interface tier functions as the front-end of the document. It handles user input, validation, and document navigation. This is all handled through the same Excel, Word, InfoPath, or Outlook interface that users are accustomed to. By basing the user interface on popular, proven front-end, very little end-user training is required for those already familiar with Microsoft Office functionality.

Client Interface

The client component is made up of a combination of the .NET assembly developed for the new VSTO-based application and the VSTO.NET engine provided by Visual Studio Tools for Office System. The engine floats on top of the Common Language Runtime infrastructure and allows the Microsoft Office document to execute managed code inside the .NET assembly.

Server Component

Every VSTO-based application has direct access to VSTO's new server component class. This server component piece functions as a document processing layer that is able to create and manipulate Microsoft Office documents without the need to automate Microsoft Office or Microsoft Excel on a web server. In case you missed it, server-side processing of Microsoft Office documents have traditionally been the weakest link in software applications that run in a web-server environment. The new server component strengthens this link by allowing software applications based on Microsoft Office to scale correctly while making prudent use of web-server resources.

The server component architecture incorporates the use of XML in its design and implementation. The data that forms part of the Microsoft Office document being manipulated on the web-server is now stored independently in XML data islands. This is a major departure from all previous Microsoft Office based implementations where the application data was stored inside the Microsoft Office document itself.

The XML used to store the application data is based on the W3C XML open standard. Data storage is no longer proprietary as in some previous technologies such as the Microsoft Office Web Components. The business implication is that data sources can be as diverse as the corporate entity allows it to be. And, data can now flow freely between Microsoft Office and third party applications built to take advantage of open standards for data exchange such as web services and remoting platforms.

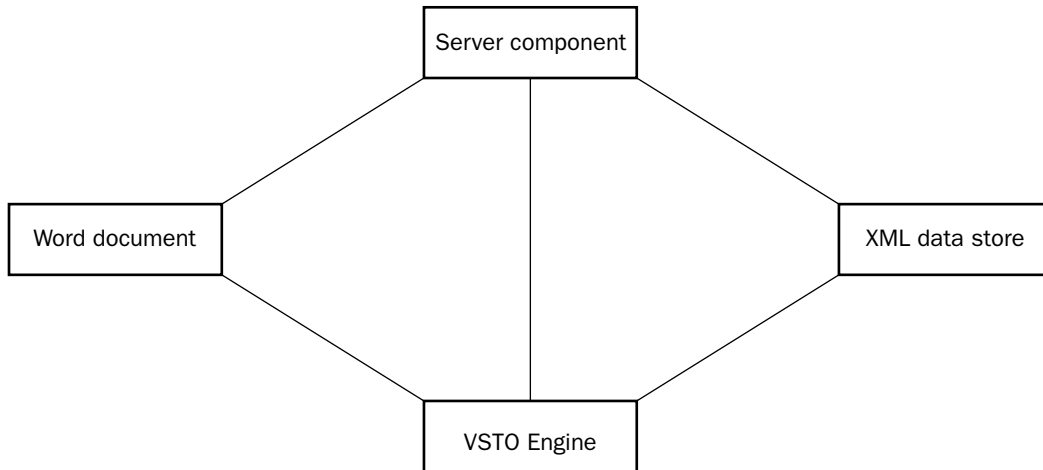


Figure 1-1

From Figure 1-1, you can see that the new architecture is flexible enough to leverage the VSTO engine for new applications based on Visual Studio Tools for Office. In other cases, the architecture allows the developer the flexibility to build applications that perform a legacy role; that is, documents may be opened without firing code housed in the .NET assembly. The new architecture also allows VSTO based applications to be written and run with embedded macros. Although this is discouraged, you should understand that there are no architectural fences that prevent such an implementation.

The VSTO Package

VSTO offers Excel, Access, Word, InfoPath, and Outlook functionality. The functionality targets the windows platform to include smart client and remoting scenarios. Although VSTO allows users to publish functionality such as spreadsheets and pivot tables on the web, the tool suite isn't specifically designed for full-blown, interactive web applications. Part of the reason lies in the fact that VSTO is primarily built for Windows-based applications.

Still, this book will explore the concept of server-side automation of Microsoft Office applications that provide ways to build code that is scalable and well-behaved when compared with other methods of building application software. Since more and more real world applications are blurring the line between desktop and web platforms, the material presented in later chapters will shed some light on these gray areas so that you can make informed choices when building new application software.

The Visual Studio Tools for Office suite also includes Visual Basic standard edition. The standard edition allows developers to write code to target the Office system. Microsoft Access and SQL server developer edition are also included. These products allow you to develop solutions based on Access and SQL server for the Office system. Finally, Microsoft also offers a separate stand-alone VSTO version that encompasses all the power and productivity of Visual Studio.

VSTO is implemented as an add-in to Visual Studio. The VSTO engine runs in-process in the Visual Studio address space. The VSTO engine depends on Office PIA's in order to communicate with Microsoft Office objects.

About Microsoft Office PIAs

Primary Interop Assemblies (PIA) are .NET wrappers built around existing COM components that allow .NET code to communicate with the COM component. Microsoft strongly recommends that Primary Interop Assemblies are to be used instead of your own auto-generated assemblies. The literature indicates that PIAs can make certain optimizations that may be beneficial to the application. While this may be true in some cases, it certainly is not always true. In fact, the dominant reason for using PIAs is that it helps combat an insidious assembly version issue in applications built with Microsoft .NET.

Microsoft .NET run-time cannot communicate directly with COM components. When communication with COM components is required, the .NET Common Language Runtime (CLR) requires an adapter or proxy to sit between the .NET assembly and the COM component. This adapter translates .NET requests into COM requests making the automation possible. It's important to understand that the proxy or Runtime Callable Wrapper (RCW) does not re-implement functionality in the COM component. It simply makes the type information inside the COM object available to the calling assembly so communication or marshalling can take place.

Visual Studio is automatically configured to generate RCWs when a reference to a legacy COM component is detected in the Integrated Development Environment (IDE). For small projects that remain in-house, these extra RCWs do not provide a cause for alarm. However, if this software must be deployed to customers, you need to package all the RCWs along with the application for it to function correctly.

Consider a scenario where one vendor makes their RCW available to a customer who already has a wrapper available for a home grown assembly of the same name. The final result will be run-time errors for the mismatched types since the assemblies used to create the wrappers are different but both of the wrappers have the same name.

To put this in perspective, consider a practical example. If company A creates an RCW for a Microsoft Outlook application for third party software called `Outlooklib` that is used by company C, the `Outlooklib` RCW must be deployed to company C. However, company C may be running another application from company B who also has an RCW for Microsoft Outlook called `Outlooklib`. This creates an ugly situation where company C's software starts to misbehave since it may be using the wrong RCW for Microsoft Outlook automation. Bugs of this caliber can be extremely difficult to resolve to say the least.

To prevent such chaos, Microsoft recommends that only one RCW be used per Microsoft Office product. This RCW or Interop assembly would be the primary source of type information for the automation call hence the term Primary Interop Assembly. In the theoretical example provided, company C's software would not misbehave since the RCW provided by company A and company B would be the same; that is, both company A and company B would be using a Microsoft Outlook PIA. Finally, note that this is a recommendation only. PIAs cannot possibly be enforced in the real world.

System Requirements

VSTO offers several benefits, but there is a cost associated with reaping the rewards of this new architecture. VSTO.NET requires the Professional version of Office 2003 Service Pack 1 on the development machine. In the absence of this requirement, the VSTO projects will not work. The requirement also makes it impossible to bind a .NET assembly to third party Excel or Office interfaces. However, Microsoft insists that applications built today based on VSTO.NET will be forward compatible with Office version 12 due out sometime in 2006. The guarantee is based in part on the fact that the data storage model is XML.

The installation packages must be installed in a specific order because there are tight dependencies between these packages. Install Microsoft Office 2003 Professional Edition first. Microsoft Office 2003 Professional Edition is an absolute requirement; VSTO 2005 cannot work with any other Office edition. Follow this install with any updates or service packs that are available. You may download and install the latest updates and service packs for Microsoft Office 2003 by visiting the Microsoft Office website. Then, install Visual Studio.NET 2005 followed by Visual Studio Tools for Office. Finally, install MSDN library so that help documentation is available. Unless you require customized features, you can simply install the products with the default settings.

The total install package can range in size from 1 to 3 gigabytes of space requirements depending on the options you choose to install. If you are challenged for space on your hard drive, you may elect to use the online help documentation instead. It is automatically configured to retrieve help documentation from the F1 short-cut key. After installing the help documentation, you should use the “check for updates” feature of the Microsoft Visual Studio 2005 installation to check for updates and patches. Normally, the entire process may take a couple of hours.

The framework must be installed first in order to create the Global Assembly Cache. Visual Studio Tools for Office uses that Global Assembly Cache to house the Primary Interop Assemblies.

You also need to have the .NET framework installed on the end-user system. While the framework offers significant advantages over legacy COM based frameworks, the requirement is a serious hindrance today. Eventually, as more end-users adopt the framework, the hindrance will fade away.

Though .NET languages are fairly numerous today, VSTO only supports C# and Visual Basic — formerly known as Visual Basic.NET. You may use either language to build applications based on Microsoft Office technology. Visual Studio Tools for Office does not currently support assemblies built from a combination of Visual Basic and C#. To be fair, this limitation is actually imposed by the Common Language Runtime. However, for those not familiar with the underpinnings of the Common Language Runtime (CLR) this lack of flexibility will be blamed on VSTO.

Of the two programming languages, Visual Basic is better for building VSTO applications. There are several reasons for this due in part to the history of the Microsoft Excel and Word components.

C# has no support for optional parameters in method calls. However, the objects that form the core part of the Office system are built on COM and tuned for VBA code. These API's expect optional parameters. Because of the lack of support in C#, special place holders must be passed in for each call. Using these placeholders for each method call can be tedious and demanding. For instance, the document's open method accepts 15 optional parameters. Consider a Visual Basic example.

```
Me.CheckSpelling()
```

Now consider a C# example.

```
this.CheckSpelling(ref missing, ref missing, ref missing,  
    ref missing, ref missing, ref missing, ref missing,  
    ref missing, ref missing, ref missing, ref missing);
```

The PIA's that are installed during VSTO setup enforce a pass by reference semantic on the caller for Microsoft Word development. For C#, the `ref` keyword must be included to satisfy the requirement or a compilation error will be generated. There is no workaround other than to add the `ref` keyword. The arguments also must be variable references. Literals or strings cannot be passed in because these do not satisfy the compiler requirement.

C# does not allow multiple arguments to properties. However, the Excel object model supports such expressions. This is not a problem for Visual Basic because the support has been continued for historic reasons. Range manipulation in particular makes heavy use of multiple arguments to properties. In such instances, you can use the optional methods provided by the Excel PIA in C#.

Visual Basic

```
ThisApplication.Range("A1")
```

C#

```
ThisApplication.get_Range("A1");
```

Although the PIA's have been updated to add substitute properties for C# developers, these properties often do not show up in the intellisense engine of Microsoft Visual Studio.NET 2005. Without solid documentation, C# developers are left holding the short end of the stick.

If you care for a vigorous exercise in typing, you may choose to develop VSTO-based applications using C#. However, if you don't particularly care for the exercise or are running on a lean productivity budget, Visual Basic is the most appropriate choice. This book will present code in both languages for the benefit of all.

Alternatives to the VSTO Office Systems

As mentioned before, VSTO is not the only approach to harnessing the power of Microsoft Office technology in next generation application software, but it offers significant advantages over other approaches as we shall see. In fact, some of these existing approaches have only been tolerated because there was no suitable alternative available to justify the expense of adopting a new approach to building Office applications. Today, with VSTO, the expense of adopting a new approach can be justified in terms improved security, scalability, and reliability.

VBA

Visual Basic for Applications (VBA) was the de facto standard for building applications based on Microsoft Office technology. VBA made it easy to build Office technology because the user interface could be designed and built easily. Underneath the hood, VBA connected directly to the Office API infrastructure. Developers could quickly build and release Office based application software using VBA.

Chapter 1

However, one major drawback was that security was an afterthought in the VBA approach. VBA macros can run embedded inside a document, often underneath the security blanket of the application. Executing code cannot usually determine the origin of the macros. For instance, the macro can be embedded by the program or it may be embedded by malicious code.

VBA falls underneath the umbrella of unmanaged code. Unmanaged code extracts a performance penalty when interacting with managed code. VBA also cannot lessen this performance hit by routing calls through a Run-time Callable Wrapper, so the performance penalty is dead weight.

VBA is also a procedural programming language. It cannot take advantage of some of the niceties such as inheritance, interfaces, and polymorphism. These terms aren't just buzzwords, they enable the creation of large, scalable application software.

VBA exposes arcane approaches to error handling. VSTO uses the mature exception handling mechanism in .NET. The transition from clunky error handling constructs in VBA to a polished exception handling surface in .NET enables applications to scale properly and remain well-behaved in abnormal circumstances. For instance, exceptions that are thrown in a remoting call contain the full context of the exception. A VBA implementation would typically involve reading an error code when some exception occurred.

VBA does provide certain advantages that VSTO cannot provide. For instance, VBA allows code to be executed in worksheet functions. VSTO is unable to perform this task. The VBA Office model is more mature than VSTO. Maturity offers certain benefits such as better documentation, developer familiarity and end-user acceptance that tend to make development in VBA a more natural choice than VSTO. VSTO has not yet achieved that comfort level because it is the new kid on the block.

Another important advantage of VBA is that VBA is available on virtually every machine running Microsoft Office 97 or later. VSTO requires installation of some key infrastructure pieces, such as the .NET Framework.

Office Web Components

The Office web components (OWC) may be used both on the desktop and on the web. They exactly mirror the fit and finish of Microsoft Excel. However, this functionality comes at a price. The components are more strictly licensed than VSTO and are only intended to be used in intranet web application scenarios. A suitable book to help you negotiate the pitfalls of Office development for the web is my recent book *The Microsoft Office Web Components Black Book with .NET*.

The OWC is based on unmanaged code so the .NET framework is not a requirement. It may be downloaded for free and installed on demand. The OWC is light enough to fit inside a browser as an ActiveX control. The components provide the full power and functionality of Microsoft Excel. Other Microsoft Office functionality is not provided.

Excel COM Interop Libraries

COM Interop packages are inherently difficult to program and require deep knowledge of COM implementation. These COM libraries do not always play well together and present all sorts of issues when scalability increases. The COM libraries are limited in the functionality they expose when compared to VSTO and are notably weak in exposing the event model of Excel and Word.

One serious issue with Excel COM automation is that it possesses serious problems in concurrent environments. In fact, instability in these environments most often leads to application failure, memory exhaustion issues, and very unpleasant end-user experiences.

Third Party Products

A number of independent software vendors make products that target Excel and Word. The common theme of these third party application software addresses the short fall of what is available to the customer through Microsoft channels. Usually, these products are not free, contain licensing restrictions, and have shaky support that limits the use and effectiveness of these products in industry.

On the other hand, some third party products such as SoftArtisan's Excel writer are very stable and mature, have a wide customer base, and perform well under load. If you are considering adopting an Office technology, it is important to become familiar with the choices that are available. Build some test applications based on these products and expose them to a variety of real world conditions such as load and stress to see how well they perform before deciding on one product over another.

Disadvantages of VSTO

As with any sizeable piece of real world software, there are advantages and disadvantages. The following sections describe in detail the major drawbacks of VSTO. This is not a complete list. But it represents the thorniest issues that decision makers must understand before making a final decision on adopting a new technology.

.NET Framework Required

One major drawback to VSTO is that it requires the .NET framework to execute. In most corporate environments, end-users simply cannot install application software, much less run-time frameworks. This requirement alone puts VSTO-based applications at a serious disadvantage. In fact, it practically rules out off-the-shelf applications based on VSTO.

Independent Software Vendors are more likely to build application software based on one of the alternatives presented previously simply because these applications are light and are guaranteed to work on Microsoft Windows systems. Even with the framework packaged along with the software, end-users may be skeptical about installing it due to the size of the .NET framework.

Security

VSTO does not implement security natively. Instead, it out-sources its security interests to the .NET framework. As usual, out-sourcing presents a number of obstacles. For the average developer, this is more than a passing inconvenience because some detailed knowledge about the security aspects of the .NET framework is required. This represents another hurdle in the learning process. Code Access Security and its related subtopics that help define .NET security are sufficiently complicated to discourage even some seasoned developers from adopting VSTO as an Office development technology.

The need for security cannot be overstated. The idea is always to prevent malicious applications from subverting the integrity of the user system. Security policies in effect may also protect the user system from misbehaving applications. Web applications and smart clients in particular should be guaranteed

to execute in a controlled manner since the user cannot always guarantee that the application is friendly. If you intend to secure your VSTO applications, it is worth your while to invest some time and effort in understanding .NET security. If you must build real world software based on VSTO, you will need to learn about .NET security.

Performance

Much of the hype around VSTO includes claims that VSTO out-performs VBA and COM approaches. That argument is without merit! While performance is not a serious cause for concern, the performance of a VSTO application does lag behind VBA and COM approaches. There are several factors that influence VSTO performance, and it is important to put these factors in perspective if you intend to draw any meaningful conclusion

The .NET start up cost is inherently expensive. Applications written with .NET must incur the overhead of Just-In-Time (JIT) compilation. JIT compilation is a necessary evil and cannot be avoided. However, steps may be taken to reduce the performance expense of JIT compilation.

One approach is to use the `NGEN.exe` application that ships with the .NET framework. This utility forces a JIT compilation of all the methods in the assembly. Since the methods are Jitted, JIT compilation is no longer necessary when the application first starts up. There are several disadvantages to using NGEN. For one, NGEN files can get out-of-sync. Another issue is that the JIT code is no longer optimized for the underlying platform, since it is done offline. A third issue is that NGEN offers no performance gains in server application code.

Another performance factor influencing VSTO-based applications has to do with the expense of calling through the thick layers of automation skin that wrap the Microsoft Office COM objects. VBA, built and optimized to interact with Microsoft Office, has a shorter distance to travel than .NET.

Finally, hosting the Excel and Word objects in the Visual Studio IDE is expensive in terms of resources. VSTO applications have a larger memory footprint than VBA applications. This resource expense bleeds into the application run-time in noticeable ways.

In essence, if you are considering adopting VSTO as a development platform for Microsoft Office technology, it is worth your while both in time and resources to become familiar with the advantages, disadvantages, and idiosyncrasies of this relatively new technology. Otherwise, the investment may not be worth it.

VSTO Automation

Strictly speaking, automation refers to the process of controlling one object with another. In that regard, VSTO is an automation controller whose primary function is to automate the objects that form part of the Microsoft Office System.

While VSTO is mainly concerned with automating Office objects, it is not necessarily limited to just those objects. Automation can occur with any object as long as it exposes the required automation interface. For instance, it is quite feasible for a VSTO based application to consume components developed by third party vendors. However, VSTO automation is particularly well-suited to objects that belong under

the umbrella term of Office products. Usually, these objects contain internal optimizations that make them particularly well-disposed to VSTO automation. Unfortunately, products such as Microsoft Outlook Express are not part of the Microsoft Office System and contain no internal support for VSTO.

Products such as Microsoft Access and InfoPath are supported in VSTO as Add-ins. Add-ins can take advantage of special internal pathways that offer some gains in performance over regular COM approaches. However, add-ins are not first-class VSTO citizens.

Automation problems have been addressed in VSTO but there are still steps that need to be taken to ensure that an automation environment is well-behaved. The dominant reason for these extra steps has to do with the fact that the underlying Microsoft Office legacy code is still based on a COM platform. Since these memory models and resource allocations differ dramatically from managed environments, you need to make sure that resource de-allocation has occurred. Later chapters will show exactly how to build these checks and balances into your code approaches.

Office XML Schemas

When developing applications for the internet, Microsoft Excel or Office is not needed on the web application server. The reason for this magic lies in the divorced document/view architecture. This separation, amicable as divorces go, actually allows processing code to operate on the data contained in the document while leaving the view untouched. It is this clever separation that provides opportunities for scaling out applications based on VSTO.

To further enhance application development, Microsoft has released the XML reference schema for Office applications. This release makes the Office XML available to developers. The implication here is that developers are now free to create customized XML documents that are guaranteed compatible with Microsoft Office applications.

In case you missed it, this is a very huge step forward coming from Redmond. Don't get it confused with generosity though, it is a calculated move to ensure the dominance of Microsoft Office products and the stifling of external competition from open-source initiatives nipping at the Office products heels.

The Microsoft Office XML schema is available for Word, Excel, and Infopath as WordprocessingML, SpreadsheetML, and FormTemplateML, respectively. There is obvious merit to this initiative. It means that client software can now read and write data in a platform independent way taking full use of the advantages and platform affinity of the XML standard. Microsoft hopes that this initiative will have broad adoption releasing developers from the tedium of customized storage access for Office integration.

Installation and deployment

Before installing VSTO, ensure that you have first installed the .NET framework, followed by Visual Studio.NET in that specific order. The following sections show how to install the tools suite on your system.

1. From the install media containing Visual Studio Tools for Office Microsoft System, run the setup application in the root folder, as shown in Figure 1-2.
2. Choose the installation option to install VSTO.

3. Examine the installation log for possible setup problems that occurred during the installation process.
4. Open the Visual Studio Integrated Development Environment.
5. Examine the project types window pane in the left window to make certain that the VSTO templates are available.



Figure 1-2

Visual Studio Tools for Office depends on the Microsoft Office Primary Interop Assemblies (PIA). During a typical installation, the Office PIA's are automatically installed. However, if you have chosen a custom installation, you may need to manually install the Microsoft Office PIA's.

Follow these few steps to install the necessary PIA's on your system.

1. Run the setup file for the Primary Interop Assembly package.
2. Select the programmability support that you require, as shown in Figure 1-3. The Language Tools node contains the currently supported languages that may be used to build Office applications. The Dotfuscator Community Edition and Remote Debugging selections are optional. The .NET Framework SDK provides tools for working with the .NET framework. Although it is optional, you should install it.
3. Click OK to install the PIA.

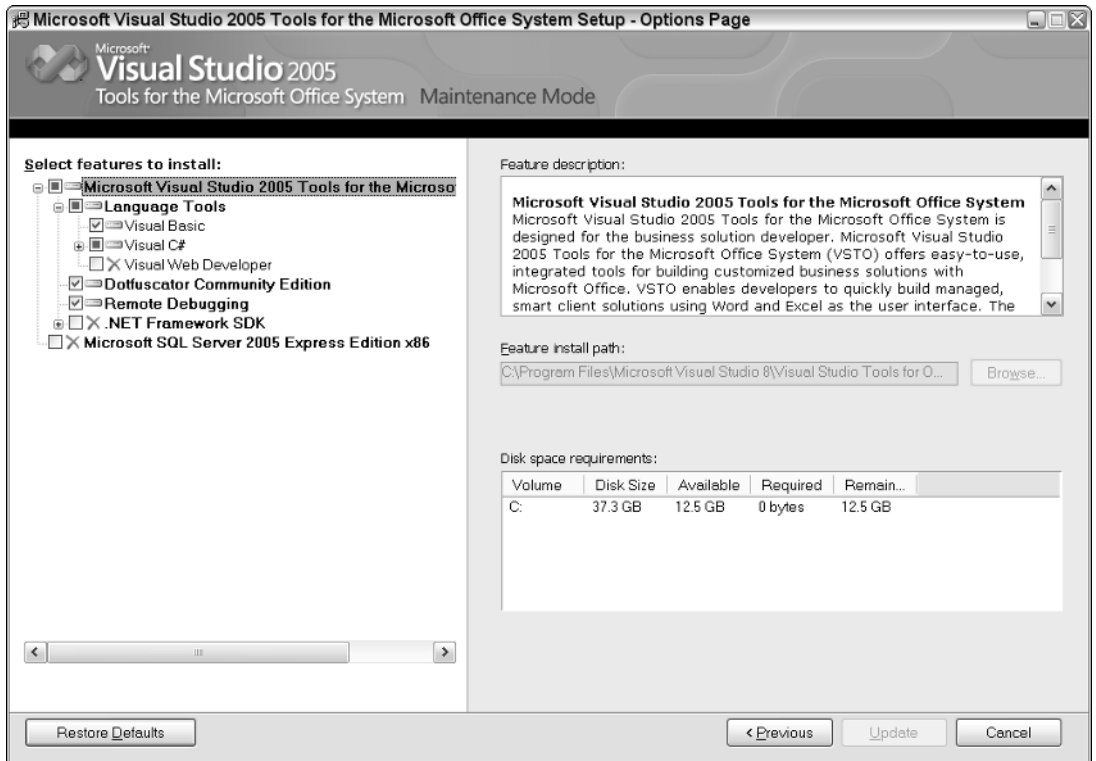


Figure 1-3

Creating VSTO Projects

Once Visual Studio Tools for Office is installed correctly, you may create a new project based on VSTO. Follow these few steps to install the necessary PIA's on your system.

1. Open Visual Studio 2005.
2. From the File menu, create a new project. The action launches the Microsoft Visual Studio Project wizard.
3. In the project types window pane, choose Office from the language tree. Notice that Visual Studio has tailored your environment to your language preference. For instance, if your language preference is C#, as in Figure 1-4, the first language will be Visual C#. Otherwise it will be Visual Basic.

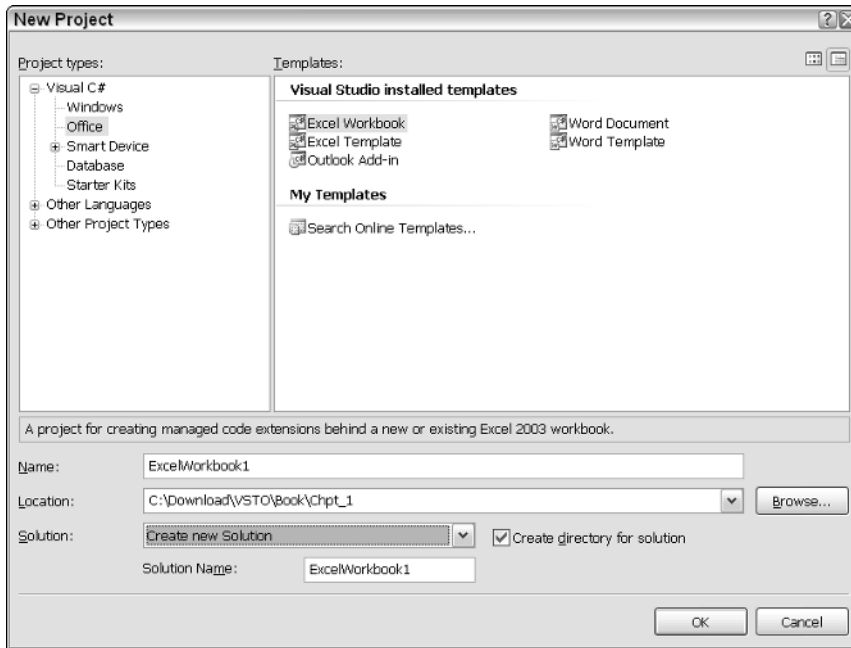


Figure 1-4

4. In the right window pane labeled Templates, select Excel workbook. You may choose to accept the default name at the bottom of the window – ExcelWorkbook1, or you may add a more meaningful name. In any event, the project will be assigned the name from the Name property text box in the wizard and the project will be created in the Visual Studio Project folder. Notice the checkbox off to the right. It allows you to keep projects in their own directory.
5. Click OK to generate a new project. Behind the scenes, VSTO first verifies that the minimum system requirements are at least present. Then, the internal plumbing is laid for the new project. If there is no misstep in the process, the wizard generates a project template and opens to a Microsoft Excel spreadsheet nested inside the Integrated Development Environment, as shown in Figure 1-5. If you had selected a word template, the application would open to a Microsoft Word document hosted inside the Integrated Development Environment.

Notice that the templates section contains the possible templates that are available in VSTO. It does not mean that these templates are configured on your system. For instance, if you do not have Microsoft Outlook professional edition installed, the template will still be available. However, it will fail to open. The purpose of the template is two-fold. It allows .NET code to automate the Excel object and it allows .NET code to be linked to the Microsoft Office document.

On the next page of the wizard, you may choose to create a new document or to copy an existing document, as shown in Figure 1-6. Click OK to complete the wizard. Notice that the name of the document in Figure 1-5 is used to title the solution in Figure 1-6. Also, the default name is used for the Excel workbook. The property window on the far right now contains a single workbook project made up of properties, references, and a workbook containing three Excel worksheets. From this point, you are ready to develop.

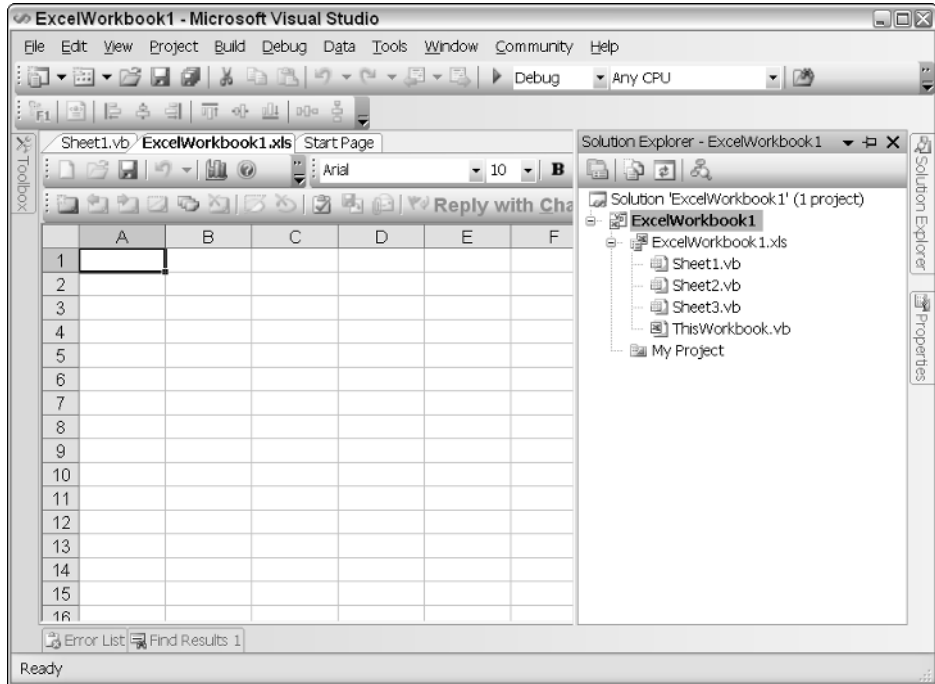


Figure 1-5

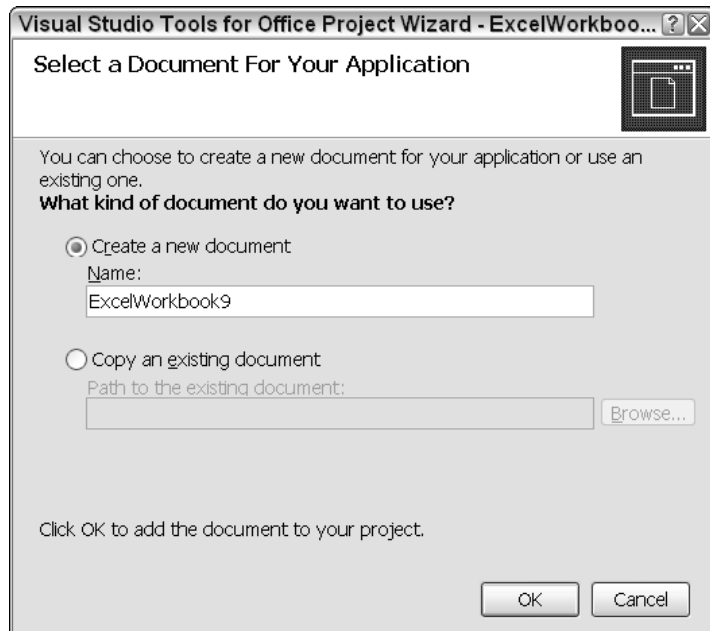


Figure 1-6

VSTO Installation Issues

In some instances, VSTO installations may be problematic. This section contains some tips for figuring out how to repair broken installations.

When you run VSTO for the first time, a security dialog box appears informing you to explicitly enable access to the Microsoft Office Visual Basic for Applications project. While this isn't an installation problem per se, selecting Cancel will cause VSTO projects to fail. Simply click OK to dismiss the dialog. Access to the Microsoft Office VBA project system is required for VSTO to run correctly. If you choose Cancel, you will need to re-open the project again so that the option can be corrected.

In certain instances, the Microsoft Office Primary Interop Assemblies may fail to install correctly. The remedy is to manually re-install the PIA's from the installation media. Locate the `PiaInstall.htm` file on the installation media. This file opens in a web browser and provides instructions for PIA installation, as shown in Figure 1-7. The `PiaInstall.htm` file resides a few folders deep in the install media. Figure 1-7 shows a typical install path `F:\vs\setup`.

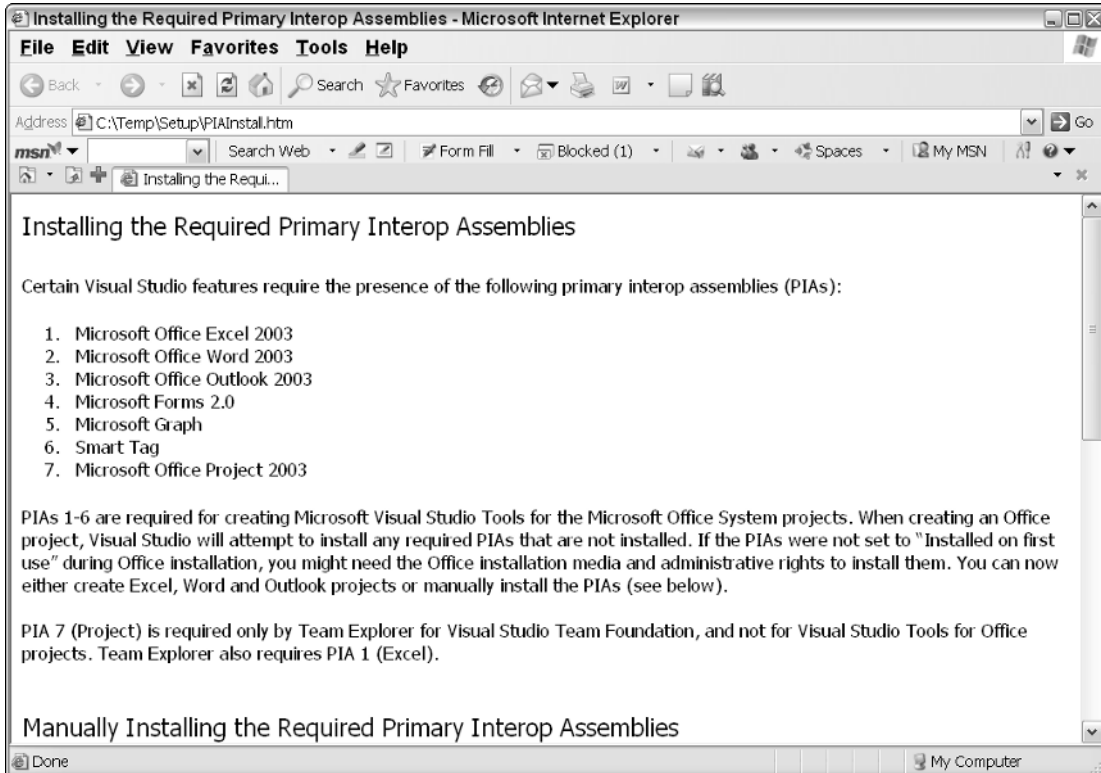


Figure 1-7

The Office PIA's are not COM components, so you cannot manually register them at a command prompt using the Windows `regsvr32` executable. However, they do incorporate COM Callable Wrappers (CCW) that allow you to call them using a COM interface. To do this, you must register the CCW interfaces using the `regasm` utility that ships with the .NET framework.

In some versions, the application may install without errors but fail to create Excel projects. To remedy this annoyance, make sure the following key is accessible in the windows registry. If it is not present, back up the registry and perform the following steps.

1. Navigate to the following registry hive.

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\8.0\Setup\VSTO]
```

2. Add the following key

```
String Value  
Name - ProductDir  
Data - C:\Program Files\Microsoft Visual Studio 8\Visual Studio Tools for  
Office
```

The Path in the Data field must point to your actual Visual Studio Tools for Office folder. Use Internet Explore to find the path to your Visual Studio Tools for Office folder. For more help editing the windows registry, please consult the Microsoft Windows help system.

A list of known issues is summarized in `VSknownIssues.rtm`. This file may be found on the installation media. Be sure to check the Visual Studio setup log for any installation failures. Issues that arise must be addressed in order for VSTO to function correctly.

The most recent update to VSTO ships with Visual Studio 2005. As of this writing, there are no patches or upgrades for the tool suite. However, you should always ensure that Microsoft Office is current and up to date by using the Windows Update site. You can find more information about Microsoft Office updates by visiting the Microsoft Office website or MSDN.

Before re-installing Visual Studio.NET to repair broken installations, run the `Vstor.exe` application manually. This application can be found on the installation media of Visual Studio. The application fixes a number of pesky issues inherent in the installation process.

Summary

While it is certainly romantic to think that VSTO was born out of innovative technology, in reality VSTO was born out of a need to stem the tide of open source competitors like OpenOffice and the like whose open source product offerings steadily erode the bottom line of Microsoft Office profitability. Even so, VSTO does bring a lot to the table where Office development is concerned.

VSTO adds productivity because applications based on the Microsoft Office System can tap into the vast reservoir of the .NET framework. Though it is still possible to accomplish all this with VBA and COM Interop, VSTO makes these tasks a lot easier. For instance, incorporating web-services into VSTO applications is a snap. The end result of these and other features really make it possible to develop applications that span corporate boundaries in a safe and secure way. VSTO is focused on increasing developer productivity.

Chapter 1

The language support for Visual Studio is also in its infancy; that is, only Visual Basic and C# are supported with Visual Basic being the language of choice for historical reasons. However, both these languages are powerful enough to build commercial-grade software. The intellisense engine in both C# and Visual Basic offer full support for VSTO.

We also covered the unpolished side of VSTO to include performance concerns. While this is certainly not a popular route and is less likely to be traveled in other resources, it is very necessary if you are to form an educated opinion of Visual Studio Tools for Office. You may recall that some of these performance problems have nothing to do with VSTO at all. In fact, much of the startup costs have to do with the .NET JIT compilation process. However, .NET and VSTO are tied together so that VSTO performance perception is influenced negatively through guilt by association.

Much of the power and flexibility of VSTO comes from the new architecture model. If you have ever spent time developing server-side applications, you will recall that these applications failed to scale well because, as load increased, the COM components started to misbehave. Part of the reason for this malfeasance has to do with the inter-twining of logic and presentation code. The new model offers a new approach so that data can be manipulated without the need to touch the code that presents the data. The implication here is that Excel is no longer needed on the web-server for document processing. And that, by itself, is one compelling reason to make the switch to VSTO.

Whether or not you have developed software for the Microsoft Office System, VSTO is a very appealing choice. Developer productivity is increased and more functionality is available through the .NET interface. VSTO is also well-designed and implemented. The end result is a very well-behaved piece of machinery that delivers Microsoft Office technology in an effective way while reducing investments in time and effort. VSTO has a long way to go, but its first few steps are well-guided and deserve applause.

2

Excel Automation

VSTO ships with an Excel object that contains a user interface and a component object. Both pieces have been repackaged for .NET. The rewrite provides tighter integration and a more robust model that resolves some of the stubborn issues that were prevalent in the earlier version of the tool suite. The new model exposes the Excel user interface object hosted inside the Visual Studio IDE. The user interface is able to respond to design-time manipulation and customization. It is also able to load and display data in design mode.

This chapter will focus on walking through Excel data loads using various data sources. The material in the following sections will show how to manipulate data using the various objects designed for manipulating data. Once you have these basic concepts, you will be able to put a bare-bones application together.

To help you learn more about the Excel model, a large portion of this chapter is dedicated to Excel manipulation to include worksheet and workbook customizations. Finally, a case study is presented to piece together the concepts so that you can build value-added software.

Excel Data Manipulation

Perhaps the most well-known functionality of Microsoft Excel is its ability to manipulate data from a wide variety of sources across different environments. For instance, the knowledge worker may use Excel's processing power to analyze large amounts of data. Decision makers may use the spreadsheet to analyze business trends, while information technology workers may use the spreadsheet to track hardware over different departments.

Although these tasks are very different, Microsoft Excel is able to perform the tasks easily and efficiently in a way that puts the end user in the control seat. Applications based on VSTO can leverage this power and flexibility, too.

In order to work with the data, Excel needs to be able to load the data first. Excel is able to load data from a wide variety of data sources. A short list of these potential data sources are Internet resources; most databases such as SQL, MYSQL, and Oracle; any type of database that exposes the

Open Database Connectivity (ODBC) interface; XML files; most Windows file types (.txt, .csv, .doc, etc.) as well as user-defined custom file types. Excel can load this data at runtime or at design time. The next section examines design-time data loads.

Design-Time Data Loads

The design-time data load is a first measure for protecting your code against nasty surprises. It is not a substitute for building robust code, but it certainly goes a long way toward providing application stability. Consider this example, where data needs to be loaded into the Excel spreadsheet at design time. Figure 2-1 shows that an XML data source is being selected from the Visual Studio main menu bar. This action opens a dialog box enabling the user to choose an XML file. VSTO will automatically create an XML schema for the file if it does not exist. The right window pane shows the solution with the newly created schema *.xsd at the bottom.

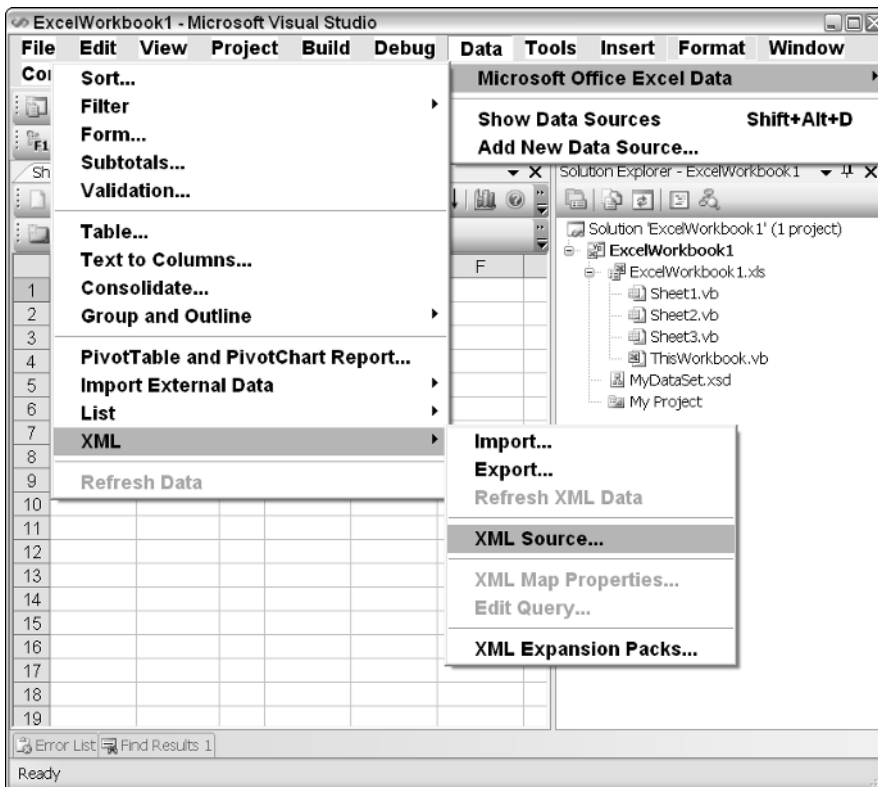


Figure 2-1

If you would like to load other file formats at design time, you should add the file format to the project as a text document using the Add new item selection from the Project menu.

Once the data is loaded into the Visual Studio IDE, the user or developer can inspect and manipulate the data to produce charts or reports. Consult the Excel help documentation if you require more information on Excel data manipulation features.

VSTO is able to merge Excel menu bars with Visual Studio IDE menu bars. One example of this occurs when using the XML import feature. If you try to perform the data load at runtime instead of design time, you should notice that the XML import feature described previously and located in the Data ⇄ XML menu is moved to Data ⇄ Microsoft Office Excel Data of the menu bar during runtime. Although the menu orientation and position has been adjusted by VSTO, the functionality remains the same.

Loading Files at Runtime

VSTO supports file loading at runtime internally and through the .NET Framework. File loads that use the .NET Framework approach depend on the `system.io.filestream` object to facilitate the data load. The `System.IO` assembly must be imported into a Visual Studio project in order for the project to compile. Once the files are read, the contents may be stored in a variable and assigned to the cells in the spreadsheet.

File Loads Using .NET

As noted previously, Excel is able to load files using the classes exposed through the .NET Framework. Consider this example that loads a comma-delimited file. To run the following snippets of code, first create a VSTO project. If you develop in Visual Basic, the Visual Basic code must be placed in `sheet1.vb`. The `System.IO` namespace needs to be imported as well so that the solution will compile correctly. Listing 2-1 shows code that reads from a file called `csvdata.txt` on the user's local hard drive.

In case you are wondering, the data in `csvdata.txt` file is test data. For your test case, you can add any piece of text and/or numerical data on multiple lines. Using commas to separate the data in the text is optional. VSTO will load the data successfully with or without delimiters.

Visual Basic

```
Public Class Sheet1
    Private Sub FileProcessor(ByVal fileName As String)
        Dim fs As FileStream = New FileStream(fileName, FileMode.Open,
        FileAccess.Read, FileShare.Read)
        'Process file here
        fs.Close()
    End Sub
    Private Sub New_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
        FileProcessor("c:\download\csvdata.txt")
    End Sub

    Private Sub New_Shutdown(ByVal sender As Object, ByVal e As System.EventArgs)
    End Sub
End Class
```

C#

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
```

```
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;
using System.IO;
using System.Text;

namespace Chpt2_Listing2_2
{
    public partial class Sheet1
    {
        private void FileProcessor(string fileName)
        {
            FileStream fs = new FileStream(fileName, FileMode.Open,
            FileAccess.Read, FileShare.Read);
            //process file here
            fs.Close();
        }

        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            FileProcessor(@"c:\download\csvdata.txt");
        }

        private void Sheet1_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}
```

Listing 2-1 File load using .NET

When a VSTO-based application starts up, and sometime after the initialization event has occurred, the startup event for the sheets object is fired. By placing code in the startup event handler, the VSTO runtime guarantees that the sheet object will be ready to service client requests. In the code example in Listing 2-2, the `FileProcessor` method is used to open a file on disk in read mode.

This piece of code is not expected to win any award for elegance; however, it does illustrate a few important concepts that warrant further explanation. There is no exception handling in the code. The code simply assumes that the file exists and is readable. Such assumptions may easily be violated in real-world software. The example is without this safety net for illustrative purposes only.

Some of the code snippets in this book are intentionally kept simple to illustrate the concepts clearly. Once you have a basic understanding of the concepts, you should be able to implement any requirement in code.

The code also assumes that the executing process has the necessary permissions to read a file from disk. By default, the executing process only has permission to read files in the application directory. Although the file load may occur without exception on the development machine, it will fail when the application is deployed if that file is located outside the application directory structure. Permissions configuration will be examined in the Chapter 3.

The code in Listing 2-2 also acts as a test harness of sorts. All code snippets to follow can be copied to this test harness with the appropriate assembly imports to work correctly. Future code snippets will mostly show the important pieces of the code for illustrative purposes. Code generated by Visual Studio will be removed to prevent clutter. Also, exception-handling code will normally not be present in the coded examples.

File Loads Using VSTO

The `FileStream` object is handy in the above scenario, but the approach is fairly unappealing because VSTO implements its own file loading functions, which are considerably more elegant and a lot richer than the meager file load implementation that the .NET Framework provides. Remember, .NET code must be generic in nature to service a wide variety of applications; however, VSTO code is optimized to take advantage of the fact that the application being built is an Office-based application. Consider the code in Listing 2-2.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Application.ScreenUpdating = False
    Application.Workbooks.Open("C:\Download\CSVData.txt", Delimiter:=",",
    Editable:=True, AddToMru:=True)
    Application.ScreenUpdating = True
End Sub
```

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
    Application.ScreenUpdating = false;
    Application.Workbooks.Open(@"C:\Download\CSVData.txt",
    System.Type.Missing, false, System.Type.Missing, System.Type.Missing,
    System.Type.Missing, System.Type.Missing, System.Type.Missing, ",", true,
    System.Type.Missing, System.Type.Missing, true, System.Type.Missing,
    System.Type.Missing);
    Application.ScreenUpdating = true;
}
```

Listing 2-2

From the code presented in Listing 2-2, the `Sheet1_Startup` routine causes the workbook's `Open` method to be called. The application object, a global object that stores references to each user interface of the Excel spreadsheet, provides a convenient hook to the `Workbooks` object. The `Open` method takes a file path and a few optional parameters. As stated before, most file types are supported. If Excel does not recognize the file type, a dialog is displayed to allow the user to choose a file-handling format. If the user fails to choose a format or cancels the dialog, the data is loaded as plain text into the document.

Notice the call to `ScreenUpdating`. This call turns off the screen-updating feature of Microsoft Excel so that each row of data will not be displayed in the Excel spreadsheet as it is loaded. The option saves processing power for the resource-intensive rendering process, while providing a significant boost in performance. It is important to remember to set this property back to a value of `true`; otherwise, the spreadsheet will not be updated.

Chapter 2

The `Open` method is powerful enough to load files from different locations. Listing 2-2 summarizes these locations with examples. To run these examples, simply replace the file path in the code snippet in Listing 2-3 with the appropriate selection from the table.

Source	Example
Absolute file path	<code>C:\Download\CSVData.txt</code>
Relative file path	<code>\myDefaultCSVData.txt</code>
UNC	<code>\\Home\Download\CSVData.txt</code>
HTTP file path	<code>http://somedomain.com/CSVData.txt</code>
HTTPS file path	<code>https://somedomain.com/CSVData.txt</code>

For the relative file path option presented in Listing 2-2, the application directory is assumed and its path is concatenated to the relative file path during application execution. For instance, the full path for a VSTO application may look like this:

```
C:\Documents and Settings\abrune\My Documents\Visual Studio 2005\Projects\Excel Workbook2\myDefaultCSVData.txt
```

Excel also exposes the `OpenText` method that provides a greater degree of control in the file load. For instance, consider the code snippet in Listing 2-3. The code assumes that a website is available and configured correctly to point to the Download domain.

Visual Basic

```
private void Sheet1_Startup(object sender, System.EventArgs e)
me.Application.Workbooks.OpenText("http://Download/CSVData2.txt",
Excel.XlPlatform.xlMacintosh,3)
end sub
```

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
this.Application.Workbooks.OpenText("http://Download/CSVData2.txt",
Excel.XlPlatform.xlMacintosh, 3,
Excel.XlTextParsingType.xlDelimited,Excel.XlTextQualifier.xlTextQualifierNone,
missing, missing, missing, false, missing, missing, missing, missing, missing,
missing, missing, missing, missing);
}
```

Listing 2-3 `OpenText` method

The observant reader should realize a few differences here. The `Type.Missing` variable has been replaced with `missing`. It isn't groundbreaking, but C# programmers will appreciate the marginal typing gains. The second parameter tells Excel to expect a file load from a Macintosh platform. Although that type of functionality is not yet supported — the CLR only supports Windows platforms — it's good to see that it's included today. That type of functionality may come in handy when developing cross-platform applications.

At some point in the future, VSTO applications may run on the Mono framework. The Mono framework is built to run on non-Windows platforms. You can read more about Mono on www.mono-project.com.

The third parameter tells the workbook to open the file and begin processing on row number 3. Little niceties such as these actually add up to significant performance boosts because extra code does not have to be written to parse the contents of the file from a specific row.

Miscellaneous Data Loads Using VSTO

The `QueryTable` object is another potent option for loading data. It is able to load data from a wide variety of sources to include database tables that support the Open DataBase Connectivity (ODBC) standard, Hypertext Markup Language (HTML) tables inside classic Office documents, and data on web pages. The code snippet in Listing 2-4 shows how a `QueryTable` object can be used to retrieve data from an Internet resource.

Visual Basic

```
With Me.QueryTables.Add(Connection:="URL;http://edition.cnn.com/WORLD/ ",
    Destination:=Range("A1"))
    .BackgroundQuery = True
    .TablesOnlyFromHTML = True
    .Refresh(BackgroundQuery:=False)
    .SaveData = False
End With
```

C#

```
object async = false;
Excel.Range rngDestination = this.Application.get_Range("A1", missing);
object connection = "URL;http://edition.cnn.com/WORLD/ ";
Excel.QueryTable tblQuery = this.QueryTables.Add(connection,
    rngDestination,missing);
tblQuery.BackgroundQuery = true;
tblQuery.TablesOnlyFromHTML = true;
tblQuery.Refresh(async);
tblQuery.SaveData = true;
```

Listing 2-4 QueryTable database load

The code in Listing 2-4 demonstrates how to add a `QueryTable` object to the `QueryTable` collection. The `QueryTable` object contains a `connection` parameter pointing to an Internet resource. The `QueryTable` also contains a `destination` parameter that indicates the position on the spreadsheet where the data must be placed. By adding the `QueryTable` object to the `QueryTable` collection, the data is loaded into the Excel spreadsheet.

You should notice that there are a several optional parameters that may be customized, depending on application requirements. A few important ones are shown in Listing 2-4. The `QueryTable` object is flexible enough to fetch data asynchronously from a resource. This is all implemented internally, so you do not have to write application code that is thread-aware or hooked to events to process the data efficiently. The `QueryTable` can also target certain types of data on the Internet resource page. For instance, Listing 2-4 returns data that forms part of an HTML table. You should note also that the data processing can only handle plain text. Pictures and embedded objects are simply ignored. You can see the difference more clearly if you open an Internet page to <http://edition.cnn.com/WORLD>. If you follow this link, you will notice that there are quite a lot of images and links on the page that aren't part of the `QueryTable` object.

The `QueryTable` object works by processing data requests through an internal Web Query object. You can think of this object as a spruced-up web service that is internally implemented. There are several options that are available to calling code to allow for precise extraction and manipulation of the resource. For instance, you can specify exactly which table that data is pulled from. You can also parse the data being extracted so that it appears formatted in the Excel spreadsheet. Finally, another nifty option allows the spreadsheet to automatically update the data from the Internet resource when the Excel spreadsheet is opened. To use these features, just set the appropriate parameters in the `QueryTables.Add` method call. For a full description of these parameters, consult the help documentation.

Where possible, you should prefer the `QueryTable` approach to extracting data from Internet resources as opposed to building web services, since there is more functionality available. You should consider using web services when the data you are consuming is already supplied through a web service. Web services are also a better option if your consuming code is not client based.

Finally, you should note that the `QueryTable` object also supports HyperText Transfer Protocol (HTTP) GET and POST method calls. `QueryTables` may also be used to open database connections to data sources that support the ODBC interface. You simply need to provide the connection information and the location of the data source and the `QueryTable` will load the data. A sample implementation is trivial and best left as an implementation exercise to the reader.

XML File Loads

The business world is awash with XML excitement. The excitement is justified because XML can expose data in a platform-independent way. Businesses today are scrambling to expose services through interfaces such as web services and .NET remoting. Both of these services are able to transfer data using XML. VSTO contains improved support for XML data. The support makes it easy to wire VSTO based applications to business infrastructure. Consider the code in Listing 2-5, which loads an XML file from disk.

Visual Basic

```
Me.Application.Workbooks.OpenXML("C:\Download\Test.xml")
```

C#

```
this.Application.Workbooks.OpenXML(@"C:\Download\Test.xml", missing, missing);
```

Listing 2-5 OpenXML method example

XML is a convenient format for cross-platform applications, but this freedom extracts a price. XML files are heavy when compared to other file formats. The extra weight is a result of the schema and formatting instructions that accompany the raw data. That kind of price becomes noticeable in applications that must service requests remotely. In addition, the internal resources used to process XML file loads may be taxed heavily, depending on the size of the file.

You should note that the `OpenXML` method call can easily support an Internet resource file load. The implication here is that the file load can occur locally or on some remote file server located halfway around the world.

ADO.NET Data Loads

VSTO.NET contains no internal support for ADO.NET datasets. The workaround involves writing code to manually loop through the dataset assigning values to cells. The code in the next example assumes that data from the dataset has already been pulled into an ADO.NET dataset object. In case you do not have a database connection or don't particularly care to retrieve data from a data source, you may use the code in Listing 2-6 to build and return a dataset before continuing with the example.

Visual Basic

```

Private Function GetDataSet() As System.Data.DataSet
    Dim ds As DataSet = New DataSet("MyDataSet")
    Dim myDataTable As DataTable = New DataTable("My DataTable")
    ds.Tables.Add(myDataTable)
    Dim myDataColumn As DataColumn
    Dim myDataRow As DataRow

    myDataColumn = New DataColumn()
    myDataColumn.DataType = GetType("System.String")
    myDataColumn.ColumnName = "Count"
    myDataColumn.Caption = "Count"
    ' Add the Columns
    myDataTable.Columns.Add(myDataColumn)

    ' Create second column
    myDataColumn = New DataColumn()
    myDataColumn.DataType = System.Type.GetType("System.String")
    myDataColumn.ColumnName = "Item"
    myDataColumn.Caption = "Item"

    ' Add the column to the table.
    myDataTable.Columns.Add(myDataColumn)

    ' Create three new DataRow objects and add them to the DataTable
    Dim i As Integer
    For i = 0 To 2 Step 1
        myDataRow = myDataTable.NewRow()
        myDataRow("Count") = i.ToString()
        myDataRow("Item") = i.ToString()
        myDataTable.Rows.Add(myDataRow)
    Next
    Return ds
End Function

```

C#

```

private System.Data.DataSet GetDataSet()
{
    DataSet ds = new DataSet("MyDataSet");
    DataTable myDataTable = new DataTable("My DataTable");
    ds.Tables.Add(myDataTable);
    DataColumn myDataColumn;
    DataRow myDataRow;

    myDataColumn = new DataColumn();
    myDataColumn.DataType = System.Type.GetType("System.String");
    myDataColumn.ColumnName = "Count";
    myDataColumn.Caption = "Count";
    // Add the Columns
    myDataTable.Columns.Add(myDataColumn);

    // Create second column
    myDataColumn = new DataColumn();
    myDataColumn.DataType = System.Type.GetType("System.String");
    myDataColumn.ColumnName = "Item";

```

```
        myDataColumn.Caption = "Item";

        // Add the column to the table.
        myDataTable.Columns.Add(myDataColumn);

        // Create three new DataRow objects and add them to the DataTable
        for (int i = 0; i <= 2; i++)
        {
            myDataRow = myDataTable.NewRow();
            myDataRow["Count"] = i.ToString();
            myDataRow["Item"] = i.ToString();
            myDataTable.Rows.Add(myDataRow);
        }
        return ds;
    }
}
```

Listing 2-6 Dataset creation routine

The dataset is an object that holds data and schema information from an underlying data source. The dataset is disconnected in nature; that is, once it retrieves the data, it is no longer attached to the data source. To some, the ADO.NET dataset presented in Listing 2-6 is the best thing since sliced bread. To others, it may be awkward and unforgiving. Thankfully, VSTO caters to both camps. VSTO controls can easily consume data from a dataset or the controls can bypass the dataset and bind directly to a database.

Listing 2-7 provides some bare-bones code to load data from a dataset.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Handles Me.Startup
    Dim count As Integer
        count = 0

        ds = GetDataSet()

        For Each dt In ds.Tables(0).Rows
            count = count + 1
            rng = Me.Application.Range("A" & count)
            rng.Value = dt.ItemArray
        Next
    End Sub
```

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
    DataSet ds = GetDataSet();
    int count = 1;
    foreach (DataRow dt in ds.Tables[0].Rows)
    {
```

```

        Excel.Range rng = this.Application.get_Range("A" + count++,
missing);
        rng.Value2 = dt.ItemArray;
    }
}

```

Listing 2-7 Dataset assignment to Range object

A few comments are in order. The `itemarray` method actually retrieves an entire row and assigns it directly to the range. The assignment is handled internal, since the `Range` object can load an array.

It is rather unfortunate that VSTO controls do not contain internal support for binding directly to the ADO.NET dataset. Having to write bind code to accomplish the data load is not only cheesy, but it is less efficient due to the expense of the executing loop and the assignment statements inside the loop. In some circumstances, .NET may perform certain optimizations to make the loop more efficient such as loop unrolling. However, an internal implementation would remove the burden of implementation from the developer.

If you are not particularly pleased with the code approach, you may opt to use the dataset's `writeXML` method to convert the data to an XML-formatted file on disk. Then, you may load the XML file into the spreadsheet using any one of the approaches presented thus far. You may find this approach tempting because VSTO offers deep support for XML data sources. Listing 2-8 has the code to accomplish this.

Visual Basic

```

    If (ds IsNot Nothing AndAlso ds.Tables.Count > 0 AndAlso
ds.Tables(0).Rows.Count > 0) Then
        Dim tempFilePath As String = "C:\download\temp.xml"
        ds.WriteXml(tempFilePath)
        Application.Workbooks.Open(tempFilePath, Delimiter:=",",
Editable:=True, AddToMru:=True)
    End If

```

C#

```

if (ds != null && ds.Tables.Count > 0 && ds.Tables[0].Rows.Count > 0)
{
    string tempFilePath = @"C:\download\temp.xml";
    ds.WriteXml(tempFilePath);
    Application.Workbooks.Open(tempFilePath, System.Type.Missing,
false, System.Type.Missing, System.Type.Missing, System.Type.Missing,
System.Type.Missing, System.Type.Missing, ",", true, System.Type.Missing,
System.Type.Missing, true, System.Type.Missing, System.Type.Missing);
}

```

Listing 2-8 Dataset writeXML method

Temporary files such as those written by datasets can accumulate on disk to cause resource issues. A well-behaved, scalable cleanup routine is certainly not a trivial task. To prevent these resource hogs from running rampant, some sort of cleanup routine must be implemented by the developer. The lesson here is that a tempting choice may not be worth it in terms of total cost. When choosing an approach, be mindful of trading elegance for performance.

As noted previously, without internal support for the ADO dataset, the code to perform the binding is much slower. Still, the actual performance cost is not that noticeable for small to medium-sized datasets in real-world software. Large dataset loads should simply not be attempted.

Large datasets on the order of several thousand rows of data present serious memory challenges to the .NET Framework that, to date, have not been satisfactorily addressed either in 1.x or 2.0, not to mention the inherent limitations of the spreadsheet beyond 65,536 rows.

But perhaps the most serious drawback is not a technical limitation at all. Presenting large amounts of data to the end user has a tendency to overwhelm and intimidate. Users who are not comfortable with software often lay the blame in all the wrong places, and it really isn't their fault. A well-designed application presents data in controlled portions to prevent data overload.

Finally, if you are less than impressed with the ADO.NET dataset or have your own reasons for avoiding its use in your application, you can certainly benefit both in terms of performance and code readability by connecting your application directly to the data source. The `QueryTable` object presented previously can make this a reality. You may also consider loading data directly from the database with the help of the `Workbooks' opendatabase` method of the `Application` object. We'll also explore some new VSTO controls in Chapter 3 that present additional alternatives to retrieving data from a data source.

Application Object Manipulation

The `Application` object is a top-level container for VSTO. The `Application` object provides a global hook for all the objects and containers in the Excel environment. In fact, the `Application` object is the Microsoft Excel application instance.

The code snippet in Listing 2-9 shows how to manipulate the `Application` object.

Visual Basic

```
Application.DisplayAlerts = false
Me.Application.ActiveWorkbook.Sheets(2).Delete()
Application.DisplayAlerts = true
Application.Quit()
```

C#

```
Application.DisplayAlerts = false;
((Excel.Worksheet)this.Application.ActiveWorkbook.Sheets[2]).Delete();
Application.DisplayAlerts = true;
Application.Quit();
```

Listing 2-9 Application object manipulation

By default, when macro code is being executed, Excel may automatically generate dialog boxes and prompts. In instances where there is no end user, such as in server-side automation or remoting scenarios, this may cause the application to hang or fail. The code shows how to gracefully override the prompting mechanism. Finally, the application terminates. Termination in this case means that the application is gracefully shut down and removed from memory. Even that type of behavior can be difficult to achieve outside of Visual Studio Tools for Office. Excel application instances can demonstrate a stubborn resistance to being terminated

Workbook Manipulation

Microsoft Excel 2000 introduced the concept of a workbook object. A workbook object is implemented as a collection. Each workbook contains a few worksheets that are available for use. In essence, the workbook functions as a housing container for internal objects. Because it is a top-level document, a large majority of the methods are geared toward housekeeping tasks.

Listing 2-10 shows some code to open a text file on disk using the workbook object's `Open` method call.

Visual Basic

```
Dim nWorkbook As Excel.Workbook
    nWorkbook = Me.Application.Workbooks.Add()
    Me.Application.Workbooks.Open("C:\Download\CSVData.txt", Delimiter:=",",
Editable:=True, AddToMru:=True)
    Me.Application.Workbooks(1).Activate()

    If (Not nWorkbook.Saved) Then
        nWorkbook.SaveAs("CSVData.txt", Excel.XlFileFormat.xlXMLSpreadsheet)
    Else
        nWorkbook.Close()
    End If
```

C#

```
Excel.Workbook nWorkbook = this.Application.Workbooks.Add(missing);
    this.Application.Workbooks.Open(@"C:\Download\CSVData.txt",
System.Type.Missing, false, System.Type.Missing, System.Type.Missing,
System.Type.Missing, System.Type.Missing, System.Type.Missing, ",", true,
System.Type.Missing, System.Type.Missing, true, System.Type.Missing,
System.Type.Missing);

((Microsoft.Office.Interop.Excel._Workbook)this.Application.Workbooks[1]).Activate(
);

    if (!nWorkbook.Saved)
        nWorkbook.SaveAs(@"CSVData.txt",
Excel.XlFileFormat.xlXMLSpreadsheet, missing, missing, missing, missing,
Excel.XlSaveAsAccessMode.xlNoChange, missing, missing, missing,missing,missing);
    else
        nWorkbook.Close(false, missing, missing);
```

Listing 2-10 Workbook manipulation example

The code presented in Listing 2-10 creates a new workbook. A reference to the new workbook is stored in the variable `nWorkbook`. Using the application object's workbook object, the workbook is opened. If the workbook has been modified, it is saved. Otherwise, the workbook is simply closed without being saved. The Boolean `false` parameter takes care of this internal plumbing.

There are also several ways to access the workbook from code. For instance, `this.SaveAs`, `Globals.ThisWorkbook`, `Application.Workbooks` are all valid methods for accessing the workbook and its gamut of methods. The flexibility is important when dealing with code outside the hosting container or in a different module such as an Excel add-in. Even with this flexibility, it is important to adopt a consistent approach to accessing workbooks. You may expect this consistency to pay back dividends in the maintenance phase of software development.

Worksheet Manipulation

The `Workbook` object in VSTO contains three worksheets by default. Sheet 1, 2, and 3 contain support for 65,536 rows and 256 columns per sheet. Excel also supports the ability to add or remove worksheets from the collection of sheet objects. You should note that this limitation on rows and columns is imposed by VSTO and not the Excel API. In fact, other methods of accessing the API (such as the Office Web Components) support a larger number of columns than VSTO. Listing 2-11 shows some code that removes sheet 2.

Visual Basic

```
me.Application.ActiveWorkbook.Sheets(2).Delete()
```

C#

```
((Excel.Worksheet)this.Application.ActiveWorkbook.Sheets[2]).Delete();
```

Listing 2-11 Worksheet manipulation

Experienced programmers should realize that there are a number of things that could go wrong with this line of code. For instance, the code makes no guarantee that the specified sheet actually exists in the workbook before attempting to delete it. Code attempting to delete a worksheet should first examine the `count` property of the `Sheets` object before blindly deleting. An attempt to delete an invalid worksheet will result in a runtime exception. The code can be modified to simply examine the `count` property of the `Sheets` object to make sure it is greater than 2 before calling the `Delete` method.

Another serious issue has to do with the availability of the sheet tagged for deletion. If the sheet is in use or locked by another process or thread, the deletion will fail with an exception. Exception handling is, therefore, a very strong recommendation when attempting to remove sheets from the collection.

Finally, the delete triggers a security dialog prompting the user for confirmation before the deletion occurs. This security message serves as a safeguard against malicious code or ill-conceived software. If the code can successfully navigate these pitfalls, the sheet is removed from the collection and the names of the remaining sheets are adjusted as necessary to maintain a consistent count in the user interface. For instance, deleting sheet 2 in the default collection causes sheet 3 to be renamed sheet 2.

By the same token, it is easy to add a new worksheet to the collection. Examine the code in Listing 2-12.

Visual Basic

```
Dim newWorksheet As Excel.Worksheet  
newWorksheet = DirectCast(Globals.ThisWorkbook.Worksheets.Add(), Excel.Worksheet)
```

C#

```
Excel.Worksheet newWorksheet;  
newWorksheet = (Excel.Worksheet)Globals.ThisWorkbook.Worksheets.Add(  
    missing, missing, missing, missing);
```

Listing 2-12 Adding a new worksheet

The code is simple but, as is often the case, there are evil devices just out of reach that can cause nasty surprises. Let us take a closer look. If you open the Visual Studio Tools for Office designer and click on the Excel spreadsheet, the properties window reveals that the sheet is of type `Microsoft.Office.Tools.Excel.Worksheet`. However, if you inspect the `Add` method by using Visual Studio IntelliSense

or the help documentation, you will find that the `Add` method creates a worksheet of type `Microsoft.Office.Interop.Excel.Worksheet`. In essence, you are adding a worksheet of a different type.

In case you missed the sleight of hand, the API is actually creating a worksheet of a lesser type; that is, a type that is not exactly equal to what existed before. This newly created worksheet type is an imposter. It has inferior functionality to a `Microsoft.Office.Tools.Excel.Worksheet` object because it is not able to handle events, respond to code, or bind to data at runtime or design time. Consequently, if you ever have the need to add a worksheet at runtime, you should be aware of the fact that you are creating an object with a significant handicap.

How can calling code overcome this shortfall? One bright idea is to try to copy or clone an existing worksheet of the correct type `Microsoft.Office.Tools.Excel.Worksheet`. Then, this new copy can be used. It seems that this approach would get around the limitation described above, because the code would copy an existing worksheet of the correct type. However, the `Copy` method is not a true deep copy. It does not actually clone the worksheet. It simply calls the `Add` method internally to create a new worksheet of type `Microsoft.Office.Interop.Excel.Worksheet`. Once the new worksheet is created, a member-wise assignment to populate the properties of the new worksheet from the source worksheet is performed. This puts us right back where we started. It simply isn't possible to add or create a worksheet of type `Microsoft.Office.Tools.Excel.Worksheet` by calling code.

In any event, here is the code to copy a spreadsheet. You should note that the copy process does not solve the problem presented previously. Later, we will examine one approach to resolving this issue. For now, we focus on some more functionality that allows us to manipulate Excel worksheets, as shown in Listing 2-13.

Visual Basic

```
Me.Copy(After:=Globals.ThisWorkbook.Sheets(1))
```

C#

```
this.Copy(missing, Globals.ThisWorkbook.Sheets[1]);
```

Listing 2-13 Worksheet copy functionality

As pointed out previously, attempting to copy an invalid worksheet results in an exception. The `worksheets` object isn't particularly interesting and does not offer a rich feature set because it acts mostly as a container for `Range` objects and a design surface for hosting other controls.

One particularly intriguing property deserves some attention. Consider the snippet of code in Listing 2-14.

Visual Basic

```
DirectCast(Globals.ThisWorkbook.Sheets(1), Excel.Worksheet) _  
    .Visible = Excel.XlSheetVisibility.xlSheetHidden
```

C#

```
((Excel.Worksheet) Globals.ThisWorkbook.Sheets[1]).Visible =  
Excel.XlSheetVisibility.xlSheetHidden;
```

Listing 2-14 Worksheet visibility example

The `XlSheetVisibility` enumeration contains three choices. The code shown in Listing 2-15 hides the selected worksheet, but the user still has the ability to make the worksheet visible from the Excel menu. Another option in the `Excel.XlSheetVisibility` enumeration, `xlSheetVeryHidden`, prevents the user from making the hidden worksheet visible. The final option in the enumeration makes the worksheet visible.

And, quite by accident, we have also stumbled upon a solution to the problem presented in the previous section. You may recall that a dynamically created worksheet is technically challenged because it is created from a type (`Microsoft.Office.Interop.Excel.Worksheet`) that is unable to handle events, respond to code, or bind to data at runtime or design time. The fix is simply to create as many worksheets as required by the application at design time and hide them as appropriate in the startup code using the code in Listing 2-14. Because the worksheets are being created in the Visual Studio designer, they will be created with the correct type: `Microsoft.Office.Tools.Excel.Worksheet`. When needed, these sheets can be brought into view by toggling the visibility through the enumeration shown in Listing 2-14. Since they are bona fide hosting containers, the sheets will respond to code changes, events, and data binding.

Excel Range Manipulation

In some cases, end-user interaction may require that data displayed in the Excel spreadsheet be saved to a data source. In order to store the data, the values in the Excel spreadsheet need to be extracted, parsed, and prepared for serialization. *Serialization* is the process of storing objects in such a way that they can be reconstructed later when needed. For instance, an application object such as the Excel Range object may be serialized into a file on disk. The Framework disassembles the object into sequential bytes and then writes these bytes to a file on disk. Later, the file can be read to obtain the series of bytes in order to reconstruct the object for use. This is normally handled internally by the Framework and requires no input from the developer. The Excel object facilitates this extraction primarily through the Range object. The next section examines data extraction at the range level.

The Excel range is an integral part of the Excel machinery. A range may represent a logical or physical relationship between cells. The group of cells can be manipulated and formatted through the reference contained inside the Range variable. Listing 2-15 shows some code to demonstrate range manipulation.

Visual Basic

```
Dim rng As Excel.Range = Globals.Sheet1.Range("a1", "g12")
    rng.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects2, True,
False, True, False, True, True)
```

C#

```
Excel.Range rng = Globals.Sheet1.Range["a1", "g12"] as Excel.Range;
    rng.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects2,
true, false, true, false, true, true);
```

Listing 2-15 Autoformatting a range

A reference to a Range is obtained from the spreadsheet representing a contiguous block of cells stretching from A1 to G12, as shown in Figure 2-2.

Using the reference to this contiguous block, some autoformatting is applied to the entire range using the AutoFormat method. Notice that the `Excel.XlRangeAutoFormat` enumeration applies a 3-D formatting effect. The enumeration contains about 30 members. The large number of options represents common functionality in spreadsheet applications.

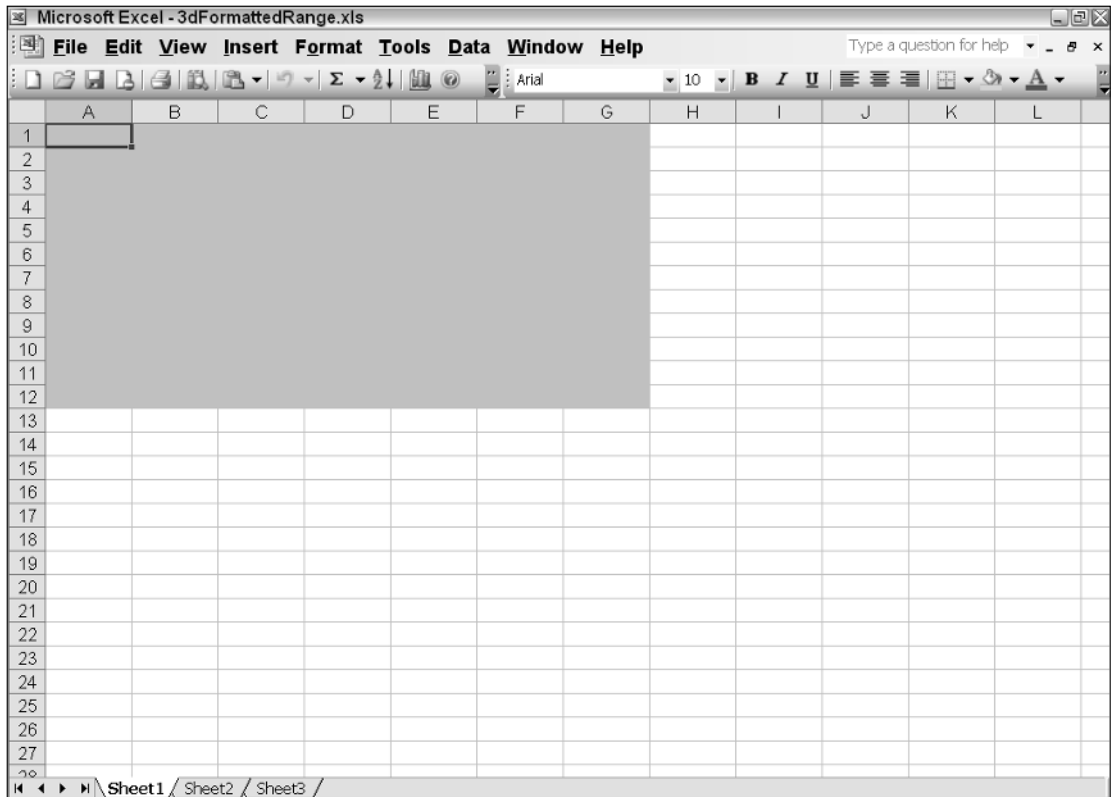


Figure 2-2

Working with Named Ranges

Microsoft Excel allows ranges to be tagged with name identifiers. The functionality is strictly a convenience and does not affect runtime performance or resource utilization. Consider the example in Listing 2-16.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim namedRange1, namedRange2 As Microsoft.Office.Tools.Excel.NamedRange
    namedRange1 = Controls.AddNamedRange(Me.Range("A1", "A10"), "namedRange1")

    namedRange2 = Controls.AddNamedRange(Me.Range("A8", "B12"), "namedRange2")

    namedRange1.Merge()
    namedRange2.Merge()

    namedRange1.BorderAround(, Excel.XlBorderWeight.xlThin,
    Excel.XlColorIndex.xlColorIndexAutomatic, )
    namedRange1.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects1,
    True, False, True, False, True, True)
End Sub
```

Chapter 2

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
    Microsoft.Office.Tools.Excel.NamedRange namedRange1 =
    Controls.AddNamedRange(this.Range["A1", "A10"], "namedRange1");

    Microsoft.Office.Tools.Excel.NamedRange namedRange2 =
    Controls.AddNamedRange(this.Range["A1", missing], "namedRange2");

    namedRange1.Merge(false);
    namedRange2.Merge(false);

    namedRange1.BorderAround(missing, Excel.XlBorderWeight.xlThin,
    Excel.XlColorIndex.xlColorIndexAutomatic, missing);

    namedRange1.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects1,
    true, false, true, false, true, true);
}
```

Listing 2-16 Named range usage

Listing 2-16 produces the image shown in Figure 2-3.

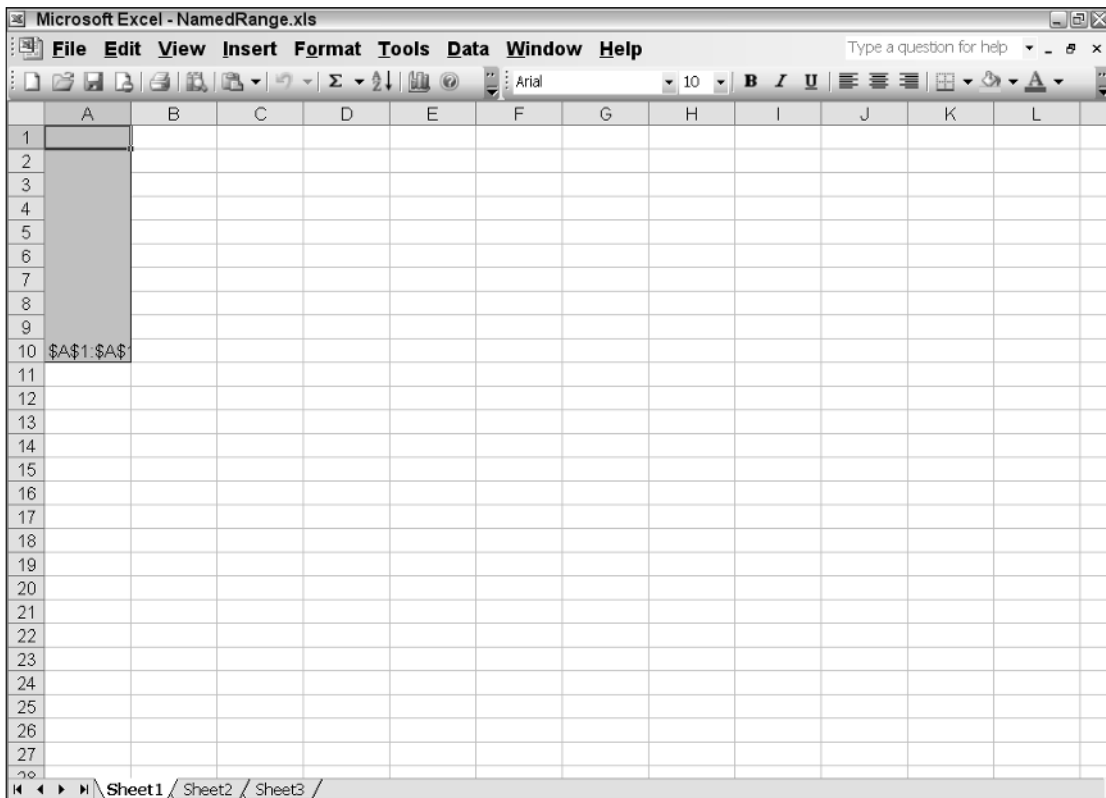


Figure 2-3

Listing 2-17 shows that two `namedrange` objects are created. A merge is performed using the `name-range` reference for each region. The `merge` parameter defaults to `false` in Visual Basic, but in C# it must be explicitly passed in. The parameter instructs VSTO how to treat the merge. In Listing 2-17, each row of the specified range is merged as separate cells. Once the merge is completed, some formatting is applied to the border and to the range for the `namedRange1` area. You can see that the merge is successful since the range A1 now extends to row A10 in Figure 2-3.

Working with Cells

The `Range` provides a great deal of functionality and is a good solution for managing data manipulation. However, you can exercise more control on the spreadsheet by working with the `cell` object. The cell represents an individual cell on the spreadsheet. The code snippets in Listing 2-17 show how to assign a value to an individual cell.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim row As Object = 1, column As Object = 2
    Dim rng As Excel.Range = Globals.Sheet1.Cells(row, column)
    rng.Value = "123456"
    rng.NumberFormat = "$#,###.0"
End Sub
```

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
    object row = 1, column = 2;
    Excel.Range rng = Globals.Sheet1.Cells[row, column] as Excel.Range;
    rng.Value2 = "123456";
    rng.NumberFormat = "$#,###.0";
}
```

Listing 2-17 Cell manipulation

Consider the code presented in Listing 2-17. A reference to a specific cell [1, 2] is obtained and stored inside a `Range` variable. The value 123456 is added to the cell and formatted. The string 123456 should then be displayed as \$123,456.0. `NumberFormatting` will be probed in greater detail in the next section.

The syntax of this effort is probably confusing to say the least because a `Cell` reference actually returns a variable to a `Range` (notice the cast to `Excel.Range`). In Listing 2-17, the range represents the particular cell. However, you should note that the returned value or reference can sometimes refer to a group of cells as opposed to a single cell. The decision to expose cells as ranges necessarily sacrifices clarity for flexibility. References to cells can be obtained through the `Application`, `Worksheet`, and `Range` objects, as shown in Listing 2-18.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim rng As Excel.Range = Globals.Sheet1.Range("B3", "B3")
    rng.Value = "123456"
    rng.NumberFormat = "$#,###.0"
    rng = rng.Cells(3, 3) as Excel.Range
    rng.Value = "new"
End Sub
```

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
    Excel.Range rng = Globals.Sheet1.Range["B3", "B3"] as Excel.Range;
    rng.Value2 = "123456";
    rng.NumberFormat = "$#,###.0";
    rng = rng.Cells[3, 3] as Excel.Range;
    rng.Value2 = "new";
}
```

Listing 2-18 Cell manipulation using relative positioning

In Listing 2-18, the C# code `rng = rng.Cells(3, 3) as Excel.Range` does not refer to the third row of the third column. Instead it refers to the third row of the third column starting from cell B3, B3—the new reference point. The same holds true for the Visual Basic. These subtle differences can always cause major headaches.

The parameters passed to the cell object in Listing 2-18 `Globals.Sheet1.Cells[row, column] as Excel.Range` is always a relative coordinate and never an absolute value. The starting point of this relative address is the upper-left corner of the Range object in question. When the returned value is an absolute value (not a relative coordinate), the reference point is always the upper-left corner of the Excel spreadsheet. It is important to understand this fundamental difference.

Working with Unions

In some situations, it may be convenient to treat two groups of cells with the same code. Instead of manipulating these areas individually, Excel offers access to these two different regions through the Union object. Listing 2-19 is a code example to target a union region.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim rngA, rngB, unionRange As Excel.Range
    rngA = Range("a2", "B3")
    rngA.Value = "Mortgage"
    rngB = Range("a5", "B6")
    rngB.Value = "Interest"
    unionRange = Application.Union(rngA, rngB)
    unionRange.Font.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Red)
End Sub
```

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
    Excel.Range rngA = Globals.Sheet1.Range["a2", "B3"] as Excel.Range;
    rngA.Value2 = "Mortgage";
    Excel.Range rngB = Globals.Sheet1.Range["a5", "B6"] as Excel.Range;
    rngB.Value2 = "Interest";

    Excel.Range unionRange = Application.Union(rngA, rngB, missing,
missing, missing, missing, missing, missing, missing, missing, missing,
missing, missing, missing, missing, missing, missing, missing, missing,
missing, missing, missing, missing, missing, missing, missing, missing) as
Excel.Range;
```

```

        unionRange.Font.Color =
        System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Red);
    }

```

Listing 2-19 Excel unions

At first glance, the union functionality does not seem awe-inspiring. However, upon closer inspection, it is really convenient to address multiple ranges through a single point of entry as the code in Listing 2-20 demonstrates. The ease of use adds up to reduced code and also acts to restrict the surface area for bugs because there is less chance of using an incorrect range reference.

There are some issues regarding unions that should concern you. Unions are exposed as `Range` objects. However, unions do not support all of the methods that are listed in intellisense for `Range` objects. For instance, the `AddComment` method used to add a comment to a cell will actually compile without issue, but throw an exception at runtime. Instead, you should use the `namedRange's AddComment` method to add a comment to a cell. There are other idiosyncrasies, too numerous to mention here, that need to be worked out with a dose of aspirin and patience. A large majority of these idiosyncrasies are not explained in the help documentation.

Working with Intersections

VSTO exposes the `intersection` range object, which may be used to find the intersection of cells. There are several reasons why intersections can be helpful. For instance, you may want to apply some formatting to the common areas between two ranges. Listing 2-20 provides an example.

Visual Basic

```

Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim rngA, rngB, intersection As Excel.Range
    rngA = Globals.Sheet1.Range("a2", "B3")
    rngB = Globals.Sheet1.Range("a5", "B6")

    intersection = Application.Intersect(rngA, rngB)
    If intersection IsNot Nothing Then
        'add code here
    End If
End Sub

```

C#

```

private void Sheet1_Startup(object sender, System.EventArgs e)
{
    Excel.Range rngA = Globals.Sheet1.Range["a2", "B3"] as Excel.Range;
    Excel.Range rngB = Globals.Sheet1.Range["a5", "B6"] as Excel.Range;

    Excel.Range intersection = Application.Intersect(rngA, rngB, missing,
    missing, missing, missing, missing, missing, missing, missing, missing, missing,
    missing, missing, missing, missing, missing, missing, missing, missing, missing) as
    Excel.Range;
}

```

```
        if (intersection != null)
        {
            //add code here
        }
    }
```

Listing 2-20 Excel intersections

Listing 2-20 obtains two references to two ranges. The references are stored in `rngA` and `rngB`. Then, the application's `intersection` method is used to find the ranges that intersect. The `if` block in the code in Listing 2-20 tests for common regions in the ranges. If there are no intersecting cells, a null value is returned.

Working with Offsets

You may recall from the “Working with Cells” section that the cell address may be a relative address instead of an absolute one. Offsets embody this concept but provide a cleaner way to implement relative addressing. The offset may be applied to the row or the column object and indicates the cell distance that must be traveled from the reference point.

Consider cell A1, which resides in the upper-left corner of the Excel spreadsheet. An offset of [3, 5] indicates that the new reference point is exactly three rows down and five columns to the right of the reference cell A1. The new cell reference is now E3, as the code in Listing 2-21 demonstrates.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Dim row As Object = 3, column As Object = 5
    Dim off As Excel.Range
    off = Globals.Sheet1.Range("a1", "a1")
    off.Value = "Initial Position"
    off = off.Offset(row, column)
    off.Value = "Final Position"
End Sub
```

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
    object row = 3, column = 5;
    Excel.Range off = Globals.Sheet1.Range["a1", "a1"] as Excel.Range;
    Off.Value2 = "Initial Position";
    off = off.get_Offset(row, column) as Excel.Range;
    off.Value2 = "Final Position";
}
```

Listing 2-21 Excel offsets

The code example in Listing 2-21 shows that a reference to a range "a1", "a1" is first obtained, followed by a reference to an offset. Then, a value is written to the cell at that position. Cell a1, a1 now contains the value "Initial Position". From position 1,1 — a1,a1 — count three rows down and then five rows to the right. Do not include cell A1, since it is the reference point. If you care to run the code, the string "Final Position" will appear in the column F4. Figure 2-4 shows the result of the code.

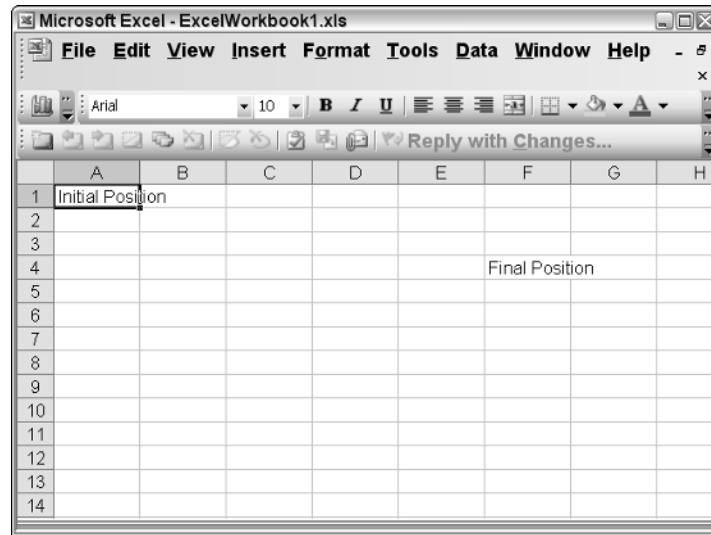


Figure 2-4

The preceding examples have shown the `Range` object to be a key player in building Excel applications. The `Range` object allows easy access to areas on the Excel spreadsheet and a convenient way to apply formatting and functionality to these areas. These areas may represent an individual cell or a group of cells, or these areas may form part of another worksheet. In the case where these groups of cells reside in the same worksheet, the cells may be referenced as contiguous blocks, unions, or intersections. For other cases, the `Range` object still affords the flexibility to apply formatting and functionality to groups of cells that reside on another sheet.

The `Range` object even allows you to target noncontiguous blocks of cells. This feature may prove handy when coding functionality based on complex range interaction.

So far, the few examples presented run the gamut of functionality that is used in a typical real-world application. Remember, most of the feature set of a real-world Excel application is already internally implemented by the spreadsheet and needs no further customization from a team of programmers. Developers are strongly advised to resist the urge to modify the internally implemented functionality, since end users have grown accustomed to these features. Instead, if additional features must be implemented, you may add customized buttons to the toolbar to incorporate the new feature set.

Although the `Value2` property of the `Range` object is exposed in Visual Basic, you should not use it. The implementation is provided for C# applications and does carry some additional overhead. Use the `Value` property of the `Range` object for Visual Basic. The `value` property is read-only for C# and cannot be the target of an assignment.

Data Formatting and Presentation

Excel is bred for formatting and presenting data to the user. Formatting data can make data analysis for accounting scenarios easier and less prone to interpretation errors. For instance, numbers that line up neatly against the decimal point — accounting format — tend to be easier to understand and analyze when compared to numbers that aren't lined up in that manner. That sort of format can be easily applied by the end user.

Excel is capable of applying formatting effects at design time as well as runtime. The design-time formatting can be accessed by selecting Format ⇨ Autoformat. Figure 2-5 shows the Excel spreadsheet with the AutoFormat dialog box that was used to create the customized background.

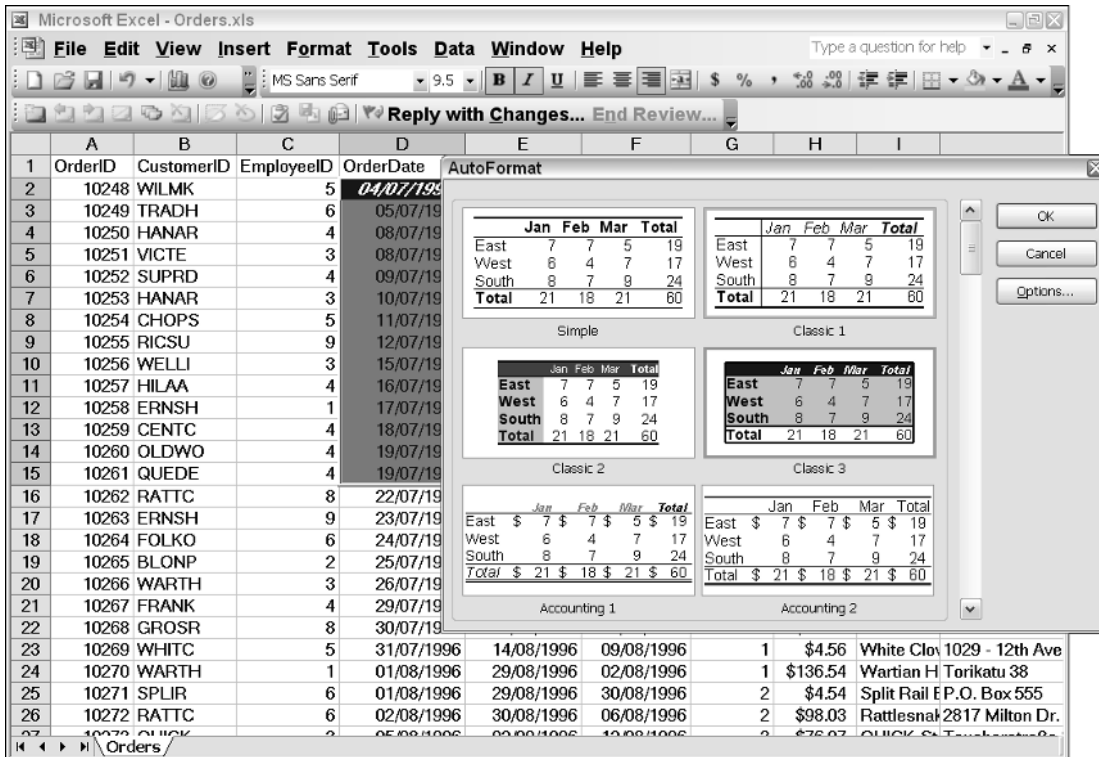


Figure 2-5

That type of design-time functionality in Visual Studio Tools for Office was simply not possible before. For instance, you may choose to add a corporate watermarked image to spiff up your spreadsheet for a professional appearance or you may choose to use a default background. At least, now you have choices. At design time, backgrounds are available by selecting Format ⇨ Sheets ⇨ Backgrounds. At runtime, the option is available by calling the `SetBackgroundPicture` method of the `Worksheet` object. An example is provided on the next page.

Excel also allows conditional formatting from the designer. For instance, a particular style may be applied to a cell based on the content of the data in that cell. The option is available by selecting Conditional Formatting from the Format menu in the designer.

It bears repeating that all of these design-time options may be accomplished at runtime. For instance, conditional formatting may be applied through the `Range` object. Other functionality, such as attaching a bitmap to the Excel spreadsheet, is simply a matter of calling the appropriate function on the worksheets object as the next snippet of code demonstrates.

```
Application.Rows.Worksheet.SetBackgroundPicture(@"C:\Documents and
Settings\All Users\Documents\My Pictures\Sample Pictures\blue hills.jpg");
```

The `SetBackgroundPicture` method supports most image formats, and the overhead of the image load is negligible if the file resides locally. If the file resides remotely, on an image server for instance, there is added overhead in loading the image. However, the extra expense is experienced only at startup. The Excel spreadsheet is smart enough to cache the image locally. As always, the executing process needs read access to the image file on disk. Chapter 3 discusses permissions access and configuration for Excel applications.

Excel also supports runtime formatting. The benefits are the same as with design-time formatting, except that some runtime overhead is incurred. The functionality is supported through the `NumberFormat` property. Consider the example in Listing 2-22.

Visual Basic

```
Dim valueRange as Microsoft.Office.Tools.Excel.NamedRange =
this.Controls.AddNamedRange(Me.Range("B2", "F6"), "ValueRange");
valueRange.NumberFormat = "#,###.00"
```

C#

```
Microsoft.Office.Tools.Excel.NamedRange valueRange =
this.Controls.AddNamedRange(this.Range["B2", "F6"], "ValueRange");
valueRange.NumberFormat = "#,###.00";
```

Listing 2-22 NumberFormat property

If the user were to enter 1234 in the cell, the Excel formatter would convert the output to 1,234.00. This output mask is sometimes called a picture format for obvious reasons.

VSTO does not support keywords with `NumberFormats`. However, there is one exception. The “General” format may be used to reset the `NumberFormat` property. As an example, consider the code in Listing 2-23.

Visual Basic

```
Dim valueRange As Microsoft.Office.Tools.Excel.NamedRange
valueRange = Me.Controls.AddNamedRange(Me.Range("B2", "F6"), "ValueRange")
valueRange.NumberFormat = "#,###.00"
'add some processing code here

'reset the range B2, F6 to its default format
```

```
valueRange.NumberFormat = "General"
```

```
'This line throws an exception because the currency keyword is not supported  
'valueRange.NumberFormat = "Currency"
```

C#

```
Microsoft.Office.Tools.Excel.NamedRange valueRange =  
this.Controls.AddNamedRange(this.Range["B2", "F6"], "ValueRange");  
valueRange.NumberFormat = "#,###.00";  
//add some processing code here  
  
//reset the range B2, F6 to its default format  
valueRange.NumberFormat = "General";  
//This line throws an exception because the currency keyword is not supported  
//valueRange.NumberFormat = "Currency";
```

Listing 2-23

It bears repeating that, except for the `General` keyword, `NumberFormats` do not support any keywords. In some cases, exceptions will be thrown to indicate this. An example of code that produces an exception is listed in the previous code snippet. However, in other cases, garbage results will be produced in the cell. As an example, consider the snippet of code that assigns a keyword "Date" to a range as in:

```
valueRange.NumberFormat = "Date"
```

This may seem like a very useful assignment and may actually succeed in VBA. However, the results that are produced in VSTO are not correct. In essence, the garbage is produced because Excel does not know how to handle the letter "a" and the letter "e" in the `NumberFormat` string. Excel does know how to handle the letter "D" and the letter "t," so the call appears to half-succeed with a mixture of valid data and junk. From the results, it is not immediately clear whether the input caused the incorrect results or the call itself produced the rubbish. Instead of spending fruitless time on figuring out the cause, just avoid keyword assignments in the `NumberFormat` property in VSTO.

VSTO does not support letters or special characters with `NumberFormats` either. If you have been accustomed to formatting data using a combination of character codes and letters, you may be disappointed with the VSTO offering. For instance, a format mask such as this:

```
"#,###;-#.###.00;[#.##];";
```

in Visual Basic for Applications code is not directly portable to VSTO. Although the code will compile, the results will be incorrect. There is also no support for padding characters and cell coloring through number formats, as is available in VBA and Excel Interop. A workaround for this type of formatting is to build it in code using the `Range` object formatting. Needless to say, the lack of advanced support for number formatting is a serious hindrance to development efforts especially when the project at hand involves porting code with a heavy implementation of `NumberFormats`.

Considerations for Excel Automation

Idiosyncrasies abound with Office automation. Knowing where these monsters hide can save time, effort, and Tylenol. Help documentation is available and considerably better than previous versions. However, the help documentation often does not contain example code. This can be problematic,

especially when time is of the essence. Trial and error will get you through. Alternatively, you may try searching the Internet or accessing the VSTO forums at <http://forums.microsoft.com/msdn/showforum.aspx?forumid=16&siteid=1>.

Another consideration is avoiding the use of macros entirely. While VSTO supports macro processing, it should not be used because the engine can perform the exact functionality as macros without the security risks. For a review on macro issues, see the “Disadvantages” section in Chapter 1.

Realize that Excel controls created at runtime cannot respond to events and data binding. The only current workaround is to add the items at design time. In the case of workbooks, there is no workaround, since the designer does not allow workbook addition. However, worksheets can benefit from this approach, and an example has been provided in the chapter.

When working with the spreadsheet using relative cell positioning, ensure that the code references cells that are actually on the worksheet. Attempting to reference cells that are not valid will result in a runtime exception being thrown.

Concerning data file loads, using the right method to load data into the spreadsheet goes a long way toward performing the job efficiently. In order to determine the right method to use, consider the type of data being loaded and whether or not any processing is required during the file load. As the code has shown, some methods offer more functionality for processing than others.

You should also be aware that VSTO contains internal support for data loads. Several examples have been presented in the chapter and the wide surface area for loading data is impressive. It’s also important to note that you should first consider using internally implemented functionality before considering .NET approaches. The internally supported functionality is optimized for VSTO-based usage, whereas the .NET functionality is optimized for usage in general. One example of an internally supported feature is query tables. As pointed out in the examples earlier in this chapter, the query table implements a web query method that provides the same functionality as a web service, albeit with a much more feature rich offering.

Finally, Microsoft Excel is an end-user system. Not all applications should be built around Excel. However, Excel lends itself particularly well to applications that require some sort of numerical analysis and data presentation. You may be able to take advantage of the internally supported functions to help with your analysis. Charts and reports are also an added bonus and are easy to use and incorporate into a VSTO-based application.

Excel Case Study — Time Sheet Software

So far, you have learned bits and pieces through snippets. This section shows you how to put an application together based on the venerable time sheet application. The code that follows is a bare-bones application that provides a worksheet to track hours worked. At a high level, consultants open the spreadsheet and enter billable hours. Once data entry is done, the application sends the time sheet via email to a predetermined email address.

Chapter 2

This type of application is very common among small companies that have neither the resources nor the budget to purchase expensive off-the-shelf time management software. These companies usually keep track of employee hours in a spreadsheet that is updated at the end of the pay period. The code presented in Listing 2-24 will simulate that functionality.

Visual Basic

```
Private Sub FormatSheet()  
    'take care of some aesthetic issues  
    Application.DisplayFormulaBar = False  
    Application.DisplayFunctionToolTips = False  
    Application.DisplayScrollBars = False  
    Application.DisplayStatusBar = False  
  
    'make some customizations  
    Application.Rows.Worksheet.SetBackgroundPicture("C:\Documents and  
Settings\All Users\Documents\My Pictures\Sample Pictures\blue hills.jpg")  
  
    'remove worksheet 2 and worksheet 3  
    Application.DisplayAlerts = False  
    CType(Me.Application.ActiveWorkbook.Sheets(2), Excel.Worksheet).Delete()  
    CType(Me.Application.ActiveWorkbook.Sheets(2), Excel.Worksheet).Delete()  
    CType(Me.Application.ActiveWorkbook.Sheets(1), Excel.Worksheet).Name =  
"Bare Bones Time"  
    Application.DisplayAlerts = True  
  
    'hide column and row headers  
    Application.ActiveWindow.DisplayGridlines = False  
    Application.ActiveWindow.DisplayHeadings = False  
End Sub  
  
Private Sub CustomizedData()  
    'set a namedrange  
    Dim formattedRange As Microsoft.Office.Tools.Excel.NamedRange = _  
Me.Controls.AddNamedRange(Me.Range("A1", "D10"), "formattedRange")  
  
    'note range names  
    Dim preFilledRange As Microsoft.Office.Tools.Excel.NamedRange = _  
Me.Controls.AddNamedRange(Me.Range("A2", "A9"), "PreFilledRange")  
  
    'formattedRange.ShrinkToFit = true;  
    formattedRange.ShowErrors()  
  
    'auto fill days of the week  
    Dim firstCell As Microsoft.Office.Tools.Excel.NamedRange = _  
Me.Controls.AddNamedRange(Me.Range("A2"), "FirstCell")  
  
    'note must seed the value  
    firstCell.Select()  
    firstCell.Value2 = "Monday"  
    'note must use the firstcell range that points to A1 for the autofill to  
work  
    firstCell.AutoFill(Application.Range("A2:A6"),  
Excel.XlAutoFillType.xlFillWeekdays)  
  
    preFilledRange.BorderAround(, Excel.XlBorderWeight.xlThin,  
Excel.XlColorIndex.xlColorIndexAutomatic, )
```

```

preFilledRange.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects1,
True, False, True, False, True, True)

    'get a reference to the header cell
    Dim MergeRange As Microsoft.Office.Tools.Excel.NamedRange =
Me.Controls.AddNamedRange(Me.Range("A1", "D1"), "MergeRange")

    'format the header cell
    MergeRange.EntireRow.Font.Bold = True
    MergeRange.Value2 = "Time Sheet [Week - " +
DateTime.Now.ToString("hh/MM/yyyy") + "]"
    MergeRange.EntireRow.Font.Background =
Excel.XlBackground.xlBackgroundTransparent

    'turn off merged prompt dialog and then merge
    Application.DisplayAlerts = False
    MergeRange.Merge(True)
    Application.DisplayAlerts = True

    'setup the range for data entry
    Dim valueRange As Microsoft.Office.Tools.Excel.NamedRange = _
Me.Controls.AddNamedRange(Me.Range("B2", "B6"), "ValueRange")
    valueRange.NumberFormat = "#,###.00"
    valueRange.Font.Bold = True
    valueRange.BorderAround(, Excel.XlBorderWeight.xlHairline,
Excel.XlColorIndex.xlColorIndexAutomatic)
    valueRange.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatColor2,
True, False, True, False, True, True)

    Dim commentRange As Microsoft.Office.Tools.Excel.NamedRange =
Me.Controls.AddNamedRange(Me.Range("B2"), "CommentRange")
    'add the comment
    commentRange.AddComment("Enter your hours worked here.")

End Sub

Private Sub Sheet1_Startup1(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Startup
    'customize the worksheet during start up
    FormatSheet()

    'add the customized data
    CustomizedData()
End Sub
Private Sub CalculateTotal()
    Dim totalRange As Microsoft.Office.Tools.Excel.NamedRange = _
Me.Controls.AddNamedRange(Me.Range("B2", "B6"), "TotalRange")

    Dim fields() As Integer = New Integer() {1, 2, 3, 4, 5}

    totalRange.Subtotal(1, Excel.XlConsolidationFunction.xlSum, fields, , ,
Excel.XlSummaryRow.xlSummaryBelow)
End Sub
Private Sub EmailDocument()
    'calculate the hours worked

```

```
CalculateTotal()

    If MessageBox.Show("Are you sure you want to submit ?", "Time sheet
submission", MessageBoxButtons.YesNo) = DialogResult.Yes Then
        'email the document to management
        Me.Application.ActiveWorkbook.SendMail("someone@example.com",
DateTime.Now.ToString("hh/MM/yyyy"))
    End If

End Sub

Private Sub Sheet1_Shutdown1(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Shutdown
    'The user is through so email the document to recipient
    EmailDocument()
End Sub
```

C#

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace BareBonesTM
{
    public partial class Sheet1
    {
        /// <summary>
        /// This routine formats the worksheet
        /// </summary>
        private void FormatSheet()
        {
            //take care of some aesthetic issues
            Application.DisplayFormulaBar = false;
            Application.DisplayFunctionToolTips = false;
            Application.DisplayScrollBars = false;
            Application.DisplayStatusBar = false;

            //make some customizations
            Application.Rows.Worksheet.SetBackgroundPicture(@"C:\Documents and
Settings\All Users\Documents\My Pictures\Sample Pictures\blue hills.jpg");

            //remove worksheet 2 and worksheet 3
            Application.DisplayAlerts = false;
            ((Excel.Worksheet)this.Application.ActiveWorkbook.Sheets[2]).Delete();
            ((Excel.Worksheet)this.Application.ActiveWorkbook.Sheets[2]).Delete();
            ((Excel.Worksheet)this.Application.ActiveWorkbook.Sheets[1]).Name =
            "Bare Bones Time";
            Application.DisplayAlerts = true;

            //hide column and row headers
            Application.ActiveWindow.DisplayGridlines = false;
```

```

        Application.ActiveWindow.DisplayHeadings = false;
    }
    /// <summary>
    /// This routine customizes the worksheet with prefetched data
    /// </summary>
    private void CustomizedData()
    {
        //set a namedrange
        Microsoft.Office.Tools.Excel.NamedRange formattedRange =
        this.Controls.AddNamedRange(this.Range["A1", "D10"], "formattedRange");

        //note range names
        Microsoft.Office.Tools.Excel.NamedRange preFilledRange =
        this.Controls.AddNamedRange(this.Range["A2", "A9"], "PreFilledRange");

        //formattedRange.ShrinkToFit = true;
        formattedRange.ShowErrors();

        //auto fill days of the week
        Microsoft.Office.Tools.Excel.NamedRange firstCell =
        this.Controls.AddNamedRange(this.Range["A2", missing], "FirstCell");

        //note must seed the value
        firstCell.Select();
        firstCell.Value2 = "Monday";
        //note must use the firstcell range that points to A1 for the autofill
to work
        firstCell.AutoFill(Application.get_Range("A2:A6", missing),
        Excel.XlAutoFillType.xlFillWeekdays);

        preFilledRange.BorderAround(missing, Excel.XlBorderWeight.xlThin,
        Excel.XlColorIndex.xlColorIndexAutomatic, missing);

        preFilledRange.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects1,
        true, false, true, false, true, true);

        //get a reference to the header cell
        Microsoft.Office.Tools.Excel.NamedRange MergeRange =
        this.Controls.AddNamedRange(this.Range["A1", "D1"], "MergeRange");

        //format the header cell
        MergeRange.EntireRow.Font.Bold = true;
        MergeRange.Value2 = "Time Sheet [Week - " +
        DateTime.Now.ToString("hh/MM/yyyy") + " ]";
        MergeRange.EntireRow.Font.Background =
        Excel.XlBackground.xlBackgroundTransparent;

        //turn off merged prompt dialog and then merge
        Application.DisplayAlerts = false;
        MergeRange.Merge(true);
        Application.DisplayAlerts = true;

        //setup the range for data entry
        Microsoft.Office.Tools.Excel.NamedRange valueRange =
        this.Controls.AddNamedRange(this.Range["B2", "B6"], "ValueRange");
    }

```



```
        valueRange.NumberFormat = "#,###.00";
        valueRange.Font.Bold = true;
        valueRange.BorderAround(missing, Excel.XlBorderWeight.xlHairline,
Excel.XlColorIndex.xlColorIndexAutomatic, missing);
        valueRange.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatColor2,
true, false, true, false, true, true);

        Microsoft.Office.Tools.Excel.NamedRange commentRange =
this.Controls.AddNamedRange(this.Range["B2", missing], "CommentRange");
        //add the comment
        commentRange.AddComment("Enter your hours worked here.");

    }
    private void Sheet1_Startup(object sender, System.EventArgs e)
    {
        //customize the worksheet during start up
        FormatSheet();

        //add the customized data
        CustomizedData();
    }
    private void CalculateTotal()
    {
        Microsoft.Office.Tools.Excel.NamedRange totalRange =
            this.Controls.AddNamedRange(this.Range["B2", "B6"], "TotalRange");

        int[] fields = new int[] { 1, 2, 3, 4, 5 };
        totalRange.Subtotal(1, Excel.XlConsolidationFunction.xlSum, fields,
missing, missing, Excel.XlSummaryRow.xlSummaryBelow);
    }
    private void EmailDocument()
    {
        //calculate the hours worked
        CalculateTotal();

        if (MessageBox.Show("Are you sure you want to submit ?", "Time sheet
submission", MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            //email the document to management
            this.Application.ActiveWorkbook.SendMail("someone@example.com",
DateTime.Now.ToString("hh/MM/yyyy"), missing);
        }
    }
    private void Sheet1_Shutdown(object sender, System.EventArgs e)
    {
        //The user is through so email the document to recipient
        EmailDocument();
    }

    #region VSTO Designer generated code

    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.

```

```

    /// </summary>
    private void InternalStartup()
    {
        this.Startup += new System.EventHandler(Sheet1_Startup);
        this.Shutdown += new System.EventHandler(Sheet1_Shutdown);
    }

    #endregion

}
}

```

Listing 2-24 Case study application

The code assumes that there is a legitimate file called `blue hills.jpg` given by the above path. If the VSTO runtime cannot find this file, an exception will be thrown.

Aside from the aesthetics, the core code used here is the typical format of an Excel-type application. Some code is used to initialize the spreadsheet and format according to user requirements and some cleanup code is necessary. The bulk of the work is mostly done through end-user interaction with the spreadsheet. Because Excel has most of the functionality already built in, it saves an enormous amount of development time. This is one true benefit of a VSTO-based application.

```

private void Sheet1_Startup(object sender, System.EventArgs e)
{
    //customize the worksheet during start up
    FormatSheet();

    //add the customized data
    CustomizedData();
}

```

The `FormatSheet` method's main purpose is to disguise the appearance of the Excel spreadsheet so that the final application looks more like a windows application than a hybrid spreadsheet. The code uses a default background, but it is easy to pretend that the background is some fancy watermark image drawn up by the company's elite graphics division.

Consider this piece of code:

```

//remove worksheet 2 and worksheet 3
((Excel.Worksheet)this.Application.ActiveWorkbook.Sheets[2]).Delete();
((Excel.Worksheet)this.Application.ActiveWorkbook.Sheets[2]).Delete();
((Excel.Worksheet)this.Application.ActiveWorkbook.Sheets[1]).Name = "Bare Bones Time";

```

The code deletes the second sheet twice. The reason for this is that the default collection contains three worksheets. The first line of code removes worksheet 2. The collection now contains only two worksheets. This is the reason for the second delete routine on worksheet 2. An exception would be thrown at this point if the code attempted to delete worksheet 3.

Chapter 2

Admittedly, the hard-coded approach is an eyesore, but it works. A more elegant approach would be to first find the count of the spreadsheet and then remove one less than the count. An example implementation is trivial and best left as an exercise to the reader.

The `customizeddata` routine handles a large part of the workload. A few points are worth noting. The code makes exclusive use of the `Range` object and named ranges. Named ranges are simply easier to maintain because the code is more readable especially by someone who is not familiar with the code. The code is also easier to update if the range changes, since the change only needs to occur in one place.

The next snippet of code makes use of the `AutoFill` routine.

```
//note must seed the value
firstCell.Select();
firstCell.Value2 = "Monday";
firstCell.AutoFill(Application.get_Range("A2:A6", missing),
Excel.XlAutoFillType.xlFillWeekdays);
```

Excel supports a few predefined `AutoFill` patterns that come in handy. Notice that the range must be seeded with the first value "Monday" in order for `AutoFill` to work correctly. If there is no value, `AutoFill` functionality is simply ignored.

Finally, the range is merged and formatted for consistency and a comment is added to the range so that the user knows what to do. As previously discussed, the code turns off the confirmation prompts when the spreadsheet is modified in the merging process. Merging isn't strictly necessary. However, it is a convenience we can afford, since it allows the code to treat a range as a single cell. Finally, the code presents an option to email the document.

Notice that the code is missing some rather basic functionality, but it is sufficient to illustrate the idea behind VSTO application development. In particular, exception handling is notably absent. For instance, the file load method call assumes that the file is present on disk and the executing application has sufficient permissions to secure the load. In the real world, these assumptions are not valid. You are certainly at liberty to expand and improve upon this implementation as needed. It merely provides a good beginning point for application functionality based on the concepts presented earlier.

Summary

This chapter focused on the basic functionality of the Excel spreadsheet. The basic functionality revolves around data. The code showed how to load data from a variety of different data sources. Excel's flexible data loads stem from the need to manipulate data in different environments.

Excel also contains internal support for XML file formats. The importance of XML as a cross-platform data source cannot be overstated: it is the standard for data exchange among corporate entities. However, the formatting features that form part of the payload are resource intensive during the render phase. Large XML files can take a considerable amount of time to be loaded into the spreadsheet.

Next, we examined the code to manipulate the data. Data loaded into the Excel spreadsheet is available for use in a number of containers. The most widely used container is the Excel `Range` object. The `Range` object contains significant formatting capabilities, and the code presented a few notable examples. Other containers such as the cell object allow fine-grained customization of data.

Finally, we examined a complete coded example that showed how to build bare-bones time management software based on Microsoft Excel. The code showed that the Excel user interface does not necessarily have to look like an Excel spreadsheet. In fact, by applying just the right amount of formatting, the application can approach the aesthetic appeal of a third-party product.

3

Advanced Excel Automation

The Excel machinery has had a few years to evolve. Today, Microsoft Excel contains a robust calculation engine that is well suited for crunching numbers. In addition, the Excel user interface is a powerful yet familiar front end that has grown into a standard piece of furniture on corporate desktops; about 80% of corporations use Microsoft Excel.

Excel can cater to a variety of environments and can derive its data from different formats across different platforms. However, the security aspects of Excel have always been worrying. Recall from Chapter 2 that macros can easily compromise legacy Office applications. A large portion of this chapter examines the security approach that Visual Studio Tools for Office has adopted to mitigate security threats. Once the security aspects are understood, the remainder of the chapter examines the new VSTO controls in depth. Finally, time is spent examining the more advanced side of Excel programming to include adding VSTO menus, server automation, and integration of VSTO with .NET web services.

VSTO Security

Security is an important concern today. However, failure to understand application development from a security perspective coupled with a lack of language support conspire to derail the security initiative. In the end, a compromise leads to second-best security practices, which leaves the application vulnerable to hackers.

VSTO provides two levels of security for application development. One level resides in the .NET Framework and the other forms part of the user interface. We consider the latter first, followed by the former. Together, the two-tiered security fortress significantly reduces the surface area available for attack.

Workbook Password Protection

VSTO offers several ways to protect the contents of data. One approach uses password authentication to restrict access to data in the workbooks. Be aware though, that VSTO does not provide any

Chapter 3

functionality to test the strength of the password associated with the workbook. It simply acts as a storage vessel for the password. If you require such functionality, you must implement it yourself.

Security initiatives should be designed and planned for. It should not be an afterthought. Security is also especially important since Excel allows up to 256 users to view a single workbook.

Consider a password implementation example, shown in Listing 3-1. For this example, the code assumes that the password property has been set previously and is available for use by the calling code.

Visual Basic

```
Public Overrides Property Password() As String
    Get
        Return MyBase.Password
    End Get
    Set(ByVal value As String)
        If Not String.IsNullOrEmpty(value) Then
            MyBase.Password = value
        Else
            Throw New ArgumentException("Password Cannot Be Blank", "Password")
        End If
    End Set
End Property

Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
Handles Me.Startup
    'Protect The Workbook
    Dim password As String = Me.Password
    Me.Protect(password)
    'Unprotect The Workbook
    Me.Unprotect(password)
End Sub
```

C#

```
string passwordData = string.Empty;
public string Password
{
    get { return passwordData; }
    set
    {
        if (!String.IsNullOrEmpty(value)
            passwordData = value;
        else
            throw new ArgumentNullException("Password Cannot Be Blank",
"Password");
    }
}

private void Sheet1_Startup(object sender, System.EventArgs e)
{
    //Protect The Workbook
```

```

        string password = this.Password;
        this.Protect(password, missing, missing,missing, missing,
missing,missing, missing, missing, missing,missing,missing, missing,
missing,missing,missing,missing);
        //Unprotect The Workbook
        this.Unprotect(password);
    }

```

Listing 3-1: Excel's internal password support

In Listing 3-1, a property `Password` is used to set or retrieve the password attached to the workbook resource in Microsoft Excel. Then, workbook protection is applied and removed. For illustrative purposes, a property was used to manage the password retrieval process. Since properties ultimately map to method calls deep inside the .NET Framework, choosing a property implementation over a method implementation for the password management process is purely aesthetic, with no significant performance or readability benefit.

Assuming that you have negotiated your way around this issue, a more sinister villain is lurking. Although passwords are encrypted, Excel passwords are notably easy to crack. In fact, there are a number of free software applications available for that express purpose. There is also no limit imposed on the number of failed attempts as is common in most security-conscious applications. For instance, the Excel application does not lock the user out after three failed attempts. Password-cracking applications can usually brute-force their way into sensitive data.

If you choose to implement secure passwords, at the very minimum, you should keep track of the failed attempts. One good approach is to keep a counter variable that is associated with the logged-on user's credentials. The counter variable represents the number of failed attempts, and the user's credentials indicate the user who is making the failed attempts. Once the maximum number of failed attempts is reached, no more attempts are allowed by that user based on his or her login credentials for a set period of time—24 hours for instance. You may use the `Datetime` object in .NET to keep track of the elapsed time during the failed attempts for a specific user.

Worksheet Security

Data protection may also be enforced at the worksheet level. This approach provides more fine-grained control over data access when compared to protection at the workbook level. Worksheets may have protection applied to them through the `Protection` object. Consider the example in Listing 3-2.

Visual Basic

```

Dim userPassword As String = Me.Password
Globals.Sheet1.Protect(userPassword)

```

C#

```

String userPassword = Password;
Globals.Sheet1.Protect(userPassword, missing, missing, missing, missing, missing,
missing, missing, missing, missing, missing, missing, missing, true, missing,
missing);

```

Listing 3-2: Password protecting worksheets

After the code in Listing 3-2 has been executed, the contents of `sheet1` are now in a protected state. Any attempt to change the contents of sheet 1 results in an error dialog shown in Figure 3-1. If you care to examine the parameters to this method, you will notice that the `protect` method is flexible enough to protect the contents of the data on the worksheet yet allow the end user to manipulate the cells. For instance, parameter 7 allows columns to be formatted on the worksheet while it is protected. This allows a user to increase the width of a particular column so that its contents can be displayed completely. You may use the IntelliSense engine in concert with the help documentation to find information about each parameter.



Figure 3-1

If the password is correct, the range is protected, otherwise it is not. The `protect` method accepts 16 parameters. We consider each in turn since the sum of these parameters adds up to varying levels of protection on the worksheet.

Parameter	Definition
Password	All passwords are case sensitive. An empty string may be passed in that signifies that no password is associated with the worksheet. As mentioned previously, the Excel password mechanism is not adequate for applications that run in the public domain.
DrawingObjects	It is possible to protect only the embedded objects in the spreadsheet. These objects include, but are not limited to, images, icons, controls, and documents. For instance, you may choose to protect an embedded Word document inside the Excel spreadsheet to prevent it from being copied.
Contents	Locking the contents of the spreadsheet prevents manipulation of the cells. However, this may be overridden by the <code>AllowFormattingCells</code> property so that locked cells can be adjusted for readability purposes.
Scenarios	Scenarios or modeling can be protected by setting this value to true. Although most modeling is interactive, some scenarios are best left out of the user's control.
UserInterfaceOnly	This property essentially protects the user interface section but not embedded code running in the user interface section. A value of <code>true</code> insulates the user interface from tampering by the user, but the user is able to modify macros or embedded code and functions in the document proper.

Parameter	Definition
<code>AllowFormattingCells</code>	This property allows the user to format any cell on a protected worksheet. One benefit of this approach is that certain content may not be cropped by the cell. A protected spreadsheet does not allow the users to view cropped content. By setting this property to <code>true</code> , the user can adjust the cell content through the cell adjuster, while the contents of the cell remain in read-only mode.
<code>AllowFormattingColumns</code>	This property provides the same functionality as the <code>AllowFormattingCells</code> property, except that it targets columns instead of cells.
<code>AllowFormattingRows</code>	This property provides the same functionality as the <code>AllowFormattingCells</code> property, except that it targets rows instead of cells.
<code>AllowInsertingColumns</code>	The property allows the user to insert columns on a protected worksheet.
<code>AllowInsertingRows</code>	The property allows the user to insert rows on a protected worksheet.
<code>AllowInsertingHyperlinks</code>	The property allows the user to insert hyperlinks on the worksheet.
<code>AllowDeletingColumns</code>	The property allows the user to delete columns on the protected worksheet if every cell in that column is unprotected. The method throws an exception otherwise.
<code>AllowDeletingRows</code>	The property allows the user to delete rows on the protected worksheet if every cell in that row is unprotected. The method throws an exception otherwise.
<code>AllowSorting</code>	The property allows the user to sort the contents of the range if every cell is unprotected.
<code>AllowFiltering</code>	The property allows the user to set filters on the protected worksheet. There are some restrictions on filtering. For instance, users can not enable or disable an autofilter.
<code>AllowUsingPivotTables</code>	The property allows the user to use pivot table reports on the protected worksheet.

You must understand that Excel allows the end user to override the protection mechanism that has been applied in code. For instance, the code in Listing 3-2 used to protect the worksheet may be removed quite easily by selecting Tools ⇨ Protection ⇨ Unprotect Sheet. The reason for this easy breach is that the code in Listing 3-2 passes in a default value as the first parameter. The first parameter accepts password values. If this value is empty, the protection mechanism may be overridden by the end user. The work-around is to pass a nonempty string. When the end user tries to remove the protection, Excel will prompt the end user to enter a password. If the password matches the value that you have provided in the protection function, the worksheet is unprotected; otherwise, it is not.

You should note also that passwords should never be hard-coded in the code. There are several reasons for this, but perhaps the most important is that the code may be decompiled to expose the password. There are many free utilities that decompile .NET code. In fact, the .NET Framework ships with a free decompiler utility. The better approach is to encrypt the password and store it in a secure place such as a database. You retrieve this encrypted key from the database, decrypt it, and pass it to the `Protect` method to protect the worksheet. MSDN provides further resources on encryption/decryption and security best practices related to software development.

Protection through Hidden Worksheets

The ability to use hidden worksheets as a protection mechanism can work well in certain circumstances. For instance, a worksheet may contain proprietary formulas that are required for worksheet calculations. However these formulas may need to be hidden from the end user. For such scenarios, the code in Listing 3-3 can serve as a protection mechanism.

Visual Basic

```
'retrieve end user values
Dim userRange As Excel.Range =
DirectCast(Globals.ThisWorkbook.Application.Range("Sheet1!A1:A3"), Excel.Range)
userRange.Copy()
Dim calcRange As Excel.Range = DirectCast
(Globals.ThisWorkbook.ThisApplication.Range("Sheet2!A1"), Excel.Range)
calcRange.PasteSpecial(Excel.XlPasteType.xlPasteAll,
Excel.XlPasteSpecialOperation.xlPasteSpecialOperationAdd, False, False)
Dim formulaRange As Excel.Range = DirectCast
(Globals.ThisWorkbook.Application.Range("Sheet2!A4"), Excel.Range)
formulaRange.Formula = "=AVERAGE(A1:A3)"
Globals.Sheet2.Visible = Excel.XlSheetVisibility.xlSheetHidden
```

C#

```
//retrieve end user values
Excel.Range userRange =
(Excel.Range)Globals.ThisWorkbook.Application.get_Range("Sheet1!A1:A3", missing);
userRange.Copy(missing);
Excel.Range calcRange =
(Excel.Range)Globals.ThisWorkbook.ThisApplication.get_Range("Sheet2!A1", missing);
calcRange.PasteSpecial(Excel.XlPasteType.xlPasteAll,
Excel.XlPasteSpecialOperation.xlPasteSpecialOperationAdd, false, false);
Excel.Range formulaRange =
(Excel.Range)Globals.ThisWorkbook.Application.get_Range("Sheet2!A4", missing);
formulaRange.Formula = "=AVERAGE(A1:A3)";
Globals.Sheet2.Visible = Excel.XlSheetVisibility.xlSheetHidden;
```

Listing 3-3: Hiding spreadsheets

In Listing 3-3, the data entered by an end user is collected using `userRange`. The contents of the range are copied to the system clipboard. Then the paste operation is applied to the hidden sheet using the reference `calcRange`. In this example, the code assumes that `formulaRange` contains some information—a proprietary company formula for instance—and that information must be hidden from the end user. For illustrative purposes, a simple Average formula is used to average the values contained in rows 1 through 3 of column A on sheet 1, and the results are stored in cell A4 of sheet 2.

Interestingly, if the code is executed with no data on sheet 1, a #DIV/0 is written to cell 4 on sheet 2. We spend time later on in the chapter examining the reason for error messages in cells on the spreadsheet. For now, we note that Excel assumes that empty cells contain undefined values. When the average is performed on the cell, the sum of the cells should be divided by the number of occurrences of the data. In Listing 3-3, that should translate roughly to the addition of 3 undefined values followed by a division of 3. Since the number of cells is clearly a finite answer (3), the actual error message generated by Excel is incorrect. It is not a divide-by-zero error. However, it is most likely an undefined result.

Another major assumption here is that assembly permissions have been correctly configured. Development environments are automatically configured to work correctly at the time of installation. However, this convenience does not extend to deployed applications. These applications need to have permissions configured appropriately; otherwise, the application will not run.

Worksheet protection and the line of reasoning that supports its use may be extended to spreadsheet cells as well. Instead of merely hiding the worksheet, it is possible to hide just the cells that contain sensitive information. For instance, the `calcRange.Cells.Hidden` property is also a good approach. However, you should note that the cells can be made visible again by the end user.

Protecting Ranges

Protection at the workbook and worksheet level is convenient, but it is a coarse solution since it allows protection for the entire workbook or worksheet. For more flexibility, Excel allows worksheet ranges in the workbook to be protected. For instance, you may be interested in preventing a particular column in the worksheet from being edited. Since the column is effectively a range of cells, you may apply protection to the column by locking the range. This is certainly more appealing than locking the entire worksheet. The code to perform this protection is relatively simple and may be used to lock a range where the range may consist of a group of cells or a single cell. Borrowing from the range example in Listing 2-4, consider the example in Listing 3-4.

Visual Basic

```
Dim formattedRange As Microsoft.Office.Tools.Excel.NamedRange =
Globals.Sheet1.Controls.AddNamedRange(Me.Application.Range("A1", "D10"),
"formattedRange")
formattedRange.Locked = True
    Dim userPassword As String = "My Password"
    Globals.Sheet1.Protect(userPassword, AllowSorting:=True
```

C#

```
Microsoft.Office.Tools.Excel.NamedRange formattedRange =
Globals.Sheet1.Controls.AddNamedRange(this.Application.get_Range("A1", "D10"),
"formattedRange");
formattedRange.Locked = true;
string userPassword = "My Password";
Globals.Sheet1.Protect(userPassword, missing, missing, missing, missing, missing,
missing, missing, missing, missing, missing, missing, missing, missing,
missing);
```

Listing 3-4: Range protection

Chapter 3

The code first obtains a reference to a named range and then applies a lock on the range. Finally, the `Protect` method is called. After the code is run, only the specific range “A1, D10” is protected. The end user is free to manipulate other parts of the worksheet.

There is a quirk hidden in the code that has been taken for granted so far. While it may be second nature to Office developers, it is less well known to outsiders. Consider Listing 3-5, which has been modified from Listing 3-4.

Visual Basic

```
Dim formattedRange As Microsoft.Office.Tools.Excel.NamedRange =
Globals.Sheet1.Controls.AddNamedRange(Globals.Sheet1.Range("A1", "D10"),
"formattedRange")
formattedRange.Locked = True
MessageBox.Show(formattedRange.AllowEdit.ToString())
Dim userPassword As String = Password
Globals.Sheet1.Protect(userPassword)
MessageBox.Show(formattedRange.AllowEdit.ToString())
```

C#

```
Microsoft.Office.Tools.Excel.NamedRange formattedRange =
Globals.Sheet1.Controls.AddNamedRange(Globals.Sheet1.Range["A1", "D10"],
"formattedRange");
formattedRange.Locked = true;
MessageBox.Show(formattedRange.AllowEdit.ToString());
String userPassword = Password;
Globals.Sheet1.Protect(userPassword, missing, missing, missing,
missing, missing, missing, missing, missing, missing, missing,
true, missing, missing);
MessageBox.Show(formattedRange.AllowEdit.ToString());
```

Listing 3-5: Range protection recommendation

If you run this code, the first message box displays a dialog with `true`, followed by another message box with the value `false`. The behavior underscores the importance of calling the worksheet `Protect` method to actually implement worksheet protection. You may think of the actual call as a recommendation that is implemented after the protection method is called. Failing to call the `Protect` method results in the VSTO runtime simply ignoring the `Locked` recommendation resulting in an unprotected range.

It's also possible to simply store sensitive data in an external module or assembly. For instance, a library containing sensitive company information may be exposed through a managed add-in and stored on a secure web application server. The calling code in the client piece will then need to make requests to this external library to retrieve data. We consider one such example later in the chapter. For now, we note that there is a performance overhead associated with making remote requests.

VSTO Security through the .NET Framework

The security implementation in the .NET Framework is impressive. This is the reason why the VSTO uses it for security. Here, we stop to consider the practical implementations of security that allow developers to

build VSTO applications that are secure and fall within the bounds of implicit end-user expectations on application software.

Configuring Secure Applications

A VSTO-based application consists of a document and a .NET assembly. For the application to work correctly, the .NET assembly must be granted full trust. By default, applications that are developed on a local machine are automatically granted full trust so that the development environment and the other associated tools such as the debugger can execute correctly. However, there is no automatic configuration for a deployed application. In fact, by default, the .NET assembly is not trusted and will not be loaded by the Framework.

The .NET Framework can configure assemblies to run with full trust, with partial trust, or untrusted. Untrusted assemblies will not load. Partially trusted assemblies can only perform specific tasks on a user machine. Full trust has unbounded access to the end-user system. You should note that VSTO applications can only function under full trust. Partially trusted and untrusted assemblies will not run.

The .NET Framework departs from the VBA approach because the Common Language Runtime determines the level of trust to grant assemblies. VBA automatically extends trust to all executing code. The Framework examines a number of criteria before applying a trust level. The criteria, or evidence, are the publisher of the document, the site it is being executed from, and the authenticity of the executing assembly among other things. If adjustments are made to these parameters in anyway, such as changing the location of the executing assembly, the assembly evidence will be invalidated and the assembly will not load.

Strongly Named Assemblies

There are a number of options that may be used to deploy applications to ensure that they are secure. One option is to provide a strongly named assembly so that it is automatically granted full trust by the Framework. To create a strongly named assembly, use the strong name utility that ships free with the .NET Framework. Here is an example of its use:

```
C:\program files\Microsoft Visual Studio .NET 2005\SDK\v2.0\Bin\Sn.exe -k mynewkey.snk
```

Listing 3-9 creates a new, random key pair and stores the value in `mynewkey.snk`. The generated key is a private key and should be stored in a secure place. A secure place is typically a solid steel bank vault located underground in the middle of a desert. However, if you cannot afford such drastic measures, you should store this key in a secure location that is typically located off-site. For obvious reasons, if the private key is stolen, it is possible to create a malicious assembly and sign it so that it appears to come from you. This sort of masquerading malfeasance is a nightmare to detect.

Next, you include this assembly directive in the `assemblyinfo` manifest file. Place the line of code in Listing 3-10 at the end of the `assemblyinfo` file. If you are not familiar with the `assemblyinfo` file, it is automatically created for every Visual Studio .NET project and contains assembly specific information. The `assemblyinfo` file is preloaded by default so the new key is automatically loaded and prepared for use. Here is an example usage:

Visual Basic

```
<Assembly: AssemblyKeyFile("mynewkey.snk")>
```

C#

```
[Assembly: AssemblyKeyFile("mynewkey.snk")]
```

A strong name assembly provides a globally unique identifier that guarantees, at the time of use, that the assembly is authentic and has not been tampered with. It does not guarantee that the assembly is free from bugs or that the assembly will respect your privacy. For more help with these topics, consult the MSDN documentation.

Security Certificates

Digital certificates also guarantee the authenticity of the publisher of the assembly. Although a certificate cannot authenticate the code, it can ensure that the assembly comes from the publisher and that it has not been tampered with in any way since it was digitally signed. Digital certificates may be obtained from third-party certification authorities for a fee. One example of a certification authority is VeriSign.

If you just need to test certificate implementation locally, consider creating a test certificate using the `makecert.exe` utility that ships free with the .NET Framework. Consider an example of its usage:

```
C:\program files\Microsoft Visual Studio .NET 2005\SDK\v2.0\Bin\makecert  
testCert.cer
```

To create a software publisher certificate from the generated x.508 cert, use this command:

```
C:\program files\Microsoft Visual Studio .NET 2005\SDK\v2.0\Bin\cert2spc  
testCert.cer testCert.spc
```

To sign the certificate, use this command:

```
C:\program files\Microsoft Visual Studio .NET 2005\SDK\v2.0\Bin\signcode -spc  
testCert.spc -k testKey "C:\testAssembly.dll"
```

Software publisher certificates (SPCs) are available from a certification authority. Part of the process involves generating a key pair. The certification authority will need to verify your identity. A legal agreement must be signed by you as well before a certificate will be issued on your behalf.

The SPC that you receive is an industry-standard x.509 certificate format with Version 3 extensions. The certificate contains your public key and identifies you as the responsible party.

The test certificate should not be deployed in a production environment. Among the more important reasons to avoid this penny-skipping behavior is that it compromises the security of the application.

Deploying Secure Applications

Recall from the previous section that a VSTO-based application has two parts—the .NET assembly and the Microsoft Office document. These two parts are loosely coupled so that there is no obligation for them

to exist on the same physical system. In fact, they may be deployed separately. In any case, two conditions must be satisfied. The Common Language Runtime executing the code must be able to find the Office customization assemblies. Once the assemblies have been found, the calling code must have the appropriate permissions in order for the CLR to load the customization assembly. An Office customization assembly is produced when the code in the Office code-behind file is compiled into a .NET binary.

If the VSTO assembly is located in the Global Assembly Cache or in the application folder, no extra work is required. Otherwise, the CLR searches for the assembly following a specific pattern determined by the application project settings. If your assembly is not located in one of these known areas such as the application folder, you will need to explicitly tell the CLR where to look for the assembly. Open the Visual Studio .NET project and navigate to the property pages. Select the "Reference paths." Add the appropriate path to the folder text box, and select Add Folder. The path will then be added to the "Reference Path list box, as shown in Figure 3-2.

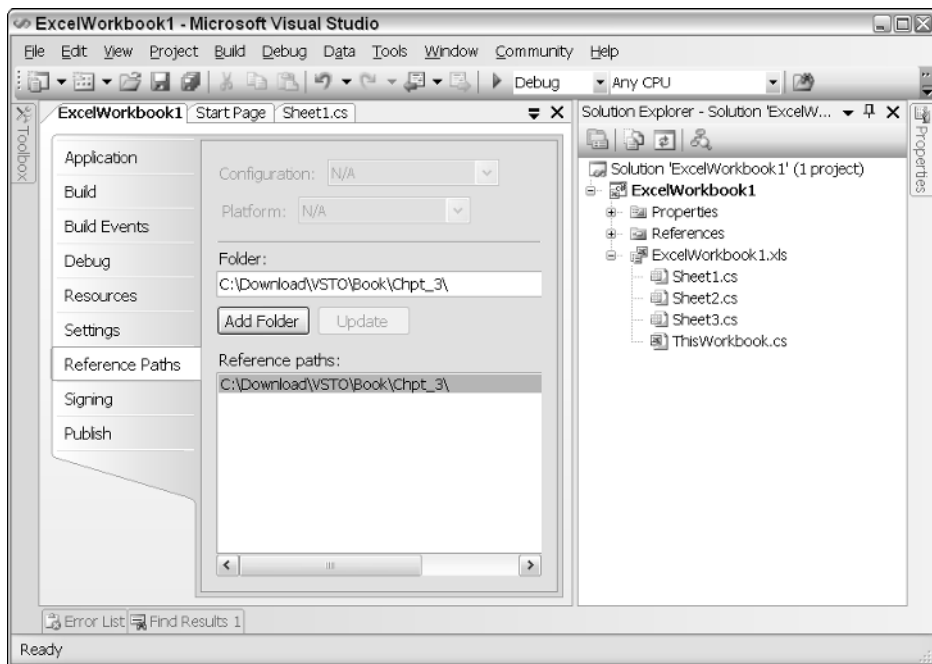


Figure 3-2

The previous steps work well for applications that are not designed to use the web. However, you will need to add some extra configuration if you are developing an Office application that uses the web and is required to continue working in offline mode. The extra step involves configuration of the web server. Once the web server has been configured, the extra steps allow Internet Explorer to download and cache the assemblies on the end-user system. To configure your Office customization assembly for offline usage, open Internet Information Services (IIS) administration tool and make the following changes as shown in Figure 3-3.

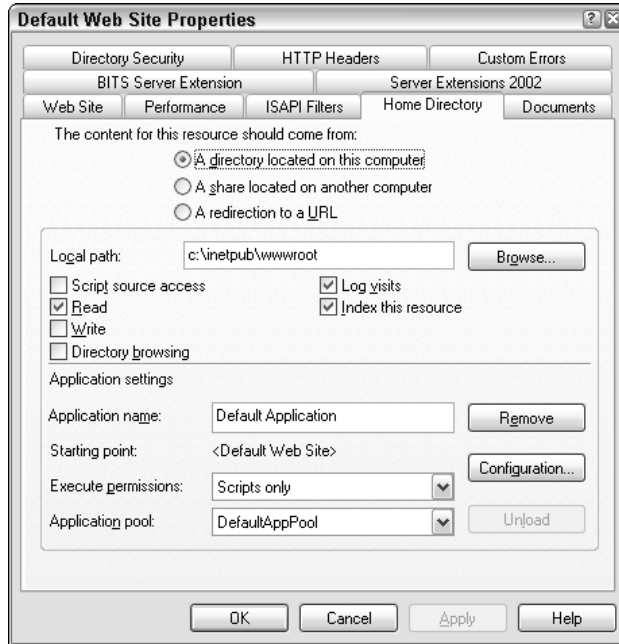


Figure 3-3

Updating applications is as easy as copying the new .NET assembly to the existing location. The .NET Framework will automatically use the new assembly the next time the application is run. For offline scenarios, VSTO caches the data required into cache storage on the end-user system. The application works with the offline cache as needed and synchronizes access when a connection is established. This is all handled automatically, and no code is required.

Working with Excel Formulas

Excel has always allowed the user to apply mathematical formulas to logical groups of cells. In fact, formulas have grown to include the Database, Date and Time, External, Engineering, Financial, Information, Logical, Look up References, Trigonometry, Statistical, and Text and Data logical groups. Visual Studio Tools for Microsoft Office exposes this functionality through several objects, including the *Worksheet* and *Range* objects. The formula machinery applies the specified formula to the target and returns a reference to the result of the operation.

Using Formulas at Design Time

.NET developers who are not familiar with Office technology may not be aware that Excel is able to respond to formula manipulation at design time. In fact, entering formulas in design mode offers a performance boost when compared to assigning formulas at runtime. The approach is especially

recommended if the range that is being used for computation isn't going to change significantly. For real-world applications, this is quite common. For instance, a spreadsheet application that computes auto loans usually contains static input and calculation ranges that are good candidates likely to benefit from design-time formula addition.

To enter a formula at design time, first create a project and navigate to the design view of the spreadsheet. Enter the formula `"=SUM(A1:A3)"` in the spreadsheet and press `Ctrl+Enter`. If the range contains data, the calculation is performed. At runtime, the calculations are automatically performed as well. Since the formula is assigned as a string variable, the quotes around the expression are required. However, an incorrect expression will not be immediately flagged at a compile time. Instead, the application will throw an exception at runtime when the calculation is performed.

Formulas that are added to a spreadsheet may be made visible by pressing `Ctrl + `` (grave accent) in a cell. The sequence of keystrokes brings up the Formula Auditing toolbar shown in Figure 3-4. The Formula Auditing toolbar can be used to customize formulas to the target range.

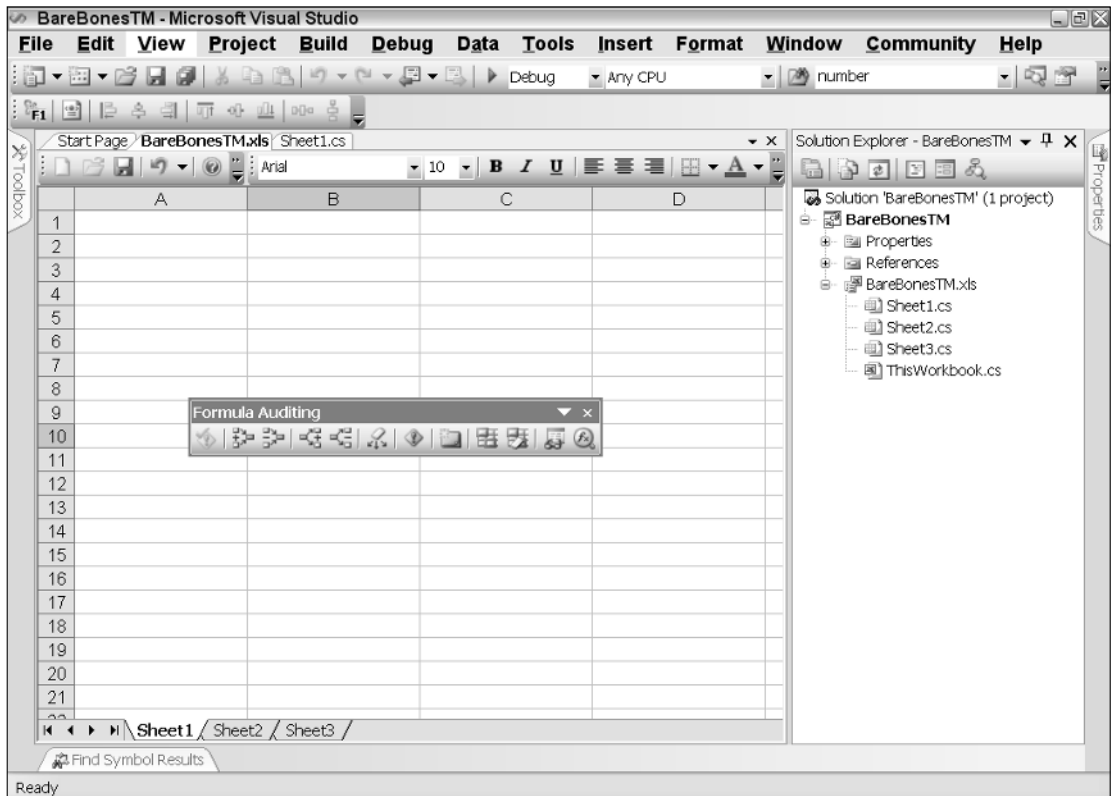


Figure 3-4

Using Formulas at Runtime

The formula manipulation can also occur in the Office code-behind file. Consider Listing 3-6.

Visual Basic

```
Dim calcRange As Excel.Range = CType(Me.Application.Range("A4"), Excel.Range)
calcRange.Formula = "=SUM(A1:A3)"
calcRange.Calculate()
```

C#

```
Excel.Range calcRange = (Excel.Range)Globals.ThisWorkbook.ThisApplication
.get_Range("A4", missing);
calcRange.Formula = "=SUM(A1:A3)";
calcRange.Calculate();
```

Listing 3-6: Runtime formula usage

The example sets a reference to the cell A4. Then a formula "`=SUM(A1:A3)`" is applied to the range. The final line of code invokes the calculation. Listing 3-13 is functionally equivalent to adding the formula from the design view, as discussed in the previous section. It's important to note that this code contains some deficiencies that affect performance. As soon as the target range receives a formula input, the calculation is performed. Therefore, it is unnecessary to call the `Calculate` method again.

The code was presented because it is a common programming error that can result in decreased efficiency of the application especially with large worksheets that require a lot of calculation. The `Calculate` method should only be invoked when a manual calculation or a refresh is required. Also note that the result of the calculation will be placed in the cell A4 because the range points to that particular cell.

Consider this refinement to the current example, shown in Listing 3-7, where only the positive integers need to be summed.

Visual Basic

```
Dim calcRange As Excel.Range = CType(Globals.ThisWorkbook.ThisApplication
.Range("A4", Type.Missing), Excel.Range)
calcRange.Formula = "=SUMIF(A1:A3, "> 0")"
```

C#

```
Excel.Range calcRange = (Excel.Range)Globals.ThisWorkbook.ThisApplication
.get_Range("A4", missing);
calcRange.Formula = "=SUMIF(A1:A3, \">>0\")";
```

Listing 3-7: Conditional formula usage

In the example, notice how the code escapes the quotes so that they are presented in the correct format to the calculation engine. Without the backslash character (C#) or double quotation marks (VB), a runtime exception would be thrown with a cheesy error message. The final result is a summation of the range, excluding values that are negative.

The use of unfamiliar formulas such as `SUMIF` may pose a problem because the parameter types may not be obvious. For instance, it is rather easy to deduce that `SUM` should contain two values given by a

reference to two cells. But it is less obvious for `SUMIF`. If you must use formulas that you are not familiar with, first look up the format using the Excel help file. At the time of this writing, the help documentation specifies the correct number and type of arguments necessary for each function.

Working with the `WorksheetFunctions` Method

Microsoft Excel supports over 300 different types of worksheet functions. All of these functions may be used in formula manipulation inside the spreadsheet. Consider Listing 3-8.

Visual Basic

```
MessageBox.Show(Me.Application.WorksheetFunction.Ceiling(20.3,20.4).ToString())
```

C#

```
MessageBox.Show(this.Application.WorksheetFunction.Ceiling(20.3,20.4).ToString());
```

Listing 3-8: `WorksheetFunctions` implementation

Listing 3-15 calculates the mathematical ceiling (the greater of the two numbers) of the input parameters and displays the result on the screen. Although the single line of code is tempting to write, you should make every effort to resist this approach because it is less readable and presents challenges during debugging. For instance, if an exception occurs in Listing 3-15 there is no way to easily identify the source of the exception. This can make debugging extremely painful. Exceptions are common when manipulating worksheets due mostly to invalid parameters, but the error messages presented by the exception object in VSTO typically contain very little usable information about the cause of the exception. Instead, your code approach should follow Listing 3-9.

Visual Basic

```
Dim wrkFunc As Excel.WorksheetFunction = Me.Application.WorksheetFunction
If wrkFunc IsNot Nothing Then
    Dim d As Double = wrkFunc.Ceiling(20.3, 20.4)
    MessageBox.Show(d.ToString())
End If
```

C#

```
Excel.WorksheetFunction wrkFunc = this.Application.WorksheetFunction;
if (wrkFunc != null)
{
    Double d = wrkFunc.Ceiling(20.3,20.4);
    MessageBox.Show(d.ToString());
}
```

Listing 3-9: Correctly structured `WorksheetFunctions` implementation

Excel Spreadsheet Error Values

Sooner or later, during spreadsheet formula manipulation, the Excel user interface may indicate an error in one or more of the cell values. It is especially troubling if the cell in question received its input from code that you wrote. Although seasoned Microsoft Excel users are familiar with these error values,

Chapter 3

the developer may not necessarily be familiar with them. In order to address these issues, you need to understand the cause of the error.

Excel can generate specific errors based on the data in the cell. The following table lists these values.

Error code	Description
#####	The cell contains a time or date format that is wider than the minimum width of the cell. Resize the cell to fit by using the auto width property of the cell.
#CIRC	A formula in the cell refers to itself either directly or indirectly. Repair the circular reference.
#DIV/0!	A formula is being divided by zero. Division by zero is not permitted in the calculation engine. Use a more appropriate value.
#N/A	The formula is not available or may be deprecated. Use another formula or a default value of #N/A for cells that may be used in calculations.
#NAME?	An incorrect formula is used. Correct the formula so that it is recognizable by the spreadsheet.
#NULL!	An invalid intersection. Make sure that the intersection is valid.
#NUM!	An invalid piece of text is being interpreted as a number. Add valid numbers only.
#REF!	Incorrect cell reference has been specified. Correct the cell reference. Make certain that your correction does not cause a circular reference.
#VALUE!	Incorrect arguments or operands have been specified. Use the correct arguments or operands.

Consider an example that forces a circular reference in a spreadsheet. Listing 3-10 sets up a range to force a circular reference error.

Visual Basic

```
Me.Application.Range("A1").Formula = "=B1"  
Me.Application.Range("B1").Formula = "=A1"
```

C#

```
this.Application.get_Range("A1", missing).Formula = "=B1";  
this.Application.get_Range("B1", missing).Formula = "=A1";
```

Listing 3-10: Circular reference error

The Microsoft Excel engine actively monitors the spreadsheet for error values and will call attention to the problem. On later versions of Excel 2002 and 2003, a circular reference toolbar appears instead of the #CIRC warning so that the end user can correct the circular reference, as shown in Figure 3-5.

The ##### can be handled with a clever use of one of the cell's properties. Consider the example in Listing 3-11, which prints a value to the cell that is wider than the default width resulting in a ##### error.

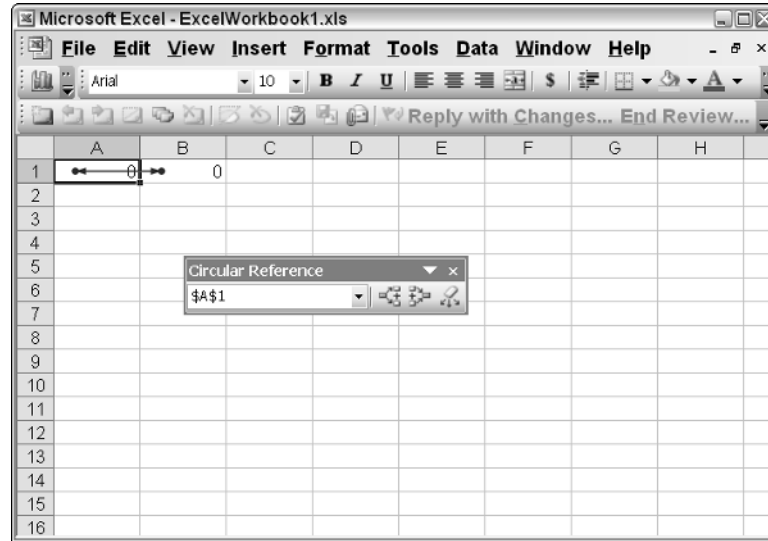


Figure 3-5

Visual Basic

```
'set a namedrange
Dim fittedRange As Microsoft.Office.Tools.Excel.NamedRange =
Globals.Sheet1.Controls.AddNamedRange(Me.Application.Range("A1", "B2"),
"fittedRange")
fittedRange.Value = DateTime.Now.ToString("mm-DD-yyyy hh:mm:ss")
fittedRange.ShrinkToFit = True
```

C#

```
//set a namedrange
Microsoft.Office.Tools.Excel.NamedRange fittedRange =
Globals.Sheet1.Controls.AddNamedRange(this.Application.get_Range("A1", " B2"),
"fittedRange");
fittedRange.Value2 = DateTime.Now.ToString("mm-DD-yyyy hh:mm:ss");
fittedRange.ShrinkToFit = true;
```

Listing 3-11: Fixing error values in ranges

The `ShrinkToFit` property forces the contents of the cell to fit within the dimensions of the cell. Internally, the spreadsheet simply adjusts the font attributes of the value or the piece of text in the cell so that it fits. One side effect is that the cell contents may no longer be readable. Exercise caution when using this feature.

Responding to Events

As mentioned earlier, the .NET assembly is linked to the Excel Workbook object. Applications based on Excel can, therefore, respond to two different event paths. One path stems from the .NET Framework

Chapter 3

linked to the document and the other path stems from the Native Excel object. These events are housed in the `ThisWorkbook` class. Code that needs to respond to Excel workbook events must be placed in this class.

The `ThisWorkbook` class provides two default event handlers whose responsibility is to handle the `ThisWorkbook_Open` and `ThisWorkbook_BeforeClose` events. These events fire when the Excel workbook is opened and immediately before the workbook is closed. You may use these event handlers to provide initialization and clean up code for the workbook. Consider the example in Listing 3-12.

Visual Basic

```
Private Sub ThisWorkbook_Startup(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Startup
    Dim namedRange1 As Microsoft.Office.Tools.Excel.NamedRange =
Globals.Sheet1.Controls.AddNamedRange(Globals.Sheet1.Range("A1"), "NamedRange")
    AddHandler namedRange1.Change, AddressOf Me.namedRange1_Change
End Sub
```

C#

```
private void ThisWorkbook_Startup(object sender, System.EventArgs e)
{
    Microsoft.Office.Tools.Excel.NamedRange namedRange1 =
Globals.Sheet1.Controls.AddNamedRange(Globals.Sheet1.Range["A1", missing],
"NamedRange");
    namedRange1.Change +=new
Microsoft.Office.Interop.Excel.DocEvents_ChangeEventHandler (namedRange1_Change);
}
```

Listing 3-12: Sheet startup event

If you care to examine the `ThisWorkbook` class, you will also find an initialization code section. This section houses the `_Startup` and `_Shutdown` methods. The `_Startup` method initializes the `ThisApplication` and `ThisWorkbook` variables. The `_Shutdown` method releases these variables. Code that needs to manipulate these variables must be placed sometime after the `_Startup` method has been called.

The `Change` event handler is free to implement any sort of programming functionality. Listing 3-13 is an example.

Visual Basic

```
Private Sub namedRange1_Change(ByVal Target As Excel.Range)
    If Not Me.Application.CheckSpelling(CType(Target.Value, String)) Then
        MessageBox.Show("This word is misspelled")
    End If
End Sub
```

C#

```
private void namedRange1_Change(Microsoft.Office.Interop.Excel.Range Target)
{
    if (!this.Application.CheckSpelling((string)Target.Value2, null, true))
        MessageBox.Show("This word is misspelled");
}
```

Listing 3-13: Event handler in action

The event handler checks the spelling on the named range when a change is detected in the cell. For the code to fire, the end user must enter data into the cell and navigate to the next cell. The change event won't fire while the end user is entering data into the cell. To test the example, enter an incorrectly spelled word in the range "A1" and press the tab key. A message box will appear on screen, indicating that the word has not been spelled correctly.

Although the worksheet object has a couple of predefined events, nothing stops you from wiring your own functionality via the events model. As long as you perform the wiring after the startup event handler has fired and the reference that holds the event is declared as a global object, your code should execute successfully. Finally, it's important to also unwire dynamically added events when they are no longer needed. Listing 3-14 shows another example that uses events to perform some calculation on the spreadsheet based on events.

Visual Basic

```
Public Class Sheet1
    Dim evtRange As Microsoft.Office.Tools.Excel.NamedRange = Nothing
    Private Sub evtRange_Change(ByVal Target As Microsoft.Office.Interop
        .Excel.Range)
        Dim currentCell As Excel.Range = Target.Application.ActiveCell
        currentCell.EntireRow.Font.Bold = True
        currentCell.EntireRow.Font.Color = System.Drawing.ColorTranslator
        .ToOle(System.Drawing.Color.Silver)
    End Sub
    Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
        Handles Me.Startup
        evtRange = Globals.Sheet1.Controls.AddNamedRange(Globals.Sheet1
        .Range("A1"), "evtRange")
        AddHandler evtRange.Change, AddressOf Me. evtRange_Change
    End Sub

    Private Sub Sheet1_Shutdown(ByVal sender As Object, ByVal e As
        System.EventArgs) Handles Me.Shutdown
        RemoveHandler evtRange.Change, AddressOf Me. evtRange_Change
    End Sub
End Class
```

C#

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace Difference
{
    public partial class Sheet1
    {
        Microsoft.Office.Tools.Excel.NamedRange evtRange = null;
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
```



```
        evntRange = Globals.Sheet1.Controls.AddNamedRange(Globals
.Sheet1.Range["A1", missing], "evntRange");
        evntRange.Change += new Microsoft.Office.Interop.Excel.DocEvents_
ChangeEventHandler(evntRange_Change);
    }

    void evntRange_Change(Microsoft.Office.Interop.Excel.Range Target)
    {
        Excel.Range currentCell = Target.Application.ActiveCell;
        currentCell.EntireRow.Font.Bold = true;
        currentCell.EntireRow.Font.Color = System.Drawing.ColorTranslator
.ToOle(System.Drawing.Color.Silver);
    }

    private void Sheet1_Shutdown(object sender, System.EventArgs e)
    {
        evntRange.Change -= new Microsoft.Office.Interop.Excel.DocEvents_
ChangeEventHandler(evntRange_Change);
    }
}
```

Listing 3-14: Change events

Listing 3-14 shows code that performs some basic formatting based on the change event. Notice that the code correctly declares the variables at the application level. This is correct form. Events that are declared inside methods will stop functioning when garbage collection occurs. For long running applications, this can be problematic since the application functionality will simply stop working.

Once a user has entered information in the cell and tabs over to another cell, the change event fires. The code in the event handler then applies a bold format and a silver font color to the text in the entire row. Excel can support over 56 colors in a workbook, so there are plenty of choices for formatting.

One important thing to note is that the actual cell that was changed is available by using the `Target` parameter of the event handler. You may use this parameter to extract the contents of the changed cell or to format it appropriately.

Working with Workbook Controls

Legacy applications that provide Office functionality have one common shortfall. These applications cannot embed any type of window control on the spreadsheet surface. VSTO now allows Windows controls to be embedded on the spreadsheet surface. Although this is a welcome addition, embedding Windows controls in the spreadsheet may pose some challenges that aren't easy to overcome. The next few sections will show how to work with controls on the Excel spreadsheet surface. A better alternative to embedding Windows controls on the spreadsheet surface will be discussed later in the next few sections.

The Range Control

The Excel spreadsheet can support both Windows COM controls and .NET controls on its surface. Simply drag a control onto the Excel spreadsheet and wire it up to an event. The process is designed to be exactly the same as for Windows forms development.

The Range control can be attached to a specific range in the spreadsheet. The properties of the control can then be used to customize the range at design time and at runtime. Here is an example that applies some formatting to a range, the same as in Listing 2-15 in the previous chapter. Listing 2-15 has been reproduced here, as Listing 3-15, for convenience.

Visual Basic

```
Dim rng As Excel.Range = Globals.Sheet1.Range("a1", "g12")
rng.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects2,
    True, False, True, False, True, True)
```

C#

```
Excel.Range rng = Globals.Sheet1.Range["a1", "g12"] as Excel.Range;
rng.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects2,
    true, false, true, false, true, true);
```

Listing 3-15: Autoformatting a range

Drop a Range control on the spreadsheet. Right-click on the spreadsheet and choose Manage Named Range. In the dialog box that pops up, you have the option to name the range and to set the target of the range. Set these properties to be identical to those in Figure 3-6.

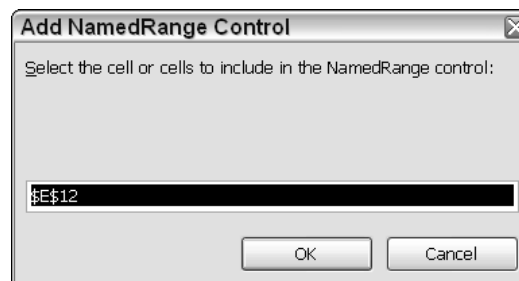


Figure 3-6

VSTO ships with a few default controls that are especially built to interact with the spreadsheet. The Range object is part of these select few. After a default installation, the Range control will automatically be available in the Excel controls node of the toolbox property page.

That's as far as the named Range control will take you. To format the range as in Figure 3-6, you will need to replicate the line of code in Listing 3-16. Here is an example:

Visual Basic

```
Globals.Sheet1.Rng.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects2, True, False, True, False, True, True)
```

C#

```
Globals.Sheet1.Rng.AutoFormat(Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects2, true, false, true, false, true, true);
```

Listing 3-16: Named range control

Although you cannot see a declaration for the control, it is available to calling code using the name of the control. In Listing 3-16, the name of the control is set to `Rng`, and this variable is automatically available in the `ThisWorkbook` file.

Definitions for host controls are available in the `designer.cs` file of the project. This file does not show up because it is hidden. But it is available for viewing through Windows Explorer.

Admittedly, the `NamedRange` control is extremely limited in design mode. Future releases may cure this malady. On the other hand, the benefit of using a named range is that the code can be made as generic as possible. If there is ever a need to readjust the target cells, the change can be accomplished without having to modify the existing code.

The List Control

The list object control is new for VSTO. The list object control allows the developer to enter data into a predetermined area on the spreadsheet and work with that control in the code behind using the list object's name.

To implement a list control, first create a new VSTO project. Name the project `ListTest`. Open the designer and drag a list object control onto the spreadsheet. From this point, the list object behaves like an ordinary Windows control. For instance, enter data into the list object column. In the property pages for the list object, change the default name of `list1` to `myListObject`. In the code behind, you are now able to access a variable named `myListObject` that provides access to the data you entered in the column. Listing 3-17 has some code to manipulate the list object.

Visual Basic

```
Dim lister As Excel.Range  
lister = Me.Range("F4", "F7")  
myListObject.Resize(lister)  
'add code to manipulate the list object here
```

C#

```
Excel.Range lister = this.Range["F4", "F7"];  
myListObject.Resize(lister);  
//add code to manipulate the list object here
```

Listing 3-17: List manipulation

In Listing 3-17, a reference to a range "F4,F7" is first obtained and stored in variable `list`. Then, the list object `myListObject` is resized with the new range. By default, a list object is tied to a single cell when it is first placed on the Excel designer form. That single cell is the implicit anchor for the list object. When the range is resized, it must be resized from the anchored cell. For Listing 3-17 to run correctly, the list object must be renamed from `list1` to `myListObject` and placed on cell F4. A runtime exception results if these two conditions are not met.

List objects can be bound directly to a data source such as a database table or to XML files. Then, the data can be manipulated in design mode or in the code behind. In Figure 3-7, a list control has been bound directly to an XML file. The data contained in the XML file is displayed in the spreadsheet. The XML file is simple and contains two nodes, `count` and `item`, each containing three sets of values. The list object is able to read the XML file and format it correctly for display in the spreadsheet.

When VSTO detects that a list object is present, a floater list object menu bar immediately appears as shown in Figure 3-8. Using either the property pages for the list object in Visual Studio or the list object floater bar, point the data source property to an XML file on disk.

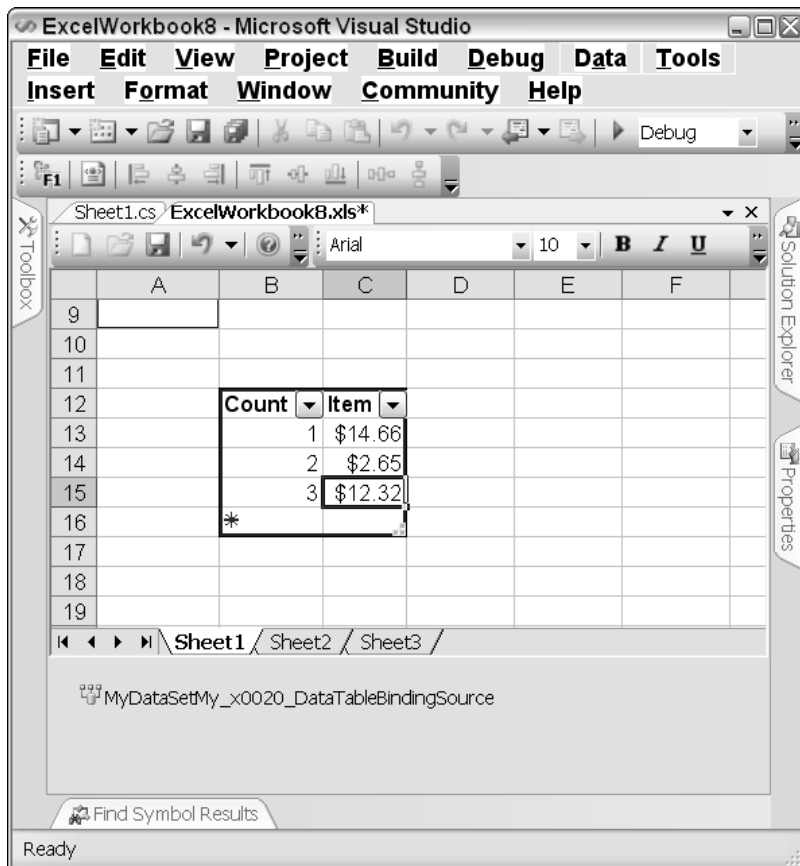


Figure 3-7

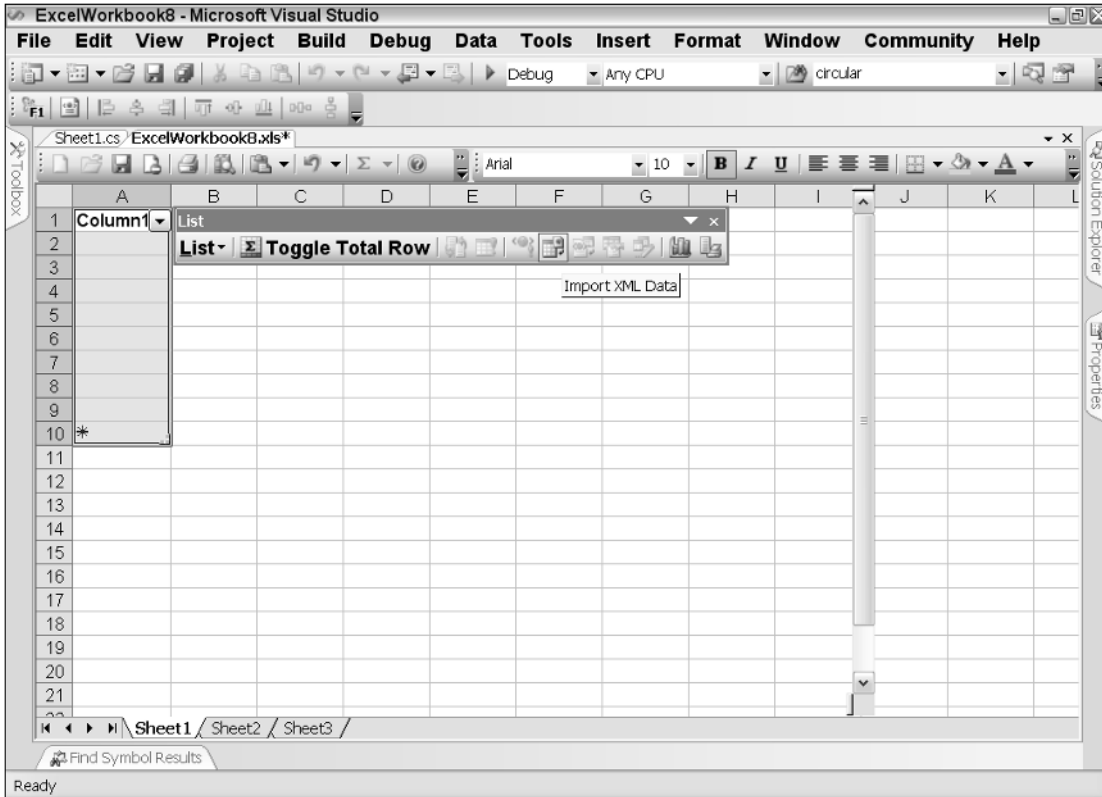


Figure 3-8

Click the second button on the floater toolbar to total the range. Finally, print the contents of the list object by using the print button on the list object floater bar. Notice that these steps were all performed at design time.

Notice the component tray at the bottom in the Visual Studio IDE. The data source component appears when the list object is bound to a data source. Double-clicking the data source opens up the code-behind file to the `currentchange` event. You can place code to react accordingly when the contents of the bound data source changes.

The Actions Pane

The Actions pane was introduced in the first version of VSTO. It was aimed at allowing developers more flexibility to work with controls on the document surface. While VSTO certainly does allow controls to be dropped on the spreadsheet surface, it is a technique that should be avoided in Excel development because it is clumsy and unprofessional. Instead, controls should be moved to the Excel actions pane. Figure 3-9 shows the actions pane.

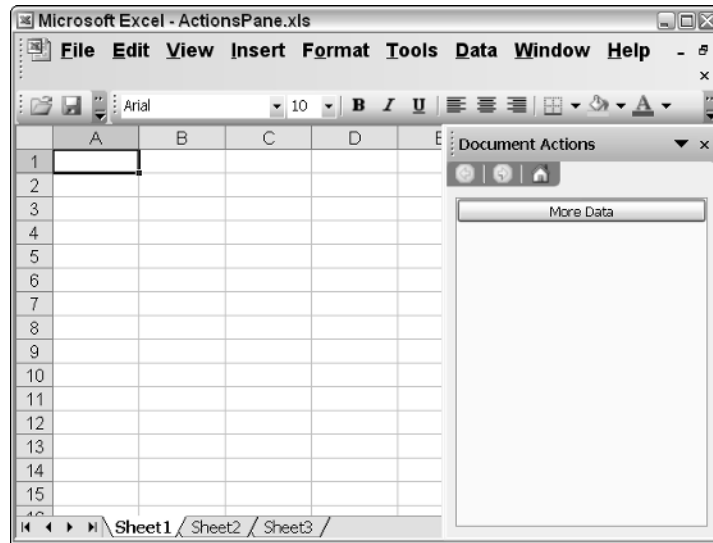


Figure 3-9

The actions pane contains a world of functionality. All sorts of controls can be added to the actions pane surface. The controls can be wired to respond to events to implement functionality either independently of the spreadsheet or integrated with it. The actions pane can also be styled and formatted. The controls that appear on its surface can be positioned and oriented in a variety of ways. Listing 3-18 contains an example of some rudimentary functionality.

Visual Basic

```
Public myBut As Button = New Button()
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
myBut.Text = "Save Data"
AddHandler myBut.Click, AddressOf Me.myBut_Click
Globals.ThisWorkbook.ActionsPane.Controls.Add(myBut)
End Sub
Private Sub myBut_Click(ByVal sender As Object, ByVal e As EventArgs)
Dim saveFileDialog1 As System.Windows.Forms.SaveFileDialog
saveFileDialog1 = New SaveFileDialog()
saveFileDialog1.ShowDialog()
End Sub
```

C#

```
public Button myBut = new Button();
private void Sheet1_Startup(object sender, System.EventArgs e)
{
myBut.Text = "Save Data";
myBut.Click += new EventHandler(myBut_Click);
Globals.ThisWorkbook.ActionsPane.Controls.Add(myBut);
}
```

```
void myBut_Click(object sender, EventArgs e)
{
    System.Windows.Forms.SaveFileDialog saveFileDialog1 = new SaveFileDialog();
    saveFileDialog1.ShowDialog();
}
```

Listing 3-18: Actions pane functionality

If you treat the actions pane as an ordinary Windows forms control, you should be able to program the pane without any difficulty. Listing 3-18 first declares an instance of a button control and then adds this control to the actions pane. Once this button is clicked, it raises a File dialog so that the user can save the data associated with the range. The actual save routine has not been implemented here. An example of the save routine may be found in Chapter 2.

Notice that the actions pane is referenced through the `Globals` object. A reference to the actions pane allows you to access the rich feature set of the actions pane form. For instance, you are able to customize the appearance of the controls on the surface among other things.

There are a couple of limitations with actions panes. First, actions panes cannot support multiple documents. An actions pane is dedicated to a single document. Second, it is not possible to reposition the actions pane because it is embedded in the task pane. One workaround is to programmatically reposition the task pane by using the position property of the `commandbar` object. Consider Listing 3-19:

Visual Basic

```
Me.CommandBars("Task Pane").Position = _
    Microsoft.Office.Core.MsoBarPosition.msoBarLeft
```

C#

```
this.CommandBars["Task Pane"].Position =
    Microsoft.Office.Core.MsoBarPosition.msoBarLeft;
```

Listing 3-19: Repositioning the task pane

End users generally position panes in convenient locations according to their own preferences in the user interface. Because of this, it is not a good idea to impose your own preferences on the end user. If you have a need to reposition a pane, you should always allow the end user to override this behavior.

Printing Workbook Data

Historically, printing data involved a lot of drama. Office-based applications built to run in a web browser can only be printed through the web browser. The web browser provides very little printing flexibility with the end result that data is either cropped or incorrectly formatted on the print surface. For other applications built using Excel Interop, entire libraries dedicated to printing had to be written. These printing libraries had the complex task of ensuring that the print document remained consistent for a wide variety of printers. Today, VSTO makes this process really simple without the headache. The printing functionality is wrapped in a new API that gets the job done while maintaining the flexibility

for different printing environments. Consider the example in Listing 3-20, which displays the Print Preview dialog.

Visual Basic

```
Dim i As Integer
    For i = 1 To Application.Worksheets.Count - 1 Step i + 1
        Dim wr As Excel.Worksheet = Application.Worksheets(i)
        If Not wr Is Nothing Then
            wr.PrintOut(1, 1, 1, True)
        End If
    Next
```

C#

```
for (int i = 1; i < Application.Worksheets.Count; i++)
{
    Excel.Worksheet wr = Application.Worksheets[i] as Excel.Worksheet;
    if (wr != null)
        wr.PrintOut(1, 1, 1, true, missing, missing, missing, missing);
}
```

Listing 3-20: Printing workbook data

From the options chosen, and assuming that the end user decides to print the document, the code starts printing from page 1 and prints one copy of the document. Listing 3-21 shows an alternative way to display the print dialog.

Visual Basic

```
CType(Me.Sheets(1), Excel.Worksheet).PrintPreview(False)
```

C#

```
((Excel.Worksheet)ThisApplication.Sheets[1]).PrintPreview(false);
```

Listing 3-21: Print preview

When documents are printed, embedded objects are printed by default. There is no way to print a document without printing the embedded object, since it is part of the document. However, if you just want to print the embedded document and not the Excel spreadsheet, consider Listing 3-22.

Visual Basic

```
Microsoft.Office.Tools.Excel.NamedRange printedRange =
    Me.Controls.AddNamedRange(Me.Range("A1", "B2"), "printedRange")
    printedRange.Value = "Adding text here for printing purposes"
    printedRange.PrintPreview(True)
```

C#

```
Microsoft.Office.Tools.Excel.NamedRange printedRange =
    this.Controls.AddNamedRange(this.Range["A1", "B2"], "printedRange");
    printedRange.Value2 = "Adding text here for printing purposes";
    printedRange.PrintPreview(true);
```

Listing 3-22: Print preview example

Before running the code, you must first create a `namedRange` control by dragging it from the toolbox unto the design surface in VSTO. Name that new `Range` control `printedRange`. Once you have completed this step, you may enter the code in Listing 3-22.

In case you were wondering, the optional parameter, set to `true` in Listing 3-22, allows the user to make changes to the document while in print preview mode. Notice how the code adds some default text to the range. The reason for this is that printing is optimized to only print dirty cells, or cells that contain data. Without this optimization, an excel worksheet would simply print all the cells available on the spreadsheet. That type of functionality would certainly be a huge waste of resources, since Excel supports 65,536 rows and 256 columns.

Excel Toolbar Customization

Historically, the Office toolbar was a COM object that was wired into the Office user interface widget. Even that has changed. The toolbar now presents as a `CommandBars` object. The `CommandBars` object is actually a collection of `CommandBarControls` consisting of buttons, combo boxes, bar controls, and pop-up components. These are the basic objects that may be used with the Office toolbar. Although Microsoft Excel imposes no upper limit on the number of toolbars that may be created, you should exercise restraint when creating toolbars, since they tend to overwhelm the end user.

Manipulating the toolbar is fairly easy. Listing 3-23 adds a button to the main menu bar. The new button allows the user to customize the appearance of a range on the spreadsheet through the styles object.

Visual Basic

```
Private Sub AddMyToolBar()  
    Dim NewCommandBar As Office.CommandBar  
    NewCommandBar = Application.CommandBars.Add("myButton", Office  
.MsoBarPosition.msoBarTop, False, False)  
    Dim NewCBarButton As Office.CommandBarButton = CType(NewCommandBar  
.Controls.Add(Office.MsoControlType.msoControlButton, Office.CommandBarButton)  
NewCBarButton.Caption = "my new button"  
NewCBarButton.FaceId = 563  
NewCommandBar.Visible = True  
AddHandler NewCBarButton.Click, AddressOf Me. ButtonClick  
  
End Sub
```

C#

```
private void AddMyToolBar()  
{  
    Office.CommandBar NewCommandBar = Application.CommandBars  
.Add("myButton", Office.MsoBarPosition.msoBarTop, false, false);  
    Office.CommandBarButton NewCBarButton = (Office  
.CommandBarButton)NewCommandBar.Controls.Add(Office.MsoControlType.msoControlButton,  
missing, missing, missing, false);  
    NewCBarButton.Caption = "my new button";  
    NewCBarButton.FaceId = 563;  
}
```

```

        NewCommandBar.Visible = true;
        NewCBarButton.Click += new
Microsoft.Office.Core._CommandBarButtonEvents_ClickEventHandler(ButtonClick);
    }

```

Listing 3-23: Toolbar customization

For events to fire consistently, the variables that hold references to the events must be declared global in scope. Otherwise, the event will fire once and stop working.

The code in Listing 3-23 first creates a `commandbar` object and then adds a specific type of button to it along with the appropriate caption. The code also sets up an event handler to handle the newly created button click event. The click event handler is shown in Listing 3-24.

Visual Basic

```

' Handles the event when a button on the new toolbar is clicked.
private Sub ButtonClick(ByVar ctrl as Office.CommandBarButton, ByRef cancel as
Boolean)
    MessageBox.Show("Hello, world!")
End Sub

```

C#

```

// Handles the event when a button on the new toolbar is clicked.
private void ButtonClick(Office.CommandBarButton ctrl, ref bool cancel)
{
    MessageBox.Show("Hello, world!");
}

```

Listing 3-24: Toolbar event handler

The `AddMyToolbar` method deserves some special attention. Consider the line of code in Listing 3-28 that adds a control of type `Office.MsoControlType.msoControlButton`.

This code creates a new command bar button and stores a reference to the newly created object in the `NewCBarButton` variable. If the `Office.MsoControlType` type is set to type `msoControlButton`, as in the example, the VSTO engine enables caption display on the button face only. Images will not be displayed. In fact, if images are present, they are overridden. If you require an image or perhaps a mixture of images and text on the button surface, change the type to `msoButtonIcon` or `msoButtonIconAndCaptionBelow`. The former causes images to display on the button surface. The latter causes icons to be displayed with the caption text neatly formatted below the image.

The image used to fill the button surface is sourced from an image control. There are about 2000 icons in the image control. By assigning an appropriate index to the `faceId` property, an image is made available to the button. In Listing 3-24, the image assignment is given by:

```

NewCBarButton.FaceId = 563

```

Two thousand image icons are more than sufficient to satisfy most end-user requirements. However, there is no way to tell what images are available. One good approach is to write a loop to iterate the possible choices in the control. Simply pick the one that is most appropriate and make a note of the

index for use later. Note that valid IDs are positive integers including zero. If no icon exists for the id, a blank icon is automatically displayed and no exception is thrown. Negative values assigned to the `faceId` result in a runtime exception. If you would like to load your own custom image, Chapter 4 discusses one approach and presents some sample code to do so.

In some cases, the code presented previously may fail to work correctly. This is especially true if there are several events wired inside the application code. The end result is that the message box may not be displayed at all. Interestingly, the event wiring is dependent on a unique identifier being assigned to the `Tag` property for event hook up to be successful. VSTO uses these unique tags to track the event handlers and events in the application. Consider Listing 3-25, which enables the message box to be displayed.

Visual Basic

```
aButton.Tag = DateTime.Now 'new tag added
aButton.FaceId = 563
AddHandler aButton.Click, AddressOf Me. aButton_Click
```

C#

```
aButton.Tag = DateTime.Now.ToString(); //new tag added
aButton.FaceId = 563;
aButton.Click += new Office._CommandBarButtonEvents_ClickEventHandler
(Button_Click);
```

Listing 3-25: A better event handler implementation

There is another problem lurking in the code. If you run the code twice in succession, it will throw an `ArgumentException` with an error message that is certain to cause some hair-pulling anxiety. The message reads, "Value does not fall within the expected range." As it turns out, the toolbar buttons that are created are permanently persisted in Excel if the last parameter in the `Add` method is set to `false`, as shown in Listing 3-24.

The exception is thrown because each toolbar button must have a unique identifier. Running the code a second time creates a duplicate "mybutton" identifier. To be sure, better error messages have been written, but this is not one of them!

Further, setting the final parameter to `false` is dangerous because your application is invasive to the Excel application living on the desktop. The average user cannot easily reverse the change after your application has run and has been terminated. And, the new feature is not expected to work when the desktop version of Excel is used outside of your application. So, in a nutshell, the code adds a button that is perfectly useless outside of the housing application, and it is difficult to remove. For these reasons, the recommended approach is to set the last parameter to `true` so that the buttons are temporary.

To move this code from a theoretical example to something closer to the real world, the event handler section of code that displays the message may be modified to actually format the spreadsheet range using a `style` object. Consider Listing 3-26.

Visual Basic

```
Private Sub ButtonClick(ByVal ctrl As Office.CommandBarButton, ByRef cancel As Boolean)
    Dim style As Excel.Style = Globals.ThisWorkbook.Styles.Add("NewStyle")
        style.Font.Name = "Arial"
        style.Font.Size = 10
```

```

        style.Font.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Black)
        style.Interior.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Beige)
        style.Interior.Pattern = Excel.XlPattern.xlPatternSolid
Dim NamedRange1 As Microsoft.Office.Tools.Excel.NamedRange
    NamedRange1 = Me.Controls.AddNamedRange(Me.Range("A1"), "NamedRange1")
        NamedRange1.Value = "Style Test"
        NamedRange1.Style = "NewStyle"
        NamedRange1.Columns.AutoFit()
End Sub

```

C#

```

void barButton_Click(Microsoft.Office.Core.CommandBarButton Ctrl, ref bool
CancelDefault)
{
    Excel.Style style = Globals.ThisWorkbook.Styles.Add("NewStyle", missing);
    style.Font.Name = "Arial";
    style.Font.Size = 10;
    style.Font.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Black);
    style.Interior.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Beige);
    style.Interior.Pattern = Excel.XlPattern.xlPatternSolid;
    Microsoft.Office.Tools.Excel.NamedRange NamedRange1 =
Globals.Sheet1.Controls.AddNamedRange(Globals.Sheet1.Range["A1", missing],
"NamedRange1");
    NamedRange1.Value2 = "'Style Test";
    NamedRange1.Style = "NewStyle";
    NamedRange1.Columns.AutoFit();
}

```

Listing 3-26: Style customization

Except for the `style` concept, the actual implementation is not new. A `style` object is created and named. The `font` object is customized appropriately and some aesthetic features are applied. Finally, the `style` object is assigned to a named range so that the functionality may be applied to it.

The `style` object offers convenient formatting options that are easy to maintain and modify when code goes to production. And there is no real overhead. Microsoft Excel provides the `style` object for formatting and customization. Excel can contain as many as 4000 styles in a workbook. The type of customization may be applied as a single unit of functionality to the target worksheet.

Excel Menu Customization

Most applications that interact heavily with the end user require some sort of menu manipulation. Menus play an organizational role in software applications by allowing structured access to application functionality. For instance, file manipulation functionality may be organized under a File menu and application configuration may be organized under the Tools menu. Successful frameworks present menus that are easy to modify and flexible to work with. VSTO is no exception. VSTO allows developers to build menus easily.

Chapter 3

Menu customization allows the developer to customize the default menus in Excel. One advantage to customizing the default menu is that very little training is required, since users are already familiar with the menu. Another advantage is that the customized functionality appears as if it is part of the Excel Office application. A third advantage is that the code to generate menus may be used in other parts of VSTO, such as Microsoft Outlook or Microsoft Office, because menu generation belongs to VSTO as a whole and not necessarily to a specific component of the tool suite.

Consider Listing 3-27, which adds a menu to the default toolbar.

Visual Basic

```
' Add the menu.
Dim menubar As Office.CommandBar = CType(Application.CommandBars.ActiveMenuBar,
Office.CommandBar)
Dim cmdBarControl As Office.CommandBarPopup
    cmdBarControl =
CType(menubar.Controls.Add(Office.MsoControlType.msoControlPopup, , ,
menubar.Controls.Count, True), Office.CommandBarPopup)
If Not cmdBarControl Is Nothing Then
    cmdBarControl.Caption = "&Refresh"
    ' Add the menu command.
    Dim menuCommand As Office.CommandBarButton
    menuCommand =
CType(cmdBarControl.Controls.Add(Office.MsoControlType.msoControlButton, , , ,
True), Office.CommandBarButton)
    menuCommand.Caption = "&Calculate"
    menuCommand.Tag = DateTime.Now.ToString()
    menuCommand.FaceId = 265
    AddHandler menuCommand.Click, AddressOf Me.menuCommand_Click
End If
```

C#

```
// Add the menu.
Office.CommandBar menubar =
(Office.CommandBar)Application.CommandBars.ActiveMenuBar;
Office.CommandBarPopup cmdBarControl =
(Office.CommandBarPopup)menubar.Controls.Add(
Office.MsoControlType.msoControlPopup, missing, missing, menubar.Controls.Count,
true);
if (cmdBarControl != null)
{
    cmdBarControl.Caption = "&Refresh";
    // Add the menu command.
    Office.CommandBarButton menuCommand =
(Office.CommandBarButton)cmdBarControl.Controls.Add(
Office.MsoControlType.msoControlButton, missing, missing, missing, true);
    menuCommand.Caption = "&Calculate";
    menuCommand.Tag = DateTime.Now.ToString();
    menuCommand.FaceId = 265;
    menuCommand.Click += new
Microsoft.Office.Core._CommandBarButtonEvents_ClickEventHandler(menuCommand_Click);
}
```

Listing 3-27: Menu customization

First, a reference to the `ActiveMenuBar` is retrieved. The `ActiveMenuBar` retrieves the instance of the menu bar that is active in the application. The code then tests to see if there are any buttons present. If there are buttons present, a button is placed at the last position and titled accordingly.

The type parameter allows you to get fancy with the type of button that you wish to create. However, you should not overdo it. The idea is to present a consistent look and feel to the user so that your new addition appears to be part of the default functionality. Buttons that stand out look unprofessional. Cheesy applications promote a lack of confidence in the software, and users are typically less tolerant of bugs found in the software.

If you require further functionality, simply use the reference to the existing button. For instance, a reference to the variable `newButton` may be used to add an event or a bitmap image, or disable the button. Visual Studio IntelliSense usually provides a slew of properties and methods to help with the customization. You may also consult the help documentation for a full explanation on each property or method.

The handler for the button click event presented in Listing 3-27 appears in Listing 3-28.

Visual Basic

```
' update the calculated range
Private Sub menuCommand_Click(ByVal Ctrl As
Microsoft.Office.Core.CommandBarButton, ByRef CancelDefault As Boolean)
    Dim updateRange As Microsoft.Office.Tools.Excel.NamedRange
    updateRange = Globals.Sheet1.Controls.AddNamedRange(Globals.Sheet1.get_Range("A1"),
"UpdateRange")
    updateRange.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects1,
True, False, True, False, True, True)
    updateRange.Calculate()
End Sub
```

C#

```
// update the calculated range
private void menuCommand_Click(Microsoft.Office.Core.CommandBarButton Ctrl, ref
bool CancelDefault)
{
    Microsoft.Office.Tools.Excel.NamedRange updateRange =
    Globals.Sheet1.Controls.AddNamedRange(Globals.Sheet1.get_Range("A1",
missing), "UpdateRange");
    updateRange.AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormat3DEffects1, true,
false, true, false, true, true);
    updateRange.Calculate();
}
}
```

Listing 3-28: Event handler for menu control

In Listing 3-28, the two lines of code simply obtain the values in the specified range, assign it to a named range object, and fire the `Calculate` method. The range is colored appropriately as a visual confirmation to the user that the calculation has occurred.

The `Calculate` method request is handled on a separate thread so that long calculations do not affect the responsiveness of the spreadsheet. If the calculation were not handled on a separate thread, long calculations would cause the spreadsheet to stop responding to end-user requests. For a review on range functionality, see the range section. The calculation engine is probed more deeply in the Excel calculation engine section.

The code presented in Listing 3-27, adds a button to an existing Excel menu. However, if the desired effect is to add a menu bar from scratch, Listing 3-29 will do nicely.

Visual Basic

```
Dim NewMenuBar as Office.CommandBar = DirectCast
    Me.Application.CommandBars.Add("NewMenu",2, True,True), Office.CommandBar)
If NewMenuBar IsNot Nothing Then
    Dim NewButton as Office.CommandBarControl = DirectCast(
        NewMenuBar.Controls.Add(Office.MsoControlType.msoControlDropdown,
Temporary:= True), Office.CommandBarControl)
        NewButton.BeginGroup = True
        NewButton.Visible = True
        NewButton.Caption = "Tooltip text"
        newMenuBar.Visible= True
    End If
```

C#

```
Office.CommandBar newMenuBar = (Office.CommandBar)
    this.Application.CommandBars.Add("NewMenu",2, true,true);
    if (newMenuBar != null)
    {
        Office.CommandBarControl newButton = (Office.CommandBarControl)
newMenuBar.Controls.Add(Office.MsoControlType.msoControlDropdown,
missing, missing, missing, true);
        newButton.BeginGroup = true;
        newButton.Visible = true;
        newButton.Caption = "Tooltip text";
        newMenuBar.Visible= true;
    }
```

Listing 3-29: Menu bar creation

The result of this code is shown in Figure 3-10.

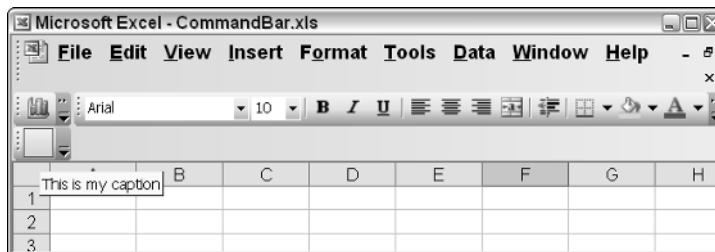


Figure 3-10

The code is slightly different from the code presented in Listing 3-32. Instead of retrieving a reference for the `ActiveMenuBar` object, the code simply adds the appropriate control to the `CommandBars` object that is part of the application object. Note that the new `commandbar` object is created invisible by default. The last line of code is required if you want to see the menu bar.

You may have noticed that the code also fails to check whether or not the item already exists before adding it. This can happen if the application is run twice in succession and you choose to create permanent menus. In this case, the code may fail with an exception if the menu bar exists, because duplicates are not allowed. It's easy to test for this condition and the implementation is left as an exercise to the reader.

VSTO and Web services

Strictly speaking, VSTO contains no internal support for web services. However, web services are supported through the .NET Framework and Visual Studio .NET. The infrastructure to harness web services in Visual Studio .NET is easy to use. In fact, once the data is retrieved from a call to a web service, the code to load data into the spreadsheet is fairly straightforward and has been presented several times already.

The lack of internal support for web services means that code needs to be written to extract the data from the web service call and parse the contents into the spreadsheet. It's not a lot of code, but it still requires some effort.

VSTO provides the `QueryTable` object that may be used in place of a web service call. Query tables are much more powerful and efficient than web services and expose more functionality.

The next example shows how to consume a web service in a VSTO-based application through Visual Studio .NET. You will need to create a VSTO project and name it `WebData`. Once you have the project created, open the property pages by pressing `Ctrl+F4` key combination. From the project hierarchy tree, right-click on the reference node and select `Add Web Reference`. Figure 3-11 shows the property pages with the web reference option selected.

The `Add Web Reference` dialog should appear on screen, as shown in Figure 3-12. Type the URL to the web service. This example uses the Amazon web service `http://soap.amazon.com/schemas2/AmazonWebServices.wsdl`. If a `wsdl` document is found at this URL, the web reference name in the right pane is enabled. You may choose to rename the web reference at this time. The example renames `com.amazon.soap` to `Amazon`. To complete the process, click `Add Reference`.

From this point, the namespace `Amazon` denotes a class that represents a web service. When the service is invoked, data is retrieved from the remote Amazon server. That data is now available to your calling code for use. Listing 3-30 provides a quick example.

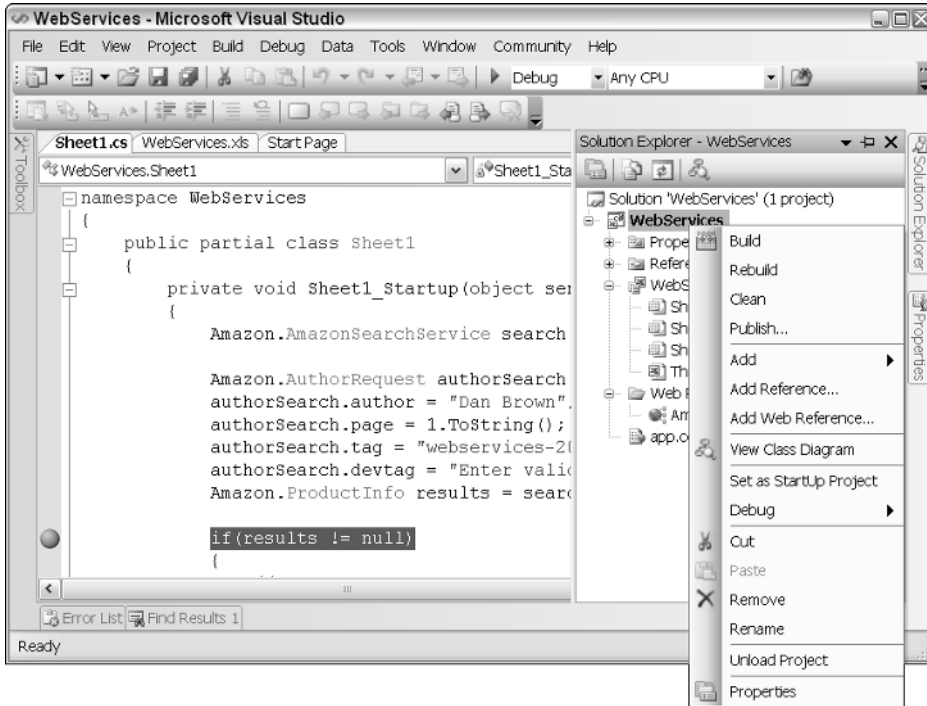


Figure 3-11

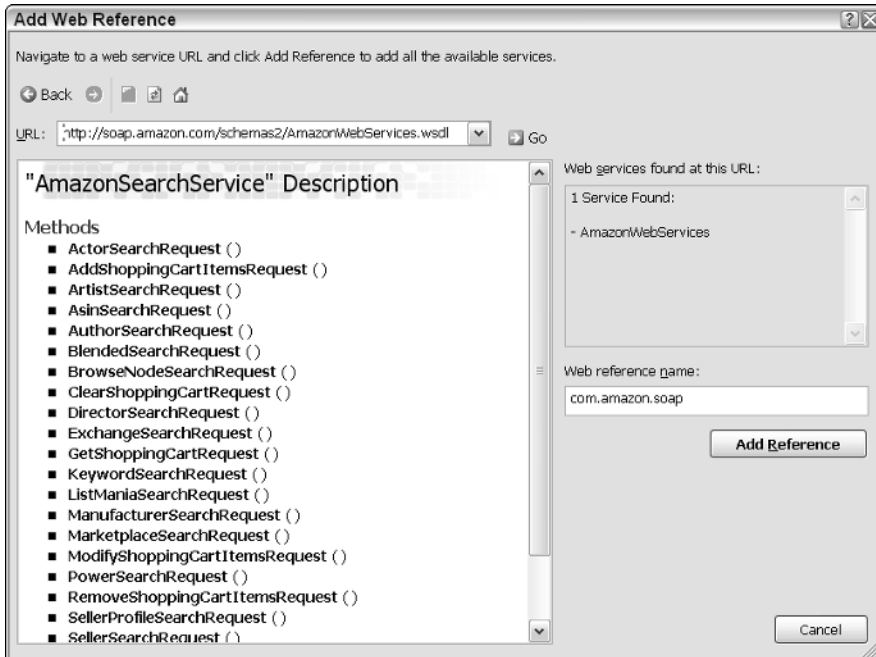


Figure 3-12

Visual Basic

```

Dim search As Amazon.AmazonSearchService = New
WebServices.Amazon.AmazonSearchService()

    Dim authorSearch As Amazon.AuthorRequest
    authorSearch = New Amazon.AuthorRequest()
    authorSearch.author = "Dan Brown"
    authorSearch.page = 1
    authorSearch.tag = "webservices-20"
    authorSearch.devtag = "Enter valid token here"
    Dim results As Amazon.ProductInfo
    results = search.AuthorSearchRequest(authorSearch)

    If Not results Is Nothing Then
        'use results here
    End If

```

C#

```

Amazon.AmazonSearchService search = new WebServices.Amazon.AmazonSearchService();

    Amazon.AuthorRequest authorSearch = new Amazon.AuthorRequest();
    authorSearch.author = "Dan Brown";
    authorSearch.page = 1.ToString();
    authorSearch.tag = "webservices-20";
    authorSearch.devtag = "Enter valid token here";
    Amazon.ProductInfo results = search.AuthorSearchRequest(authorSearch);

    if(results != null)
    {
        //use results here
    }

```

Listing 3-30: Web services in .NET

Stepping through the code briefly, you will notice that a search object is created. Next, an author object is created, since we intend to query author information. We set the author query to "Dan Brown", a popular author. The next line of code indicates the number of pages of data we would like returned to us. We then set the devtag property with an Amazon token. The actual request is initiated when the ProductInfo object is created.

You should note that Amazon provides tokens to developers for testing purposes. In order to run the code, you must register for the service using the following URL <http://associates.amazon.com/gp/associates/network/reports/main.html>. Please be sure to read the licensing terms of use before registering. Once your registration is complete, you should receive a token via email. For more help accessing Amazon web services, you can download and install the Amazon toolkit from the Amazon website.

You may be disappointed to discover that the approach in Listing 3-35 does not involve VSTO at all. It is purely .NET and Visual Studio working the magic. However, the data that is returned from the web service may be harvested for use in a VSTO application.

Chapter 3

In the real world, application functionality isn't always built so easily. In fact, web services are relatively new and companies have only just begun exposing business services through that kind of interface. In a number of cases, corporations expose business services through the web using a combination of `querystring`, `Get` and `Post` parameters. For instance, a book vendor may expose book information through a publicly available URI accepting `querystring` parameters.

Instead of exploring the nuts and bolts through theoretical explanation, we proceed with an example that shows how to retrieve data from a web resource and load the data into an Excel spreadsheet. The idea behind the test application is simple. We write an application to fetch book information from one of the major online book vendors.

The major book vendor will be Alibris. Alibris provides book retrieval information based on the ISBN number through its web interface. The application simply provides a list of books that are available for sale from the Alibris vendor through their web interface. As of this writing, Alibris does not provide a web service interface for book retrieval; however, this information is available by querying a public Universal Resource Identifier (URI) with the required parameters.

From Visual Studio, create your project so that it resembles the spreadsheet shown in Figure 3-13.

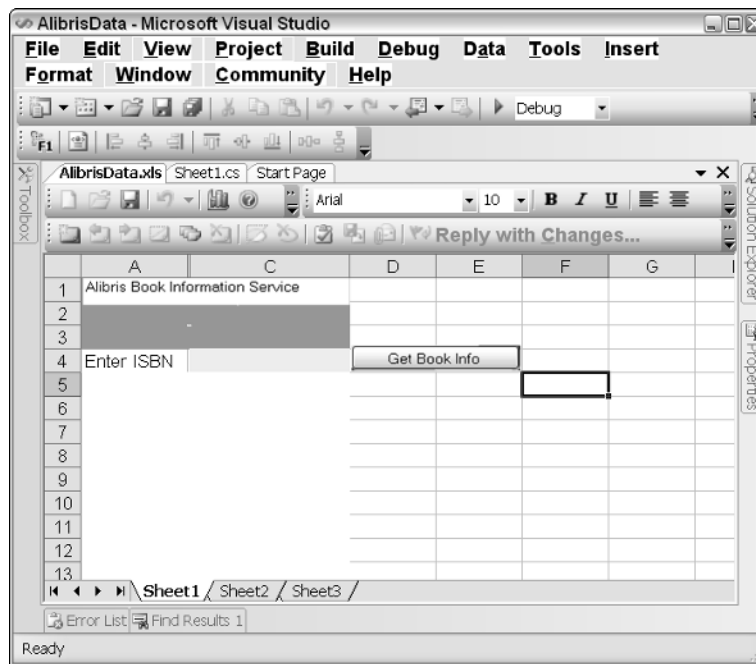


Figure 3-13

Name your project `AlibrisData`. In the `Startup` method, create a reference to the URL. The implementation is shown in Listing 3-31. Once the `Startup` event fires, the code will attempt to connect to the service. If it is successful, data will be returned and stored.

Visual Basic

```

Public Shared Function GetBookInfoFromIsbn(ByVal isbn As String) As DataSet
If isbn = Nothing OrElse isbn.Trim() = String.Empty OrElse isbn.Length <> 10 Then
    Throw New ArgumentNullException("Alibris", "Parameter cannot be Nothing")
End If
'UsedBooksInfo ubInfo = new UsedBooksInfo();
Dim str As System.Text.StringBuilder = New
System.Text.StringBuilder("http://direct.alibris.com/cgi-
bin/texis/chapters/search/search.xml?")
str.Append("qisbn=").Append(isbn)
'This is where we ask the search engine to get us stuff based on our requirements
Dim myWebRequest As System.Net.WebRequest =
System.Net.WebRequest.Create(str.ToString())
' Set the 'Timeout' property in Milliseconds.
myWebRequest.Timeout = 10000
' This request will throw a WebException if it reaches the timeout limit before it
is able to fetch the resource.
Dim ds As DataSet = New DataSet("Alibris")
Try
    Dim myWebResponse As System.Net.WebResponse = myWebRequest.GetResponse()
    Dim reader as System.IO.StreamReader
    reader = New
    System.IO.StreamReader(myWebResponse.GetResponseStream())
    Dim read As System.Xml.XmlTextReader
    read = New System.Xml.XmlTextReader(reader)
    ds.ReadXml(read)
Catch ex As System.Net.WebException
    'inform the user that there is a problem
    MessageBox.Show("Service timed out.")
End Try
Return ds
End Function

```

C#

```

public static DataSet GetBookInfoFromIsbn(string isbn)
{
if (isbn == null || isbn.Trim() == string.Empty || isbn.Length != 10)
    throw new ArgumentNullException("Alibris", "Parameter cannot be null");
//UsedBooksInfo ubInfo = new UsedBooksInfo();
System.Text.StringBuilder str = new
System.Text.StringBuilder("http://direct.alibris.com/cgi-
bin/texis/chapters/search/search.xml?");
str.Append("qisbn=").Append(isbn);
//This is where we ask the search engine to get us stuff based on our requirements
System.Net.WebRequest myWebRequest = System.Net.WebRequest.Create(str.ToString());
// Set the 'Timeout' property in Milliseconds.
myWebRequest.Timeout = 10000;
// This request will throw a WebException if it reaches the timeout limit before it
is able to fetch the resource.
DataSet ds = new DataSet("Alibris");
try
{
    System.Net.WebResponse myWebResponse = myWebRequest.GetResponse();

```

```
System.IO.StreamReader reader = new
System.IO.StreamReader(myWebResponse.GetResponseStream());
System.Xml.XmlTextReader read = new System.Xml.XmlTextReader(reader);
ds.ReadXml(read);
}
catch (System.Net.WebException ex)
{
    //inform the user that there is a problem
    MessageBox.Show("Service timed out.");
}
return ds;
}
```

Listing 3-31: Web query implementation

The first line of code validates the input. An exception is thrown if the validation fails. While you may think that this approach is draconian in nature, it really is the best possible approach. The code should not proceed to create an expensive remote request with faulty arguments. And, the method should not simply return an error code either. In fact, the exception represents a violation of the assumption that an ISBN number is a valid 10-digit number. When assumptions are violated, exceptions must be thrown. To proceed otherwise is an exercise in creating ill-behaved software. As an important side note, the ISBN length should not be hard-coded to a value of 10 either. The ISBN length is scheduled to be changed to an 11-digit number in 2007, at which point, the code may no longer work as intended.

Next, the `url` parameter is parsed into a `StringBuilder` object. A `StringBuilder` object is used for efficiency because it avoids the temporary objects that are associated with .NET string manipulation. It's important to note that the reason the code is wrapped in an exception-handling block is because the code sets an explicit timeout on the request. If the request takes longer than the timeout value, the `WebRequest` object automatically throws an exception.

Again, the assumption made by the `WebRequest` object is simple. A request within a reasonable time period must have a valid response, otherwise the assumption has been violated and an exception must be thrown. This is good program design enforced internally by the .NET Framework, and you should have the discipline to adopt this approach.

You should notice how the code uses an in-memory stream to read the contents into the dataset instead of writing a file to disk first. Once the data is in the dataset, the regular parsing of the dataset into the spreadsheet can occur. You must write the code to manually bind to the spreadsheet cells. If you care to, you may also use a `listobject` or a `Range` control to accept the data. Once the data is in the control, it will appear in the spreadsheet. Listing 3-32 shows the code.

Visual Basic

```
Dim dt As DataRow
For Each dt In ds.Tables(0).Rows
    Dim rng as Excel.Range
    rng = Me.Application.Range("A1")
    rng.Value = dt.ItemArray
Next
```

C#

```

foreach (DataRow dt in ds.Tables[0].Rows)
{
    rng = this.Application.get_Range("A1", missing);
    rng.Value2 = dt.ItemArray;
}

```

Listing 3-32: Dataset assignment to Range object

Figure 3-14 shows the application in action.

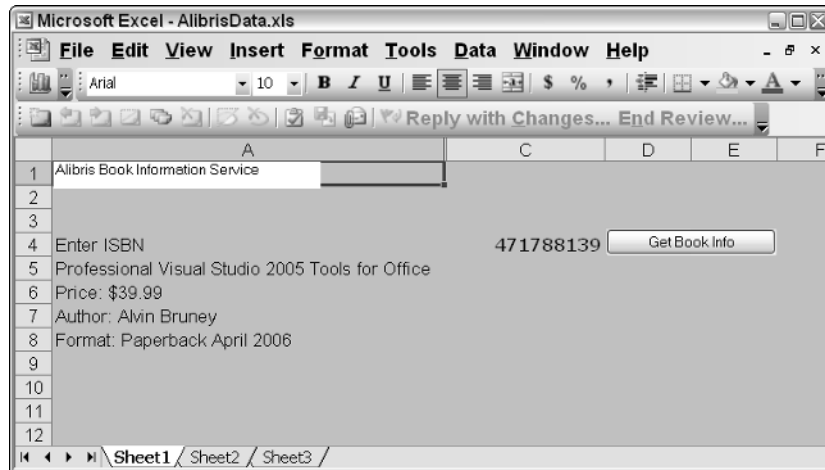


Figure 3-14

The code provides basic functionality, but it is easy to see that a lot can be added. In particular, the Alibris service accepts a wide variety of parameters that provide more flexibility to the calling application. Additionally, you can customize the user interface to add toolbars to introduce new functionality or simply format the display appropriately using the range formatting techniques discussed in Chapter 2. The application uses ISBNs because ISBN numbers are guaranteed to be unique. But you might consider adding author name and title instead of ISBN numbers.

Another more elegant enhancement would be to request the same information from a different vendor if there is an exception or timeout. This would increase the probability of getting results for the end user. You can test this method by entering an invalid ISBN number to trigger the code that queries for data from a different vendor.

Excel Server Automation

VSTO has been redesigned to work on the server without requiring a running instance of Excel on the server. Server-side automation with Excel was not without pitfalls. In fact, Microsoft specifically recommends against server automation for various reasons. However, customers insisted on processing documents on the server using Microsoft Office components because there was a compelling need to do so. VSTO has been designed to provide a clean solution to this nagging problem.

Chapter 3

The marketing hype baked into VSTO claims that data contained inside VSTO-based applications can be manipulated without the need to start an instance of Microsoft Office on the server. It's a bold claim to make especially when history demonstrates rather clearly that software applications that automate Microsoft Office on the server do not scale well.

So, let's put this claim to the test. The idea is to develop an application that houses some data on the server. We then create another application that can access and modify the data. Finally, we test the claim by writing a third piece that simply monitors the server for an instance of Microsoft Office. If an instance is detected, our monitor will bark rather loudly. Listing 3-33 shows the code for the monitor.

Visual Basic

```
Imports System
Imports System.Runtime.InteropServices

Module Module1
    Sub Main()
        Dim automator As Object = Nothing
        While automator Is Nothing
            Try
                automator = Marshal.GetActiveObject("Word.Application")
                Console.WriteLine("Ok, somebody lied to us! Word is running.")
                Console.Read()
                Marshal.ReleaseComObject(automator)
            Catch
                'Microsoft Word is not running
                Console.WriteLine("Watching...")
            End Try
        End While
    End Sub
End Module
```

C#

```
using System;
using System.Runtime.InteropServices;

namespace WatchDog
{
    class Program
    {
        static void Main(string[] args)
        {
            object automator = null;
            while (automator == null)
            {
                try
                {
                    automator = Marshal.GetActiveObject("Word.Application");
                    Console.WriteLine("Ok, somebody lied to us! Word is running.");
                    Console.Read();
                    Marshal.ReleaseComObject(automator);
                }
                catch (System.Runtime.InteropServices.COMException)
                {
                    //Microsoft Word is not running
                    Console.WriteLine("Watching...");
                }
            }
        }
    }
}
```

```

}
}
}
}
}

```

Listing 3-33: Watchdog process code

The watchdog application is conceptually simple. A `while` loop drives the process. During each iteration, the code searches for an instance of Microsoft Word running on the server. If no instance exists, the application prints an appropriate message and continues monitoring. To test the watchdog, compile and run the application. While the application is running, open Microsoft Word and watch this bad boy spring into action. Remember, for a Microsoft Office automation application to scale well, it must necessarily avoid creating an instance of Microsoft Word on the server. Watchdog will monitor the server.

With the trap set, let's see if we can dangle some live bait in the hope of attracting something big. Here is the code to create and house the VSTO-based data. Create a new VSTO-based project and enter the code in Listing 3-34.

Visual Basic

```

Imports System
Imports System.Data
Imports System.Drawing
Imports System.Windows.Forms
Imports Microsoft.VisualStudio.Tools.Applications.Runtime
Imports Word = Microsoft.Office.Interop.Word
Imports Office = Microsoft.Office.Core

Namespace WordDocument1
    Public partial Class ThisDocument
        <Cached> _
        Public data As DataSet
        Private Sub New_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
            data = New DataSet()
            data.ReadXml("sampledata.xml")
        End Sub

        Private Sub New_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs)
        End Sub
    End Class
End Namespace

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;

namespace WordDocument1

```



```
{
    public partial class ThisDocument
    {
        [Cached]
        public DataSet data;
        private void ThisDocument_Startup(object sender, System.EventArgs e)
        {
            data = new DataSet();
            data.ReadXml("sampledata.xml");
        }

        private void ThisDocument_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}
```

Listing 3-34: Application data

This application sets up the necessary plumbing that allows data to be stored inside a VSTO-based application. The first line of code declares a public dataset with a cached attribute. The cached attribute informs VSTO that the data contained in the dataset should be stored in a special way so that it is accessible outside the application. As part of the new architecture, VSTO no longer stores data inside the application document, because this approach does not scale well. Instead, the data is stored inside data islands.

Next, the code loads some arbitrary data, `sampledata.xml`, into the dataset. In the real world, this application would typically read from a data store and apply some sort of business logic to the data. This isn't the real world, so we skip the business logic piece. The sample data in `sampledata.xml` can contain any type of XML-formatted data for testing purposes.

The final piece of code simply tries to access the data in the application. Listing 3-35 shows the code.

Visual Basic

```
Public Sub ManipulateData()
    Dim NewDataSet As DataSet
    'point to the doc file in the debugger directory
    Dim fileName As String =
"C:\WordDocument1\bin\debug\WordDocument1.doc"
    If ServerDocument.IsCacheEnabled(fileName) Then
        Dim servDoc As ServerDocument = Nothing
        Try
            servDoc = New ServerDocument(fileName)
            NewDataSet = New System.Data.DataSet()

            'grab the namespace and the class that contains the cached data

            Dim hostI As CachedDataHostItem =
servDoc.CachedData.HostItems("WordDocument1.ThisDocument")

            Dim dataI As CachedDataItem = hostI.CachedData(0)

            ' load the data
```

```

        If Nothing <> dataI.Xml And Nothing <> dataI.Schema Then
            Dim xmlReader As System.IO.StringReader = New
System.IO.StringReader(dataI.Xml)
            Dim schemaReader As System.IO.StringReader = New
System.IO.StringReader(dataI.Schema)

            NewDataSet.ReadXmlSchema(schemaReader)
            NewDataSet.ReadXml(xmlReader)

If NewDataSet <> Nothing And NewDataSet.Tables <> Nothing And
NewDataSet.Tables.Count > 0 Then
' Modify the data by adding some arbitrary information
    Dim row As DataRow
    For Each row In NewDataSet.Tables(0).Rows
        row(0) = "my New value goes here"
    Next
End If

        dataI.SerializeDataInstance(NewDataSet)
        servDoc.Save()
    End If
Finally
    If Not servDoc Is Nothing Then
        servDoc.Close()
    End If
End Try
End If
End Sub

```

C#

```

Public void ManipulateData()
{
    DataSet newDataSet;
    //point to the doc file in the debugger directory
    String fileName = "C:\WordDocument1\bin\debug\WordDocument1.doc";
    if (ServerDocument.IsCacheEnabled(fileName))
    {
        ServerDocument servDoc = null;
        try
        {
            servDoc = new ServerDocument(fileName);
            newDataSet = new System.Data.DataSet();

            //grab the namespace and the class that contains the cached
data

            CachedDataHostItem hostI =
servDoc.CachedData.HostItems["WordDocument1.ThisDocument"];

            CachedDataItem dataI = hostI.CachedData[0];

            // load the data
            if (null != dataI.Xml && null != dataI.Schema)
            {

```

```
        System.IO.StringReader xmlReader = new
System.IO.StringReader(dataI.Xml);
        System.IO.StringReader schemaReader = new
System.IO.StringReader(dataI.Schema);

        newDataSet.ReadXmlSchema(schemaReader);
        newDataSet.ReadXml(xmlReader);

if(newDataSet != null && newDataSet.Tables != null && newDataSet.Tables.Count > 0)
{
// Modify the data by adding some arbitrary information
    foreach (DataRow row in newDataSet.Tables[0].Rows)
    {
        row[0] = "my new value goes here";
    }
}

        dataI.SerializeDataInstance(newDataSet);
        servDoc.Save();
    }
}
finally
{
    if (servDoc != null)
        servDoc.Close();
}
}
```

Listing 3-35: Application to manipulate data

The code isn't that difficult to follow. First, the code tests to see if the VSTO-based application contains a data cache. The data cache is a new container that is able to access and manipulate the data inside a VSTO-based application. If the document supports VSTO data caching, an instance of the `ServerDocument` class is created. This is a special class that is able to manipulate the actual data inside the application. In the real world, this piece of software could represent another business object that must apply some business logic to the data.

Notice how the code uses a special naming syntax `WordDocument1.ThisDocument` to access the data. This is because the data that is displayed in a VSTO-based Microsoft word document is no longer stored inside the `worddocument1.doc` file. It is now stored in a data island and is accessible through the `ServerDocument` class using this special syntax.

Once access to the data is obtained, the code can simply read the data into a dataset and manipulate it. Finally, the `ServerDocument` class's `Save` method is used to write the changed data in the dataset back into the application store.

So let's fire it up. Run the application watchdog first to begin monitoring for an instance of Microsoft Office. If you don't have a dedicated server, simply run the applications on your desktop as a substitute. Then run the code in Listing 3-29 so that data can be loaded. Finally, fire up the code in Listing 3-30 so

that the data can be manipulated. All we need to do is sit tight and be patient. Sooner or later, Microsoft Office will be invoked to help in the automation. Patience

If you have waited for a few days for an instance of Microsoft Office to show, there's no point in waiting anymore! The marketing hype is true! Microsoft Office is not required to manipulate data contained in VSTO-based applications on the server. This huge step forward necessarily implies that VSTO-based applications can avoid the scalability issues that plagued previous generations of Office software built to run in a server environment.

The cure for this ailment is the new design that separates data from the code that manipulates it. This divorced architecture allows calling code to service data contained inside VSTO-based applications without the need to start an instance of either Microsoft Office or Microsoft Excel. Because using an instance of Office is avoided during the automation, the scalability problems that accompany Office automation disappear entirely.

The code presented here demonstrates that the divorced architecture actually works and is scalable — although we haven't tested the scalability claim. However, there are a couple of drawbacks to this silver bullet. The application must be created using Visual Studio Tools for the Office System 2005, since it needs to support data caching. This is quite a shortcoming because it necessarily means that you must migrate your applications to VSTO solutions first if you intend to take advantage of data caching. Also, data caching is only supported in Microsoft Word and Microsoft Excel. There is no support for data caching in Microsoft InfoPath or Microsoft Outlook.

Another drawback is that the VSTO tools suite is not free. It does cost a fair amount of cash. When compared to the regular Office development based on COM, which is essentially free, the extra cash cost can seem like an unnecessary investment burden. Still, if you have a requirement for a highly scalable piece of Office automation software, VSTO seems like a good alternative.

Excel COM Add-Ins

An add-in is an application that extends the functionality of the Office application. The add-in runs in-process with the Office application, and its functionality is available for use by the Office application. Any number of add-ins may be loaded at any point in time; however, each add-in consumes additional machine resources, and misbehaving add-ins can cause application instability in your custom application.

Add-ins can be developed using either managed or unmanaged code. For unmanaged code, Runtime Callable Wrappers (RCWs) are created to help .NET Interoperate with the unmanaged code. RCWs are created automatically by Visual Studio, or they may be created manually using `tlbimp.exe`, which ships free with the .NET Framework. In either case, the interoperation happens behind the scenes, so it only receives a brief mention here.

One of the major drawbacks of VSTO is that it contains no support for user defined functions (UDF). UDFs are external libraries that add functionality to the Excel spreadsheet. Consider the case where a company has built up a large library of external Excel functions over the years and has been using this library in Excel automation outside of VSTO. It's clearly difficult to make a case for adopting VSTO,

Chapter 3

since there is no internal support for that type of functionality. However, there is a workaround based on an Excel COM add-in.

Consider Listing 3-36, which defines an external function that may be called from Excel.

Visual Basic

```
Imports System.Runtime.InteropServices

Namespace UDFAddIn
    <ClassInterface(ClassInterfaceType.AutoDual)> _
    Public Class Library
        Public Function EchoInput(ByVal v1 As Integer) As String
            Return "You entered " + v1
        End Function

        <ComRegisterFunctionAttribute()> _
        Public Shared Sub RegisterFunction(ByVal type As Type)
            Microsoft.Win32.Registry.ClassesRoot.CreateSubKey("CLSID\\{" +
            type.GUID.ToString().ToUpper() + "\\Programmable")
        End Sub

        <ComUnregisterFunctionAttribute()> _
        Public Shared Sub UnregisterFunction(ByVal type As Type)
            Microsoft.Win32.Registry.ClassesRoot.DeleteSubKey("CLSID\\{" +
            type.GUID.ToString().ToUpper() + "\\Programmable")
        End Sub
    End Class
End Namespace
```

C#

```
using System;
using System.Runtime.InteropServices;

namespace UDFAddIn
{
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class Functions
    {
        public string EchoInput(int v1)
        {
            return "You entered " + v1;
        }

        [ComRegisterFunctionAttribute]
        public static void RegisterFunction(Type type)
        {
            Microsoft.Win32.Registry.ClassesRoot.CreateSubKey(
                "CLSID\\{" + type.GUID.ToString().ToUpper() +
                "\\Programmable");
        }

        [ComUnregisterFunctionAttribute]
        public static void UnregisterFunction(Type type)
        {

```

```

        Microsoft.Win32.Registry.ClassesRoot.DeleteSubKey(
            "CLSID\\" + type.GUID.ToString().ToUpper() +
            "\\Programmable");
    }
}
}

```

Listing 3-36: COM add-in automation

The code presented in Listing 3-41 simply grabs input from the caller and echos it back with a short message. The idea is to show you how to manipulate arguments inside the COM add-in. For instance, `EchoInput` could contain some complex business logic or proprietary formula that may be best deployed on the server instead of in the application directory.

You should note that the add-in is created as a class library project. For help on creating class library projects, consult the MSDN documentation. Since this is a COM add-in, the `System.Runtime.InteropServices` assembly is required. The namespace makes the COM libraries available to the code.

There are two functions that set up the internal plumbing to enable the COM add-in to function correctly. The methods simply register and unregister the Globally Unique Identifier (GUID) representing the COM add-in. The registration process is required, since this call works through the magic of COM Interop. The attributes that precede the methods are necessary as well. Consult the help documentation for further details on .NET attributes.

To use the COM add-in, fire up Microsoft Excel and select Tools ⇨ Add-ins. Select the Automation tab and scroll through the list on the Automation Servers dialog to find and select the `UDFAddin.Library` assembly. You will notice that the assembly depends on `mscorlib.dll`. Click OK to select it. At this point, you may be presented with a dialog indicating that `mscorlib.dll` cannot be found. Select No to ignore the message and proceed to the Excel spreadsheet. Enter `"=EchoInput(12)"` in the cell and press Enter. You should see a message "You entered 12" displayed in the cell.

Obviously, your legacy library will need to be rewritten in Microsoft .NET, and that may be a tougher sales pitch than the actual implementation. Legacy libraries are typically huge beasts that have evolved over the years. Assuming that you can sell upper management on the advantages, the implementation portion should follow the trail blazed in Listing 3-38. You should note that if the formula is entered incorrectly, an error will be displayed. Chapter 2 covered spreadsheet errors in detail.

Finally, VSTO does not expose project templates that allow Excel add-ins to be implemented across the application. VSTO Excel add-ins are document specific. If application requirements dictate that an Excel add-in be application specific, you should use the shared add-in project template in Visual Studio to build an add-in that works across the entire application.

The option is accessed by opening Visual Studio and selecting New ⇨ New Project from the File menu. In the new project dialog, select Other Project Types from the project types pane. Then, select Extensibility. In the templates pane on the left, select Shared Add-in. The Shared Add-In Wizard appears and walks you through the process. The process is very simple with the final result being an add-in that loads and executes when the host process (Microsoft Excel) initializes. From this point, the code approach is no different from an ordinary Excel project. Figure 3-15 shows the add-in option from the Visual Studio project.

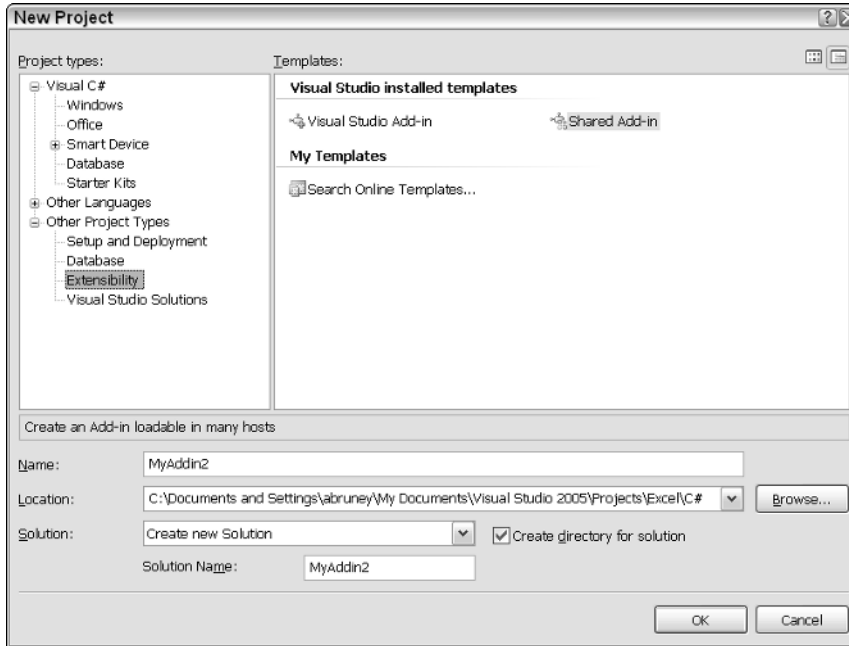


Figure 3-15

The main file in the Office shared add-in, `connect.cs`, contains help documentation before each method that explains the purpose of each method and provides a detailed guide on how to manipulate these methods. Read through the code comments in the `connect.cs` file to determine how to use the methods in a shared add-in. Then, add your custom code as appropriate.

When entering formulas into cells, use lowercasing. Excel parses the contents of the cell and then converts any matching formulas to uppercase. You can quickly determine if a formula was mistyped or not entered correctly by examining the casing of the values in the cell.

Summary

The .NET Framework has been designed from the ground up to address and resolve security concerns. It's no wonder that VSTO uses the .NET Framework for security. This is both a feature and a fault. It is a feature because the .NET Framework addresses security concerns adequately for most programming needs and the implementation is robust and scalable. However, the ugly side deals with the enormous learning curve inherent in .NET security.

The security implementation in VSTO is tiered, occurring at the Workbook level and cascading down to the spreadsheet cell level for the Excel user interface. The approach is not limited to locking the target

object. In fact, hiding and password protecting specific objects is a bona fide security approach as well, and VSTO has internal support for both approaches.

Excel is flexible enough to allow full protection as well as partial protection. For instance, you may require protection for an embedded object, but you may also want to unprotect the spreadsheet. Or, you may want to protect the entire spreadsheet but still afford the user some control, such as the ability to insert rows and columns in a protected spreadsheet. That kind of flexibility caters well to diverse implementation scenarios. And Excel supports these scenarios in an implementation-friendly way.

Office developers will recall the frustration of programming the toolbar due in part to the limited documentation on the topic and the volume of COM Interop material that had to be consumed in order to write a bare-bones application. Today, the interface has been refined to simplify the programming model. Programming buttons into the toolbar is no longer an exercise in patience and discipline. Today, the toolbar has been rewrapped and repackaged for VSTO. It exposes a rich interface that is reasonably well documented and easy to use. The toolbar is also a bona fide part of the Excel spreadsheet, and no cheesy Interop is required to coax functionality out of it.

The chapter also explored the new controls that are available. The list object control is powerful and flexible enough to allow data to be loaded, manipulated, and printed from the design view. That sort of approach reduces the burden on the runtime platform and is more practical to implement, since the developer can see and manipulate the data in real time. The list object is also flexible enough to load data from XML and database sources.

The `Range` object is a programming convenience whose importance is expected to grow as the platform matures. For now, the `Range` object allows developers to use a `Range` control to assign a range and then use generic code to implement the functionality behind the `Range` object. Using generic code means that the target of the range can be adjusted as application requirements change without having to rewrite code.

Finally, we examined data manipulation on the server using the `ServerDocument` class. This new object allows workbook data to be processed without the need to automate Excel. The implementation does involve learning some new concepts but it certainly pays dividends where scalability and performance are concerned.

4

Word Automation

Developers who are eager to leverage Microsoft Office word functionality in enterprise software applications may be encouraged to find that the VSTO Word object model provides most of the functionality of Microsoft Office Word available today, including support for embeddable objects and macro processing. The implementation process to get that type of functionality working correctly has been greatly simplified. In addition, the development is integrated into the Visual Studio integrated development environment (IDE) so that Microsoft Office menus and toolbars are available in Visual Studio .NET. The end result is that development can take place inside the Visual Studio IDE without the need to run external applications or to fire up Microsoft Word or Excel.

This chapter will focus on automating Microsoft Word. First, we examine some key objects responsible for driving the automation process. Then, code will be presented to perform some common tasks associated with a document such as formatting and searching. Later in the chapter, we'll focus on events and customizing the toolbar and menus. Finally, we'll probe some common controls and some new controls such as the background worker control and actions pane. We'll complement the theory with a good dose of code that will show how to program common tasks associated with these controls.

Key Application Objects in Word

VSTO documents are quasi-applications. They behave like applications because the document proper can execute code in .NET assemblies via a few key application objects. However, the document may simply be used as an ordinary text document. In both instances, Microsoft Word is designed to be well behaved. Part of the magic is handled by a redefined object hierarchy that covers the entire range of document functionality. The object hierarchy is listed in Figure 4-1.

In Figure 4-1 observe that the object hierarchy design is clean and relatively simple. A clean architecture reduces the learning curve and stems the confusion that may result from improper implementation. The architecture is also powerful and flexible enough to provide the functionality that is necessary to build robust, enterprise software.

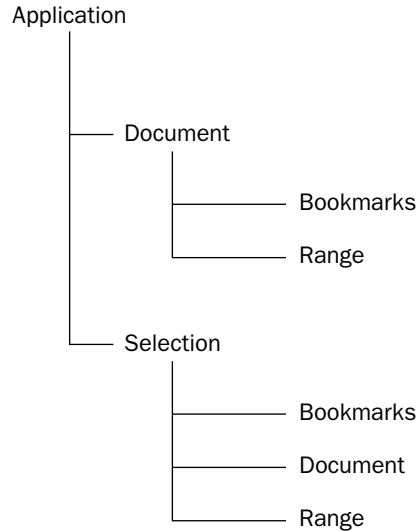


Figure 4-1

Using the application object hierarchy, it is possible to create Microsoft Office Word documents that combine text with shapes. The next few sections examine some key objects that are most likely to be used when building enterprise software in VSTO.

ThisApplication Instance

Perhaps the most important object is the `ThisApplication` instance. This object represents an instance of the executing application. The `ThisApplication` object makes several other key Office Word objects available to executing code. In addition, the `ThisApplication` object exposes methods and properties that may be used to access information about the executing application in general. For instance, the application path may be determined from the `ThisApplication` object. Consider another example in Listing 4-1.

Visual Basic

```

Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    Dim fileName As Object = "c:\setup.log"
    Dim myDoc As Microsoft.Office.Interop.Word.Document =
Me.Application.Documents.Open(fileName)
myDoc.AddToFavorites()
myDoc.Kind = Microsoft.Office.Interop.Word.WdDocumentKind.wdDocumentLetter
myDoc.ShowSpellingErrors = True
End Sub
  
```

C#

```

private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    object fileName = @"c:\setup.log";
    Microsoft.Office.Interop.Word.Document myDoc =
this.Application.Documents.Open(ref fileName,
  
```

```

ref missing, ref missing, ref missing, ref missing, ref missing,
ref missing, ref missing, ref missing, ref missing, ref missing,
ref missing, ref missing, ref missing, ref missing, ref missing);
myDoc.AddToFavorites();
myDoc.Kind = Microsoft.Office.Interop.Word.WdDocumentKind.wdDocumentLetter;
myDoc.ShowSpellingErrors = true;
}

```

Listing 4-1: Application open method

Listing 4-1 shows the application object being used to gain access to the document object's `Open` method. The `Open` method will get its 15 minutes of fame later. For now, just note that the call accepts a filename and some optional parameters that may be used to open a file on disk.

Notice that the example actually opens the Visual Studio .NET installation file log. Although this file extension may not necessarily be associated with an application, Notepad for instance, the file is still able to open. In that case, the contents are interpreted as plain text.

Once the document is opened successfully, it is added to the Favorites folder in Windows Explorer and Internet Explorer so that it may be retrieved quickly. This is a nifty technique that should be used with caution. For instance, end users may not find favor with application links suddenly showing up in the Favorites folder. Etiquette absolutely demands that if such actions must be taken by the application, the code should prompt the end user for confirmation first.

Working with the Range Object

The `Range` object is probably the most popular object in the Word hierarchy of objects. It is responsible for defining a contiguous block of text on the document's surface. The `Range` object is very flexible and even allows for the addressing of several different `Range` objects at a time. Let's consider an example that manipulates some text. We are interested in deleting a word from the second sentence and replacing it with some other word. Here is the sample text: Note that the second sentence starts on line 2 and ends on line 3.

First create a new Office project called `SampleText`. Enter the sample text above into the Word document. Listing 4-2 shows the code to manipulate the sample text.

Visual Basic

```

Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
If Me.Paragraphs.Count > 0 Then
    Dim par As Word.Range = Me.Paragraphs(Paragraphs.Count).Range
    Dim singleSent As Word.Range = par.Sentences(2)
    If singleSent IsNot Nothing AndAlso singleSent.Words IsNot Nothing
AndAlso singleSent.Words.Count > 0 Then
        Dim i As Integer
        For i = 1 To singleSent.Words.Count - 1
            If singleSent.Words(i).Text.ToLower().Trim() = "documents" Then
                Dim removeChar As Word.Range = singleSent.Words(i)
                removeChar.Delete()
                'delete and add customizations
                removeChar.InsertAfter(" document's ")
            End If
        Next i
    End If
End If

```

```
        removeChar.Font.Name = "Arial"
        removeChar.Font.Size = 10.0F
        removeChar.Font.Color = Word.WdColor.wdColorDarkRed
        removeChar.Underline = Word.WdUnderline.wdUnderlineDash
    End If
Next
End If
End IfEnd Sub

C#
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    if (this.Paragraphs.Count > 0)
    {
        Word.Range par = this.Paragraphs[Paragraphs.Count].Range;
        {
            Word.Range singleSent = par.Sentences[2];
            if (singleSent != null && singleSent.Words != null &&
                singleSent.Words.Count > 0)
            {
                for (int i = 1; i < singleSent.Words.Count; i++)
                {
                    if (singleSent.Words[i].Text.ToLower().Trim() ==
"documents")
                    {
                        Word.Range removeChar = singleSent.Words[i];
                        removeChar.Delete(ref missing, ref missing);
                        //delete and add customizations
                        removeChar.InsertAfter(" document's ");
                        removeChar.Font.Name = "Arial";
                        removeChar.Font.Size = 10F;
                        removeChar.Font.Color =
Word.WdColor.wdColorDarkRed;
                        removeChar.Underline =
Word.WdUnderline.wdUnderlineDash;
                    }
                }
            }
        }
    }
}
```

Listing 4-2: Range reference manipulation

Technically, a paragraph is denoted by a line feed character. Striking the Enter key inserts a return line feed in the document that VSTO uses as a paragraph marker. From our sample text, there are four lines of text, each line being followed by a line feed marker. Another easy way to discern paragraphs, according to VSTO's yardstick, is to choose to view the document while revealing all formatting.

Once there is a reference to a paragraph, the code picks out the second sentence of the document and sets a variable to point to the range. It may be of some importance to note that the sentence relates to the range,

whereas the paragraph relates to the document. The range may be made up of several sentences. Or it may be made up of one sentence that spans several lines in the document. However, the paragraph is not determined by the number of lines. Rather, it simply depends on the line feed character. On the other hand, a sentence in Word is defined by the number of punctuation marks in a paragraph. Based on that argument, we rightfully index into the sentence collection and return the first sentence of the `Range` object.

Next, we iterate the words that form part of this single sentence. Our approach examines each word to see if it matches the word “documents.” If we find the word, we delete it. Next, we insert a piece of text and add some cosmetic touch ups to highlight the change in the document. Notice that a lot can be accomplished in terms of formatting. And the Word API spares no expense in making the full formatting API available to you through the `Range` object. However, you must be careful not to overdo it, since these changes can certainly be distracting to the user. This is a very crude spell-check engine, obviously lacking the sophistication and efficiency of the spell-check engine provided by Word. However, both are based on the same principle.

By the way, this is not the only approach to retrieving and formatting text. Another good approach is to execute the `find` method on the document. The search method returns a `Range` reference to the found items, and you can proceed to customize and format the target as you see fit. Listing 4-3 demonstrates an example of this approach based on the data presented previously.

Visual Basic

```
Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    ' Set the second paragraph as the search range.
    Dim rng As Word.Range = Me.Paragraphs(1).Range
    Dim searchRange As Word.Find = rng.Find

    Dim textToFind As Object = "the"
    If searchRange.Execute( textToFind) Then
        MessageBox.Show("Found text: " + Cstr(textToFind), "Finding...")
        rng.Font.Italic = 1
        rng.Font.Color = Word.WdColor.wdColorBlue
    End If
End Sub
```

C#

```
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    // Set the second paragraph as the search range.
    Word.Range rng = this.Paragraphs[1].Range;
    Word.Find searchRange = rng.Find;

    object textToFind = "the";
    if (searchRange.Execute(ref textToFind, ref missing, ref missing, ref
missing, ref missing, ref missing, ref missing, ref missing, ref
missing, ref missing, ref missing, ref missing, ref missing, ref missing))
    {
        MessageBox.Show("Found text: " + textToFind, "Finding...");
        rng.Font.Italic = 1;
        rng.Font.Color = Word.WdColor.wdColorBlue;
    }
}
```

Listing 4-3: Implementing document search functionality

In Listing 4-3, a variable is set to point to the first paragraph. With the reference, the code simply executes the find method on the range. If the find is successful, the range is formatted appropriately. This approach is less crude than Listing 4-2 and a lot closer in implementation to the Word spell-check engine implementation as well.

There may be more approaches that are cleaner and more efficient to implement. However, these two examples provide a good starting point. Also, with the second approach, you may need to write extra code to determine if this piece of text is found in the last line, since there may be identical words scattered throughout the search range. An implementation is left to the reader.

Working with the Bookmark Object

Bookmarks represent objects that mark a specific position in the document. Once a bookmark is set up in the document, it is easy to navigate to that particular bookmark. In addition, it is possible to manipulate the target of the bookmark. Consider an example that first creates a bookmark using the Word Office menu and then manipulates this bookmark in code.

For this example, reuse the piece of text from the previous example. From the Insert menu on the VSTO word document, select Bookmark. The bookmark dialog should now be present on screen. Enter the title myBookMark. Set the options according to Figure 4-2.



Figure 4-2

Once the bookmark is set up, navigate to the `OfficeCodeBehind` file and add the code shown in Listing 4-4.

Visual Basic

```
Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    If ThisApplication.ActiveDocument.Bookmarks <> Nothing And
ThisApplication.ActiveDocument.Bookmarks.Count > 0 Then
```

```

        Dim mark As Word.Bookmarks
        For Each mark In ThisApplication.ActiveDocument.Bookmarks
            If mark.ToString().ToLower().Trim() = "mybookmark" Then
                MessageBox.Show("Found bookmark")
                Exit For
            End If
        Next
    Else
        Dim par as Word.Range = Me.Paragraphs(Paragraphs.Count).Range

        Dim singleSent As Word.Range = par.Sentences(1)
        If singleSent IsNot Nothing AndAlso singleSent.Words IsNot
Nothing AndAlso singleSent.Words.Count > 0 Then

            Dim i As Integer
            For i = 1 To singleSent.Words.Count- 1 Step i + 1
                If singleSent.Words(i).Text.ToLower().Trim() = "the"
Then
                    Dim bookMark As Object = singleSent.Words(i)

                    ThisApplication.ActiveDocument.Bookmarks.Add("myBookMark", bookMark)
                End If
            Next
        End If
    End Sub

```

C#

```

private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    if (this.Bookmarks != null && this.Bookmarks.Count > 0)
    {
        foreach (Word.Bookmarks mark in this.Bookmarks)
        {
            if (mark.ToString().ToLower().Trim() == "mybookmark")
            {
                MessageBox.Show("Found bookmark");
                break;
            }
        }
    }
    else
    {
        Word.Range par = this.Paragraphs[Paragraphs.Count].Range;
        {
            Word.Range singleSent = par.Sentences[1];
            if (singleSent != null && singleSent.Words != null &&
                singleSent.Words.Count > 0)
            {
                for (int i = 1; i < singleSent.Words.Count; i++)
                {
                    if (singleSent.Words[i].Text.ToLower().trim() == "the")
                    {

```



```
        object bookMark = singleSent.Words[i];
        this.Bookmarks.Add("myBookMark", ref bookMark);
    }
}
}
}
}
```

Listing 4-4: Bookmark code manipulation

The approach involves checking the document to see if a bookmark exists in the bookmarks collection. Since bookmarks are document specific in nature, it makes sense to check to see if one exists first. If it does, the code simply displays a message indicating that the bookmark has been found. Otherwise, the code creates the bookmark and inserts it into the document. In the absence of such a check, the newly created bookmark simply overwrites the old.

Aside from this minor inconvenience, there is a bug in the code. Most collections in VSTO are not zero-based collections. If there is only one paragraph in the document, the `Paragraphs.Count` property line of code will return 1. After the subtraction operation is performed, the code will then attempt to access data using an invalid zero index. Be wary about indexing into collections in VSTO, especially when calculations are performed as in Listing 4-4. You should be able to repair the code without any more help.

Working with the Selection Object

The `Selection` object represents the cursor selection in the active document. Listing 4-5 has an example.

Visual Basic

```
Dim start As Object = 0
Dim end As Object = 0
Word.Range par = Me.Paragraphs(Paragraphs.Count).Range
    Par.Select()
If(ThisApplication.Selection <> Nothing)
    ThisApplication.Selection.TypeText = "This is some selected text"
    Par.InsertAfter = "This is the end."
End If
```

C#

```
Object start = 0;
Object end = 0;
Word.Range par = this.Paragraphs[Paragraphs.Count].Range;
    Par.Select();
If(ThisApplication.Selection != null)
{
    ThisApplication.Selection.TypeText = "This is some selected text";
    Par.InsertAfter = "This is the end.";
}
```

Listing 4-5: Selection object example

The obvious difference between the `Range` and the `Selection` object has to do with the fact that a range must be selected for the selection object to be valid. However, more differences exist. Selection objects are more difficult to work with and expose less functionality than `Range` objects. Attempting to manipulate selection objects will result in visible selection changes in the user's document. Often, users will not welcome such changes, even though they will likely not affect the data in the document.

In both cases, the selection and the `Range` objects' lifetimes are affected by scoping rules. If these objects go out of scope, the `Range` and `Selection` objects follow suit. As a precaution, always test the variables pointing to selection and `Range` objects for validity before using them.

Additionally, a range provides a suitable way to work with a contiguous block of text without selecting that piece of text in the interface. By using the `Range` object instead of the selection object, code can safely work with data on the document surface without drawing the user's attention to the operation.

Although, the selection object does provide access to noncontiguous blocks on the document surface, the technical limitation is that the selection object referencing the noncontiguous blocks only points to the last selected range. So technically, addressing noncontiguous blocks with the selection object is simply not feasible in this version of VSTO.

VSTO Table Manipulation

The Microsoft Office suite of products has always had built-in support for table objects, and this has been extended to include VSTO from the very first public version. However, the most recent release of VSTO has added improved support for tables. Conceptually, tables group logical pieces of information into rows and columns. Figure 4-3 shows a Word table.

The programmatic `table` object is functionally equivalent to the Office table shown in Figure 4-3. That is, it supports rows, columns, cells, and various built-in formatting capabilities. Listing 4-6 shows an example that adds a table to the document with some text.

Visual Basic

```
Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    Dim rng As Word.Range = Range()
    Dim tRange As Word.Range = Range()
    Dim table As Word.Table = rng.Tables.Add(rng, 3, 3 )
    table.Rows(1).Cells(1).Range.Text = "Table Heading 1"
    table.Rows(1).Cells(2).Range.Text = "Table Heading 2"
    table.Rows(1).Cells(3).Range.Text = "Table Heading 3"
    Dim style As Object = "Table Grid"
    table.Style = style
End Sub
```

C#

```
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    Word.Range rng = Range(ref missing, ref missing);
    Word.Range tRange = Range(ref missing, ref missing);
    Word.Table table = rng.Tables.Add(rng, 3, 3, ref missing, ref missing);
    table.Rows[1].Cells[1].Range.Text = "Table Heading 1";
}
```

```
table.Rows[1].Cells[2].Range.Text = "Table Heading 2";  
table.Rows[1].Cells[3].Range.Text = "Table Heading 3";  
object style = "Table Grid";  
table.set_Style(ref style);  
}
```

Listing 4-6: Adding tables programmatically via VSTO

Let's walk through the code in order to understand the basics. The Range object provides access to the global table collection. The first parameter determines the location of the table. Once the table is imposed on the document's surface, the cells of the table may be filled with data. The Rows property of the table object returns a reference to each row in the table. Each row must contain at least one cell. For each row object, the cells correspond exactly to the column object, and vice versa. The implementation provides a coordinate type addressing schema that is both convenient and intuitive.

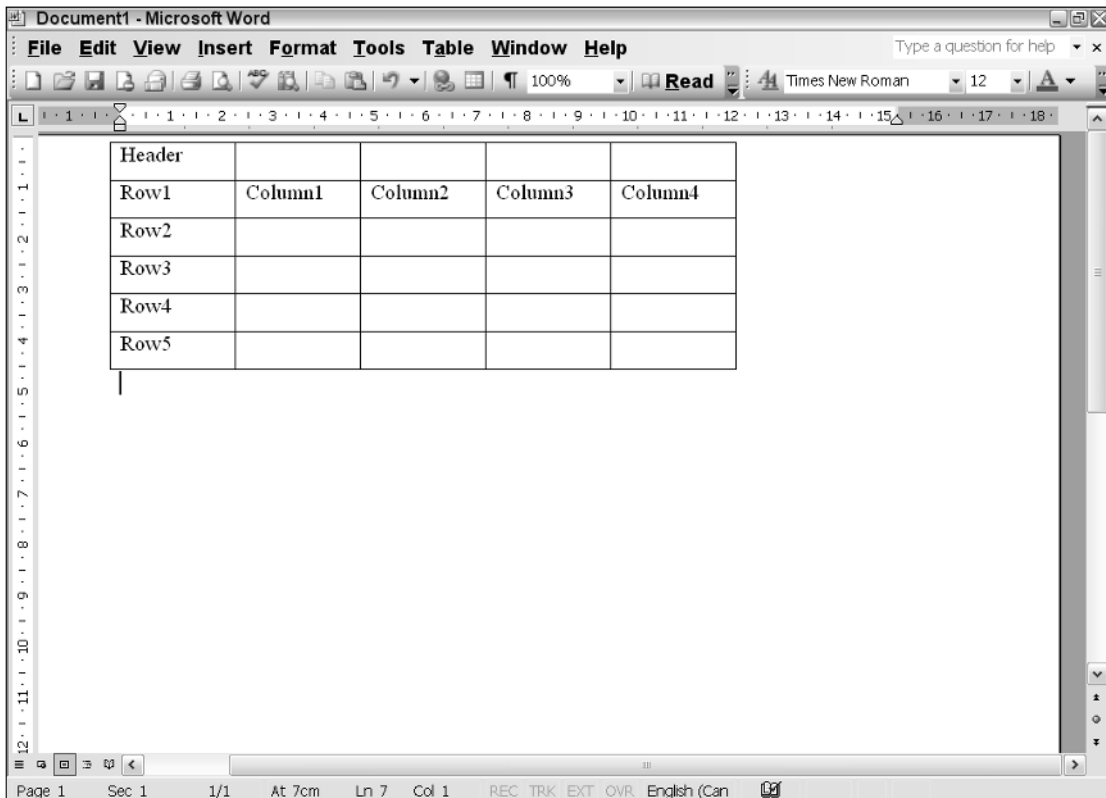


Figure 4-3

If you click the Modify button on the dialog shown in Figure 4-4, you will be able to customize the default styles. Figure 4-5 shows the possible customizations.

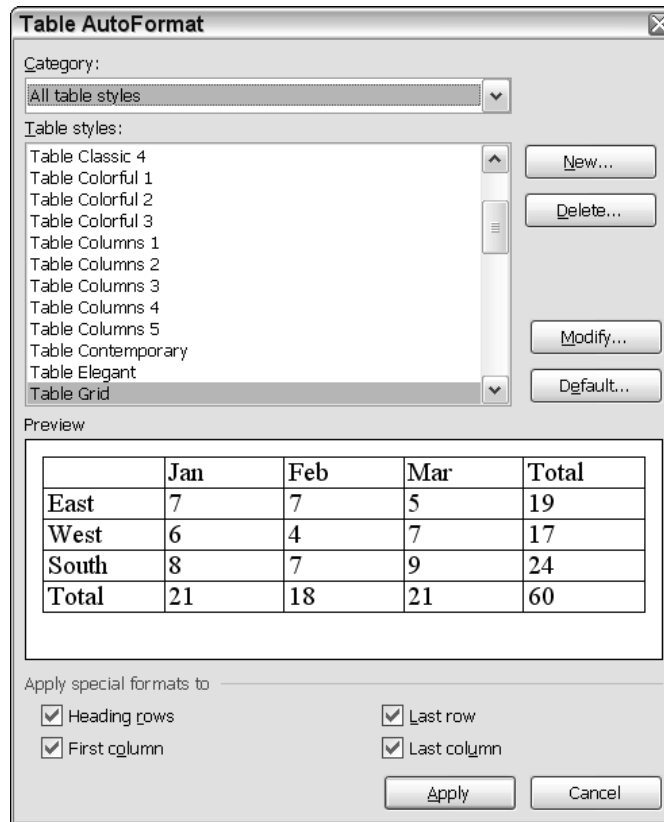


Figure 4-4

Notice that the table is also able to respond to custom styles. There are over 100 table styles that are available. You can view table styles by selecting Table ⇨ Table AutoFormat in a Microsoft Office Word document. Figure 4-4 shows a snapshot of the available styles.

All the customizations that are possible in design mode or through the word menu may also be applied through code. In fact, Listing 4-6 uses one of the custom styles. That sort of advanced table manipulation may be necessary if application requirements dictate a special look and feel or if you simply are not satisfied with the default table offerings. Consider the example shown in Listing 4-7, which shows how to manipulate tables. The example is based on Listing 4-6.

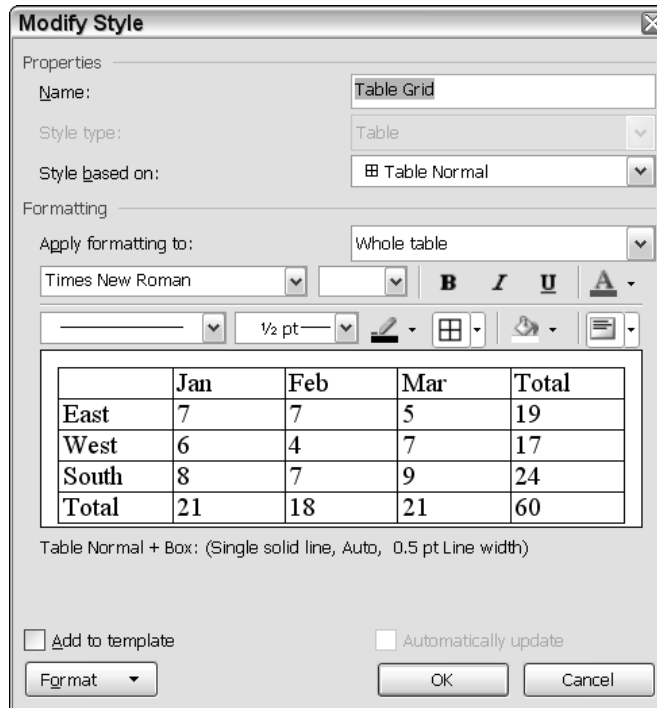


Figure 4-5

Visual Basic

```

Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
Dim rng As Word.Range = Range()
Dim tRange As Word.Range = Range()
Dim table As Word.Table = rng.Tables.Add(rng,3,4, )
table.Rows(1).Cells(1).Shading.BackgroundPatternColor = Word.WdColor.wdColorGray40
table.Rows(1).Cells(1).Range.Text = "Heading 1"
table.Rows(1).Cells(1).Range.Font.Bold = 1
table.Rows(1).Cells(2).Range.Text = "Heading 2"
table.Rows(1).Cells(2).Range.Underline = Word.WdUnderline.wdUnderlineDouble
table.Rows(1).Cells(2).Range.Font.Italic = 1
table.Rows(1).Cells(2).Range.Font.Color = Word.WdColor.wdColorOliveGreen
table.Rows(1).Cells(3).Range.Text = "Heading 3"
table.Rows(3).Cells(1).Shading.BackgroundPatternColor = Word.WdColor.wdColorPaleBlue

Dim row As Object = 1,col As Object = 2
table.Rows(1).Cells(4).Split( row, col)
table.Rows(2).Cells(2).Merge(table.Rows(3).Cells(2))
End Sub

```

C#

```

private void ThisDocument_Startup(object sender, System.EventArgs e)
{
Word.Range rng = Range(ref missing, ref missing);

```

```

Word.Range tRange = Range(ref missing, ref missing);
Word.Table table = rng.Tables.Add(rng, 3, 4, ref missing, ref missing);
table.Rows[1].Cells[1].Shading.BackgroundPatternColor = Word.WdColor.wdColorGray40;
table.Rows[1].Cells[1].Range.Text = "Heading 1";
table.Rows[1].Cells[1].Range.Font.Bold = 1;
table.Rows[1].Cells[2].Range.Text = "Heading 2";
table.Rows[1].Cells[2].Range.Underline = Word.WdUnderline.wdUnderlineDouble;
table.Rows[1].Cells[2].Range.Font.Italic = 1;
table.Rows[1].Cells[2].Range.Font.Color = Word.WdColor.wdColorOliveGreen;
table.Rows[1].Cells[3].Range.Text = "Heading 3";
table.Rows[3].Cells[1].Shading.BackgroundPatternColor =
Word.WdColor.wdColorPaleBlue;

object row = 1, col = 2;
table.Rows[1].Cells[4].Split(ref row, ref col);
table.Rows[2].Cells[2].Merge(table.Rows[3].Cells[2]);
}

```

Listing 4-7

The example in Listing 4-7 adds appropriate pieces of texts as headings. The example shows how to color the background of some cells and add font formatting as well. Once the final piece of formatting has been applied, the code applies a split to some cells followed by a cell merge.

Cell splitting is handy when you need to work with more cell real estate than what is currently allocated. Simply select the current cell or range and call the `Split` method, passing in the number of rows and columns that need to be added, as shown in Listing 4-7.

Cell merging follows the same principle and may be used to gain more cell space by merging adjacent cells into one contiguous block of cells. Notice that after a cell is merged, you can only target the merged cell as a single unit.

Advanced table manipulation requires some attention to detail, so let's outline some guidelines. There is no way to add controls to table cells. That is, you cannot add a dropdown control to a table cell for instance. Cells must only contain textual data. However, you can apply formatting to the text. There is also no support for columnar merging. Columnar merging allows the user to highlight and copy a rectangular block of text on the document surface vertically. This means that a merge operation can only be horizontal or vertical. It cannot be diagonal or columnar.

To avoid runtime exceptions, merging and splitting need to occur on cells that have no prior merge or split directives. You cannot access individual cells if the table contains merged cells. You also cannot merge a split cell—the functionality is mutually exclusive. Any violation of these conditions results in a runtime exception. To avoid these issues, the discerning reader may have noticed that the code in Listing 4-10 added an extra column to create more cell real estate.

Working with Documents

Microsoft Office word functionality revolves around documents. Documents serve as a container for other objects such as text and shapes. Documents, and the objects that are embedded in them, may be

stored and massaged by the user, passed among coworkers, or emailed across the globe. The following section examines document manipulation in some detail.

Manipulating Office Documents

Office documents are complex beasts, but the interface exposed for automation has been simplified so that document manipulation is straightforward. Let's consider an example.

Creating, Saving, and Closing Documents

Listing 4-8 contains some sample code to create a document.

Visual Basic

```
Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    Dim type As Object =
Word.WdDocumentType.wdTypeDocument
    'create a new document
    Dim NewDoc As Object = "normal.dot"
    ThisApplication.Documents.Add( NewDoc, DocumentType:=type )
    Dim prompt As Object = False
    Dim format As Object =
Word.WdOriginalFormat.wdOriginalDocumentFormat
    If ThisApplication.Documents.Count > 1 Then
        prompt = True
    End If
    ThisApplication.Documents.Save( prompt, format)
End Sub
```

C#

```
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    object myDoc = missing;
    object type = Word.WdDocumentType.wdTypeDocument;
    //create a new document
    object newDoc = "normal.dot";
    ThisApplication.Documents.Add(ref newDoc, ref missing, ref type, ref missing);
    object prompt = false;
    object format =
Word.WdOriginalFormat.wdOriginalDocumentFormat;
    if (ThisApplication.Documents.Count > 1)
        prompt = true;
    ThisApplication.Documents.Save(ref prompt, ref format);
}
```

Listing 4-8: New document creation

Let's examine this code in some detail; it will help separate the weeds from the seeds. The `Add` method creates a new document and adds it to the `Documents` collection object. New documents must be based on existing Word templates. This is the reason for the file path pointing to a `.dot` extension. For simplicity, the `normal.dot` file is assumed to reside in the application directory. By default, the file may be

found in the [Drive]:\Documents and Settings\[user name]\Application Data\Microsoft\Templates folder. A deeper probe of the template machinery is available in the “Working with Word Templates” section later in this chapter.

At some point during document creation, two events are fired. The new document event of the application object fires first, followed by the new document event for the newly created document object. We examine documents and events later in this chapter. For now, we note that these events provide two entry points for document customization at the application level and at the document level. For instance, the application document event may be used for printer initialization, because a printer device is usually application specific. However, custom controls that reside on the document surface may need to be initialized inside the document event handler, since these controls are document specific.

Notice how the code examines the `count` property of the `Documents` object collection before saving. The reason for this is that a save request implicitly saves all documents in the collection. The code simply suppresses the prompt for the active document; the user is saving the active document so there really is no need for a prompt. All other documents receive a user prompt.

If you care to examine the third parameter of the `Add` method, you will find that it accepts an enumeration of the following type:

```
Microsoft.Office.Interop.Word.WdDocumentType.wdTypeDocument;
```

You can use appropriate members to create framesets and emails in addition to new documents. To open existing documents, consider Listing 4-9.

Visual Basic

```
Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    Dim type As Object = Word.WdDocumentType.wdTypeDocument
    Dim NewDoc As Object = "c:\myDocument.doc"
    ThisApplication.Documents.Add( NewDoc, DocumentType:=type )
End Sub
```

C#

```
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    object type = Word.WdDocumentType.wdTypeDocument;
    object newDoc = "c:\myDocument.doc";
    ThisApplication.Documents.Add(ref newDoc, ref missing, ref type, ref missing);
}
```

Listing 4-9: Existing document retrieval

A few comments are in order. The document being retrieved must exist at the specified location; otherwise, an exception of type `COMException` is thrown. The exception is also thrown if the document is corrupted or unreadable. For unfortunate situations like these, it makes sense to attempt to repair the document on the fly by displaying the text recovery dialog. Later in the chapter there is code that will illustrate how to display dialogs in Word.

In some instances, you may not require any sort of template processing, or you may simply want to open a file on disk, so it makes sense to use the `Open` method of the document to open files. Consider the example in Listing 4-10.

Visual Basic

```
Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    Dim fileName As Object = "c:\test.txt"
    Dim myDoc as Word.Document = Me.Application.Documents.Open(
        fileName)
End Sub
```

C#

```
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    object fileName = @"c:\test.txt";
    Microsoft.Office.Interop.Word.Document myDoc =
    this.Application.Documents.Open(ref fileName,
    ref missing, ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing, ref missing);
}
```

Listing 4-10: Opening an existing document

The code to open the document fires when the `ThisDocument_Startup` event fires. The event executes the `ThisDocument_Startup` event handler, passing it the document and arguments. One major assumption being made is that the document given by the filename `text.txt` exists and is readable. Otherwise, an exception is thrown. `Open` does not create a new document if one does not exist.

Although the `Open` method is tedious in C#, it is actually more efficient for opening existing documents than using the `Add` method. The reason for the efficiency gains is that the template being used is the default template. No extra work needs to be done to load any custom template. As always, if the file does not exist, a runtime exception occurs.

Spellchecking Functionality

By default, every Microsoft Office document is proofed for spelling and grammar. That functionality is also available through the Word model. An implementation example is shown in Listing 4-11.

Visual Basic

```
Dim spellChecker as Boolean= Me.Application.CheckSpelling("anatomie")
```

C#

```
bool spellChecker = this.Application.CheckSpelling("anatomie",
    ref missing, ref missing, ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing, ref missing, ref missing);
```

Listing 4-11: Spell checking documents

There isn't much to spell checking. A required string representing the items to be spelled checked is passed in to the `CheckSpelling` method. On success, a Boolean value of `true` is returned; `false` is returned otherwise. Note that, if the word does not appear in the dictionary, a value of `false` will be returned, since the Microsoft Office API cannot guarantee that the item is spelled correctly. For this reason, you may decide to funnel the `CheckSpelling` call through a custom dictionary. Simply pass in the path to the custom dictionary, and it will be used to spell check your document.

Working with Word Templates

Documents are always associated with a template. Templates serve as a pattern or mold used to generate new documents. If no template is specified, the default template, `normal.dot`, is used. The association between a document and its template occurs during the creation of a document. At that time, the document inherits styles, page attributes, and content from the template. Once the inheritance of attributes occurs, the document no longer relies on the parent template. Changes made in one do not affect the other. The parent template does not totally drop out of the picture, though. The parent template may provide certain functionality to the document, such as toolbar menus and macros among other things.

Templates may be used to simplify document processing. For instance, invoices, letterheads, fax cover sheets, certificates, and brochures may be produced in a document based on these templates. If you ponder on this for a moment or two, you should realize that a good template is all that is required to provide customized functionality. For instance, it's a lot less hassle to base a document on an appropriate template than it is to build that functionality in using VSTO every time it is needed. If you care to look, there are thousands of free templates available for Microsoft Word and Excel on the Microsoft Office website and other online resources.

Template projects may be created by the usual means, as described in Chapter 1, with the exception that a Word document template is selected in the Templates pane of the New Project dialog, as shown in Figure 4-6.

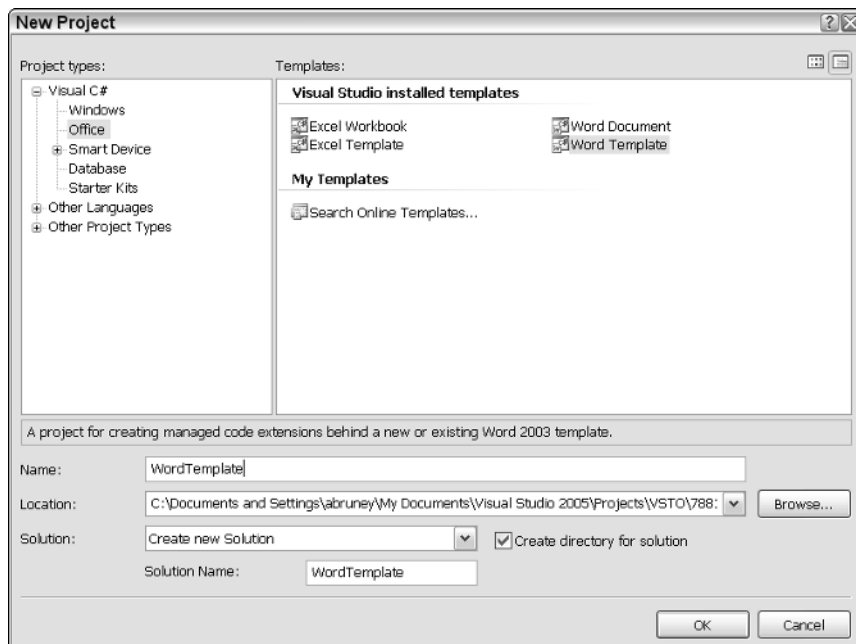


Figure 4-6

The approach is identical to document creation; however, you will notice that a `wordtemplate1.dot` file replaces the `worddocument1.doc` file in the designer. All the .NET controls are available as in regular document development and the `OfficeCodeBehind` file is identical. It bears repeating that the only difference between the two is that the template project produces a document with a `.dot` extension, whereas a word project produces a document with a `.doc` extension.

To see how templates affect VSTO applications, let's write some code to change a template on the fly. Recall that templates are associated with documents at the time of creation. If you would like to associate a document with another existing template, you may either instruct the user to do so or you may do it through code. The end user can associate a document with a new template by selecting Tools ⇄ Templates and Add-ins. Implement the changes according to Figure 4-7.

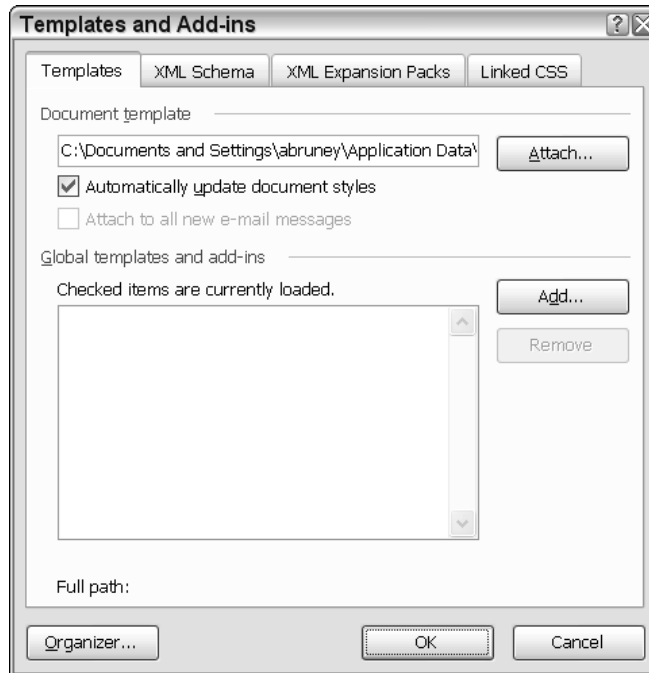


Figure 4-7

Aside from having the user do the task, if you are a programming tough guy, you can make the changes through code. To associate a document with a template, locate one of the default templates installed with Microsoft Office (`C:\Documents and Settings\\Application Data\Microsoft\Templates\`) or download a new template from the web. If you can't easily find one, there are well over 1000 templates on the Microsoft Office website: <http://office.microsoft.com/en-us/templates>.

Choose a template and save it to your application directory so that you can find it easily. Then, create a new VSTO project called `Templating` and accept all the defaults. Add a button to the document. From the property pages, change the `id` to `modify`. Add a file dialog control from the toolbox page. Change its `id` to `openTemplate`. Set the property mask to `Template Files|*.dot`. Once this is done, you are ready to write some code, as shown in Listing 4-12.

Visual Basic

```

Imports System
Imports System.Data
Imports System.Drawing
Imports System.Windows.Forms
Imports Microsoft.VisualStudio.Tools.Applications.Runtime
Imports Word = Microsoft.Office.Interop.Word
Imports Office = Microsoft.Office.Core
Namespace Templating
Public partial Class ThisDocument
Private Sub New_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
End Sub
Private Sub New_Shutdown(ByVal sender As Object, ByVal e As System.EventArgs)
End Sub
Private Sub button1_Click(ByVal sender As Object, ByVal e As EventArgs)
    openTemplate.ShowDialog()
End Sub
Private Sub openTemplate_FileOk(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs)
If openTemplate.FileName <> String.Empty Then
Dim template As Word.Template = DirectCast(Me.AttachedTemplate, Word.Template)
    MessageBox.Show("Current template: " + template.Name)
Me.AttachedTemplate = openTemplate.FileName
    template = DirectCast(Me.AttachedTemplate, Word.Template)
    MessageBox.Show("New template: " + template.Name)
End If
End Sub
End Class
End Namespace

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
namespace Templating
{
public partial class ThisDocument
{
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
}
private void ThisDocument_Shutdown(object sender, System.EventArgs e)
{
}
private void button1_Click(object sender, EventArgs e)
{
    openTemplate.ShowDialog();
}
private void openTemplate_FileOk(object sender,
System.ComponentModel.CancelEventArgs e)
{
}
}
}

```

```
if (openTemplate.FileName != string.Empty)
{
    Word.Template template = Globals.ThisDocument.AttachedTemplate as
    Word.Template;
    MessageBox.Show("Current template: " + template.Name);
    Globals.ThisDocument.AttachedTemplate = openTemplate.FileName;
    template = Globals.ThisDocument.AttachedTemplate as Word.Template;
    MessageBox.Show("New template: " + template.Name);
}
}
}
```

Listing 4-12: Template manipulation

A button click event handler is used to drive the template attaching code. When this event fires, the `openfile` dialog is displayed to allow the user to browse for a template file. Because the filter is set in the `openfile` control, only files with `*.dot` extensions are visible. Once an appropriate template file is chosen, the code first shows the existing template and then attaches a new template. Another message box confirms that the new template is what the user has chosen.

Listing 4-12 uses a button to fire the click event that is responsible for attaching the template. This is bare-bones functionality. Your creativity might lead you to use a toolbar button instead and that is acceptable as well. Also, templating changes only take effect after the document has been closed and reopened.

Toolbar Customization

As previously pointed out in Chapter 2, the toolbar control forms part of the VSTO suite. You can expect to port the same approach for customizing toolbars in the Excel environment and have it work as expected in the Office Word environment. In case you missed that part of the book, Listing 4-13 has a brief review through example.

Visual Basic

```
Dim NewCBarButton As Office.CommandBarButton
Private Sub AddMyToolBar()
    Dim val As Object = False
    Dim NewCommandBar As Office.CommandBar =
Application.CommandBars.Add("myButton", Office.MsoBarPosition.msoBarTop, False, False)
    NewCBarButton = DirectCast
(NewCommandBar.Controls.Add(Office.MsoControlType.msoControlButton, Temporary:=False
), Office.CommandBarButton)
    NewCBarButton.Caption = "my new button"
    NewCBarButton.FaceId = 563
    NewCommandBar.Visible = True
    AddHandler NewCBarButton.Click, AddressOf NewCBarButton_Click
End Sub
```

C#

```
Office.CommandBarButton NewCBarButton = null;
private void AddMyToolBar()
{
```

```

        object val = false;
        Office.CommandBar NewCommandBar =
Application.CommandBars.Add("myButton", Office.MsoBarPosition.msoBarTop,
false, false);
        NewCBarButton =
(Office.CommandBarButton)NewCommandBar.Controls.Add(Office.MsoControlType.msoContro
lButton, missing, missing, missing, false);
        NewCBarButton.Caption = "my new button";
        NewCBarButton.FaceId = 563;
        NewCommandBar.Visible = true;
        NewCBarButton.Click += new
Microsoft.Office.Core._CommandBarButtonEvents_ClickEventHandler(NewCBarButton_Click
);
    }
}

```

Listing 4-13: Toolbar customization

Although the approach has been covered in Chapter 2, let us take some time to review the process. First, an object of type `Office.CommandBar` is created. This object holds a reference to the new button. Next, the code creates a toolbar on which to locate the button. This code is sufficient to display a single button on a new toolbar that displays at the top. Observe that the third parameter is set to true so that the new button is destroyed after our application exits. Otherwise, the button will continue to exist in other Office documents that run on that particular machine.

The rest of the code is concerned with customizing the appearance of the button. The `FaceId` property simply sets the button's face to a predefined icon type. There are a couple thousand predefined types, and the code chooses icon 563 arbitrarily. Finally, the button's click event is wired to an event handler. The event handler is triggered automatically by the Common Language Runtime (CLR) when the button is clicked. The code doesn't win any award for creativity but is straightforward enough so that the explanation should be sufficient.

Now that the basics are safely out of the way, let us examine a challenging requirement. Consider a company that wants to customize a VSTO-based application in such a way that the company logo appears on the button's surface. The implementation is not necessarily trivial, and there are several approaches.

One approach embeds a Windows forms `ImageList` control into the office document. The `ImageList` control stores customized company images. At runtime, the code will create the button and load the image from the `ImageList` control to the new toolbar button face via the system clipboard. The approach is attractive because it loads the images at design time rather than runtime. This is expected to improve performance.

However, there are many issues lurking. The code may fail to work as expected because it is dependent on the system clipboard. If the clipboard is unavailable or some other process copies data over the image before the `CopyFace` method is called, the application will likely misbehave.

One approach is to synchronize access to the clipboard. This approach is certainly appealing at first thought. However, a more tempered analysis reveals it to be a horrible idea. The system clipboard is global in nature and code should not attempt to gain exclusive access to it.

A better approach is simply to do nothing about the potential data corruption. The idea, while absurd, has merit. There is only a small chance of data corruption. This is sufficiently small to not be of concern.

Finally, the user piece will be run on a single desktop, mitigating the risk of concurrent clipboard access during the time that the clipboard manipulation must occur. If Murphy's law were to take effect at this time, the application can simply be closed and reopened to reinitialize the toolbar code.

Another issue is that the code that copies the image from the `ImageList` control to the clipboard assumes that the clipboard does not contain meaningful data and simply overwrites it. In a corporate environment, this is most likely not the case. End users typically copy and paste items while working on applications. The copy/paste combination implicitly uses the system clipboard. After our application runs, the end user would have lost the data. Or worse, the end user may find that a paste operation results in an MSN icon being inserted into a document. This is not necessarily disastrous, but it certainly is in poor taste.

One good approach is to clear the contents of the clipboard after the image has been copied. This will remedy the latter issue. The former issue is not so easy to fix. The code can either prompt the user to proceed with the operation, indicating that the clipboard data may be lost, or the code can internally save the contents of the clipboard to a local variable and restore it once the operation is complete. Finally, the developer must be aware that system clipboard manipulation requires that the appropriate permissions levels be set on the executing assembly. By default, a fully trusted assembly is able to manipulate the system clipboard. Assemblies with less than full trust may need to have the security policy edited in order to gain access to the system clipboard. While this is not a problem for Windows forms (Office assemblies must have full trust to be loaded), it may pose a problem for some server applications or thin clients based on VSTO.

There must be an easier way to do this. The better approach is to use the `picture` property of the command bar object to insert an icon into the toolbar. With this approach, we sidestep all of the above issues. Listing 4-14 shows the code.

Visual Basic

```
Imports WordTools = Microsoft.Office.Tools.Word
Public Class ThisDocument
    Public Class ConvertImage
        Inherits System.Windows.Forms.AxHost
        Public Sub New()
            MyBase.New("59EE46BA-677D-4d20-BF10-8D8067CB8B32")
        End Sub

        Public Shared Function Convert(ByVal Image _
            As System.Drawing.Image) As stdole.IPictureDisp
            Convert = GetIPictureFromPicture(Image)
        End Function
    End Class

    Dim NewCommandBar As Office.CommandBar
    Dim NewCBarButton As Office.CommandBarButton
    Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
        System.EventArgs) Handles Me.Startup
        NewCommandBar = Application.CommandBars.Add("myButton",
            Office.MsoBarPosition.msoBarTop)
        NewCBarButton =
            DirectCast(NewCommandBar.Controls.Add(Office.MsoControlType.msoControlButton,
                Temporary:=False), Office.CommandBarButton)
```

```

        NewCBarButton.Caption = "my new button"
        NewCBarButton.Picture = GetImageResource()
        NewCommandBar.Visible = True
    End Sub
    Private Function GetImageResource() As stdole.IPictureDisp
        Dim tempImage As stdole.IPictureDisp = Nothing

        Dim newIcon As Icon = New Icon("C:\msn.ico")
        Dim newList As New ImageList
        newList.Images.Add(newIcon)
        return ConvertImage.Convert(newList.Images(0))
    End Function

    Private Sub ThisDocument_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Shutdown

        End Sub

    End Class

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
using WordTools = Microsoft.Office.Tools.Word;

namespace WordDocument1
{
    public partial class ThisDocument
    {
        sealed public class ConvertImage : System.Windows.Forms.AxHost
        {
            private ConvertImage()
                : base(null)
            {
            }
            public static stdole.IPictureDisp Convert
                (System.Drawing.Image image)
            {
                return (stdole.IPictureDisp)System.
                    Windows.Forms.AxHost
                    .GetIPictureDispFromPicture(image);
            }
        }

        private void ThisDocument_Startup(object sender, System.EventArgs e)
        {
            object val = false;

```



```
        Office.CommandBar NewCommandBar =
Application.CommandBars.Add("myButton", Office.MsoBarPosition.msoBarTop, false,
false);
        Office.CommandBarButton NewCBarButton =
(Office.CommandBarButton)NewCommandBar.Controls.Add(Office.MsoControlType.msoContro
lButton, missing, missing, missing, false);
        NewCBarButton.Caption = "my new button";
        NewCBarButton.Picture = GetImageResource();
        NewCommandBar.Visible = true;
    }
private stdole.IPictureDisp GetImageResource()
{
    System.Drawing.Icon newIcon = new Icon(@"C:\msn.ico");
    ImageList newList = new ImageList();
    newList.Images.Add(newIcon);

    return ConvertImage.Convert(newList.Images[0]);
}

private void ThisDocument_Shutdown(object sender, System.EventArgs e)
{
}
}
}
```

Listing 4-14: Toolbar customization

The code in Listing 4-14 shouldn't disturb you. We have already explained most of it in Listing 4-13. That portion of the code in the startup event handler simply creates a `CommandBar` button and places it on the toolbar. However, the last line of code is where the magic happens.

The call to `GetImageResource` returns an image that is used on the button's surface. The image is returned by accessing a method on a special class `convert`. The reason for creating the class is that the call to `GetIPictureDispFromPicture` is protected. In order to call this method, we must derive a class based on the parent `System.Windows.Forms.AxHost`. The constructor of the class is necessary plumbing to cause the object to be created correctly.

Inside the body of `GetImageResource`, an icon object is constructed from a path that points to a `.ico` file. In my case, the path points to the `msn` icon file. The icon is then loaded into an `ImageList` control. The contents of the image list control are passed to the `axhost` library so that it can be converted into a picture. When the code is executed, the MSN image should appear as an icon on the toolbar along with an appropriate caption. See Figure 4-8.

Consider a better approach that sidesteps a lot of the thorny issues that plagued the code presented in Listing 4-9.

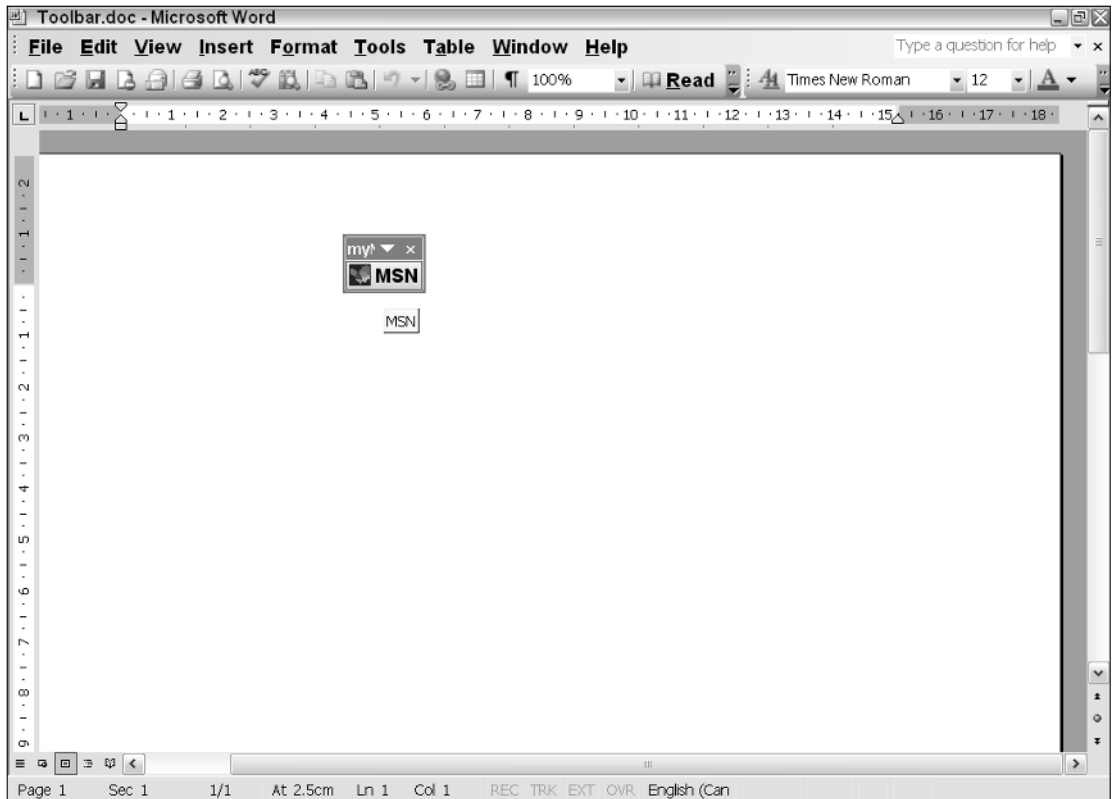


Figure 4-8

Menu Customization

If you've played with VSTO for a bit, you may have observed that the Word default menus are merged in with Visual Studio .NET menus. This is part of the grand design to ensure that both of these environments are seamlessly integrated.

The grand design also ensures that menus are easy to build into your application. In fact, the menu control applies to the VSTO suite as a whole and not just to one particular object in the tools suite. For instance, the code presented in Chapter 2 can be ported without reservation to wire up events to Word menus.

Let's consider an example that builds a menu into the Office bar. Our menu is conceptually simple. We use a dialog box to display the information about the active document. In fact, the code recreates the built-in document statistics functionality that appears in Microsoft Office. The functionality is available on the Spelling and Grammar tab of the standard toolbar control in Office.

Listing 4-15 contains the code.

Visual Basic

```
Dim menubar as Office.CommandBar
Dim cmdBarControl as Office.CommandBarPopup
Dim menuCommand as Office.CommandBarButton

' Add the menu.
menubar = DirectCast (Application.CommandBars.ActiveMenuBar, Office.CommandBar)
Office.CommandBarPopup cmdBarControl = DirectCast( menubar.Controls.Add(
Office.MsoControlType.msoControlPopup, Before:= menubar.Controls.Count, True),
Office.CommandBarPopup)
If cmdBarControl IsNot Nothing Then
    cmdBarControl.Caption = "&Refresh"
    ' Add the menu command.
    menuCommand = DirectCast( cmdBarControl.Controls.Add(
        Office.MsoControlType.msoControlButton, Temporary:= True),
Office.CommandBarButton)
    menuCommand.Caption = "&Calculate"
    menuCommand.Tag = DateTime.Now.ToString()
    menuCommand.FaceId = 265
    AddHandler menuCommand.Click, AddressOf menuCommand_Click
End If
```

C#

```
Office.CommandBar menubar;
Office.CommandBarPopup cmdBarControl;
Office.CommandBarButton menuCommand;
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    // Add the menu.
    menubar = (Office.CommandBar)Application.CommandBars.ActiveMenuBar;
    cmdBarControl = (Office.CommandBarPopup)menubar.Controls.Add(
Office.MsoControlType.msoControlPopup, missing, missing, menubar.Controls.Count,
true);
    if (cmdBarControl != null)
    {
        cmdBarControl.Caption = "&Tools";
        // Add the menu command.
        menuCommand = (Office.CommandBarButton)cmdBarControl.Controls.Add(
Office.MsoControlType.msoControlButton, missing, missing, missing, true);
        menuCommand.Caption = "&Word Count...";
        menuCommand.Tag = DateTime.Now.ToString();
        menuCommand.Click += new
Microsoft.Office.Core._CommandBarButtonEvents_ClickEventHandler(menuCommand_Click);
    }
}
void menuCommand_Click(Microsoft.Office.Core.CommandBarButton Ctrl, ref bool
CancelDefault)
{
    Word.Dialog dlg =
Application.Dialogs[Word.WdWordDialog.wdDialogDocumentStatistics];
    object timeOut = 0;
    dlg.Show(ref timeOut);
}
}
```

Listing 4-15: Menu customization

The code is straightforward and the general approach has been explained in Chapter 2. However, we'll examine the important parts briefly. First, a reference is obtained for the `ActiveMenuBar` object. Next, a menu item is added using the reference obtained earlier. Once the menu item has been added successfully, the code subscribes to the click event so that menu click notifications invoke the registered handler, `menuCommand_Click`.

Notice too that the `Caption` and `FaceId` are purely cosmetic. However, the `Tag` property is not. As explained in Chapter 2, the `Tag` property needs to be unique in order for the event hook up to fire successfully. The assignment to the current time satisfies the unique condition.

The point of the exercise is to show that Microsoft Office contains a slew of built-in components that provide much needed document functionality that would be expensive and tedious to create otherwise. More so, because of the flexibility in the VSTO architecture, these components can be used outside of an Office document application.

Consider a custom application developed with Windows forms that needs to provide some sort of document statistics functionality to the end user. For that type of application, it is easier to hook into the VSTO Word document statistics dialog and let that component provide the raw statistics as opposed to writing custom code to provide that type of functionality. To harness the functionality, the application can simply maintain a hidden document and import the custom data into the hidden document. Then, the code can create the appropriate Word dialog to generate the necessary document statistics.

But there are some drawbacks that you should be aware of. Word dialogs are not intrinsically attached to the document. They become aware of the document only at runtime when the dialog is created. At this time, and only at this time, the dialog box discovers the applicable properties of the document.

As an example, consider the code that needs to set the `PageWidth` of a document. You cannot simply create a dialog box object and set its `PageWidth` property in C#. You will need to use reflection to obtain the available properties and methods in C#. For Visual Basic, you will need to turn off `Option strict` or use VB's `CallByName` function. Listing 4-16 shows some sample code to demonstrate the process.

Visual Basic

```
Dim dlgType as Word.Dialog =
Application.Dialogs(Word.WdWordDialog.wdDialogFilePageSetup)

'this will not compile
dlg.PageWidth = 10
'this will work at run-time
CallByName(dlgType, 'PageWidth', CallType.Set, 10)
' Display the dialog box.
Dlg.Execute()
```

C#

```
Word.Dialog dlgType = Application.Dialogs[Word.WdWordDialog.wdDialogFilePageSetup];
//this will not compile
dlgType.PageWidth = 10;
//this will work at run-time
dlgType.GetType().InvokeMember("PageWidth",
    System.Reflection.BindingFlags.SetProperty |
    System.Reflection.BindingFlags.Public |
```

```
        System.Reflection.BindingFlags.Instance,
        null, dlgType, new object[] {10},
        System.Globalization.CultureInfo.InvariantCulture);
// Display the dialog box.
dlgType .Execute();
```

Listing 4-16: Sample late binding code

Hopefully, the ugly C# code should scare you into using Visual Basic. If it doesn't, consider wrapping the eyesore in a method that takes the appropriate parameters and calls the `InvokeMember`.

Working with Other Office Applications

Under the crust, all Microsoft Office Applications are COM servers. Any application that requires interoperation with these servers must be able to interoperate with COM. VSTO is no exception. Fortunately, the interaction between managed code and a COM server occurs seamlessly, so there is no need to explicitly write ugly Interop code. Since every Office API exposes a well-defined interface for automation purposes, let's consider an automation example.

A PowerPoint Automation Example

Listing 4-17 shows an example of automation with PowerPoint.

Visual Basic

```
Imports System
Imports System.Data
Imports System.Drawing
Imports System.Windows.Forms
Imports Microsoft.VisualStudio.Tools.Applications.Runtime
Imports Word = Microsoft.Office.Interop.Word
Imports Office = Microsoft.Office.Core
Imports PowerPoint = Microsoft.Office.Interop.PowerPoint
Imports Graph = Microsoft.Office.Interop.Graph
Imports System.Runtime.InteropServices
Namespace PowerPoint
Public partial Class ThisDocument
Private Sub SlideShow()
Dim background As String = "C:\Program Files\Microsoft
Office\Templates\Presentation Designs\Blends.pot"
Dim stringStyle As String = "C:\Windows\Blue Lace 16.bmp"
Dim objApp As PowerPoint.Application = New PowerPoint.Application()
objApp.Visible = Microsoft.Office.Core.MsoTriState.msoTrue
Dim presentContext As PowerPoint.Presentations = objApp.Presentations
Dim presenter As PowerPoint._Presentation =
presentContext.Open(background, Microsoft.Office.Core.MsoTriState.msoFalse, Microsoft
.Office.Core.MsoTriState.msoTrue, Microsoft.Office.Core.MsoTriState.msoTrue)
Dim objSlides As PowerPoint.Slides = presenter.Slides
'Build slide #1:
Dim name As PowerPoint.Slide =
objSlides.Add(1, PowerPoint.PpSlideLay.ppLayoutTitleOnly)
```

```

Dim textSlide As PowerPoint.TextRange = name.Shapes(1).TextFrame.TextRange
textSlide.Text = "Programming VSTO 2005"
textSlide.Font.Size = 20
name.Shapes.AddPicture(stringStyle, Microsoft.Office.Core.MsoTriState.msoFalse,
Microsoft.Office.Core.MsoTriState.msoTrue, 150, 150, 500, 350)
name = objSlides.Add(2, PowerPoint.PpSlideLay.ppLayoutBlank)
name.FollowMasterBackground = Microsoft.Office.Core.MsoTriState.msoFalse
Dim objShapes As PowerPoint.Shapes = name.Shapes
Dim objShape as PowerPoint.Shape =
objShapes.AddTextEffect(Microsoft.Office.Core.MsoPresetTextEffect.msoTextEffect27,
"Buy Now!!!", "Arial", 90, Microsoft.Office.Core.MsoTriState.msoFalse,
Microsoft.Office.Core.MsoTriState.msoFalse, 200, 200)
Dim counter() As Integer = New Integer(2) {}
Dim i As Integer
For i = 0 To 2- 1 Step 1
    counter(i) = i + 1
Next
Dim objSldRng As PowerPoint.SlideRange = objSlides.Range(counter)
Dim objSST As PowerPoint.SlideShowTransition = objSldRng.SlideShowTransition
objSST.AdvanceOnTime = Microsoft.Office.Core.MsoTriState.msoTrue
objSST.AdvanceTime = 6
objSST.EnTryEffect = PowerPoint.PpEnTryEffect.ppEffectBlindsVertical
'Run the Slide
Dim objSSS As PowerPoint.SlideShowSettings = presenter.SlideShowSettings
objSSS.StartingSlide = 1
objSSS.EndingSlide = 2
objSSS.Run()
presenter.Close()
objApp.Quit()
End Sub
Private Sub New_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
SlideShow()
End Sub
Private Sub New_Shutdown(ByVal sender As Object, ByVal e As System.EventArgs)
GC.Collect()
End Sub
End Class
End Namespace

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
using PowerPoint;
using Graph = Microsoft.Office.Interop.Graph;
using System.Runtime.InteropServices;
namespace PowerPoint
{
public partial class ThisDocument
{
private void SlideShow()

```

```
{
    string background = @"C:\Program Files\Microsoft Office\Templates\Presentation
    Designs\Blends.pot";
    string stringStyle = @"C:\Windows\Blue Lace 16.bmp";
    PowerPoint.Application objApp = new PowerPoint.Application();
    objApp.Visible = Microsoft.Office.Core.MsoTriState.msoTrue;
    PowerPoint.Presentations presentContext = objApp.Presentations;
    PowerPoint._Presentation presenter = presentContext.Open(background,
    Microsoft.Office.Core.MsoTriState.msoFalse,
    Microsoft.Office.Core.MsoTriState.msoTrue,
    Microsoft.Office.Core.MsoTriState.msoTrue);
    PowerPoint.Slides objSlides = presenter.Slides;
    //Build slide #1:
    PowerPoint.Slide name = objSlides.Add(1,
    PowerPoint.PpSlideLayout.ppLayoutTitleOnly);
    PowerPoint.TextRange textSlide = name.Shapes[1].TextFrame.TextRange;
    textSlide.Text = "Programming VSTO 2005";
    textSlide.Font.Size = 20;
    name.Shapes.AddPicture(stringStyle, Microsoft.Office.Core.MsoTriState.msoFalse,
    Microsoft.Office.Core.MsoTriState.msoTrue, 150, 150, 500, 350);
    name = objSlides.Add(2, PowerPoint.PpSlideLayout.ppLayoutBlank);
    name.FollowMasterBackground = Microsoft.Office.Core.MsoTriState.msoFalse;
    PowerPoint.Shapes objShapes = name.Shapes;
    PowerPoint.Shape objShape =
    objShapes.AddTextEffect(Microsoft.Office.Core.MsoPresetTextEffect.msoTextEffect27,
    "Buy Now!!!", "Arial", 90, Microsoft.Office.Core.MsoTriState.msoFalse,
    Microsoft.Office.Core.MsoTriState.msoFalse, 200, 200);
    int[] counter = new int[2];
    for (int i = 0; i < 2; i++)
    counter[i] = i + 1;
    PowerPoint.SlideRange objSldRng = objSlides.Range(counter);
    PowerPoint.SlideShowTransition objSST = objSldRng.SlideShowTransition;
    objSST.AdvanceOnTime = Microsoft.Office.Core.MsoTriState.msoTrue;
    objSST.AdvanceTime = 6;
    objSST.EntryEffect = PowerPoint.PpEntryEffect.ppEffectBlindsVertical;
    //Run the Slide
    PowerPoint.SlideShowSettings objSSS = presenter.SlideShowSettings;
    objSSS.StartingSlide = 1;
    objSSS.EndingSlide = 2;
    objSSS.Run();
    presenter.Close();
    objApp.Quit();
}
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    SlideShow();
}
private void ThisDocument_Shutdown(object sender, System.EventArgs e)
{
    GC.Collect();
}
}
```

Listing 4-17: PowerPoint automation

PowerPoint slides are made up of individual slides that are displayed on screen in sequential fashion till the end. With that in mind, the most challenging task is to create the individual slides. In Listing 4-17, the code sets up the background and picture styles for each slide. By default, Microsoft Office installs some default backgrounds that are convenient for demonstration purposes.

Next, the required objects are created. Automating other Microsoft Office applications has some key ingredients, namely creating an instance of the automation server. In Listing 4-17, an instance of the PowerPoint object is created and some basic object plumbing is set up.

```
objApp.Visible = Microsoft.Office.Core.MsoTriState.msoTrue
objPresSet = objApp.Presentations
objPres = objPresSet.Open(strTemplate,
Microsoft.Office.Core.MsoTriState.msoFalse,
Microsoft.Office.Core.MsoTriState.msoTrue,
Microsoft.Office.Core.MsoTriState.msoTrue)
objSlides = objPres.Slides
```

Then, the PowerPoint harness is initialized using the `Open` method call. Each slide that forms part of the PowerPoint presentation is then created using the following code.

```
'Build Slide #1:
'Add text to the slide, change the font and insert/position a
'picture on the first slide.
objSlide = objSlides.Add(1, PowerPoInteger.PpSlideLay.ppLayoutTitleOnly)
objTextRng = objSlide.Shapes(1).TextFrame.TextRange
objTextRng.Text = "My Sample Presentation"
objTextRng.Font.Name = "Comic Sans MS"
objTextRng.Font.Size = 48
objSlide.Shapes.AddPicture(strPic, Microsoft.Office.Core.MsoTriState.msoFalse,
Microsoft.Office.Core.MsoTriState.msoTrue,
150, 150, 500, 350)
```

Finally, the code displays the slides that form part of the PowerPoint application. Other Office applications may be automated using this same approach. First, import the required namespace and then create an instance of the application. Then, make the appropriate method calls to automate the object. The default location for most of the type libraries is `C:\Program Files\Microsoft Office\Office[version]`. The garbage collector call is provided as a safe guard to ensure that a collection is performed deterministically.

An Office Automation Executable Example

The material presented so far in this chapter focused on automating Office examples from inside VSTO. The example in Listing 4-18 demonstrates how to automate Office applications from a console application.

Visual Basic

```
Imports System
Imports System.Runtime.InteropServices
Imports Word = Microsoft.Office.Interop.Word
Imports Office = Microsoft.Office.Core

namespace ConsoleAutomation
```



```
class Program

    Shared Sub Main(ByVal args() As String)
    Dim automator As Object
    Try
        automator = Marshal.GetActiveObject("Word.Application")
        Dim app As Word.Application = automator as Word.Application
        app.ShowClipboard()
    Catch
        'Microsoft Word is not running so no automation can occur
    Finally
        Marshal.ReleaseComObject(automator)
    End Try
    End Sub
End namespace
```

C#

```
using System;
using System.Runtime.InteropServices;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;

namespace ConsoleAutomation
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                object automator = Marshal.GetActiveObject("Word.Application");
                Word.Application app = automator as Word.Application;
                app.ShowClipboard();
            }
            catch (System.Runtime.InteropServices.COMException)
            {
                //Microsoft Word is not running so no automation can occur
            }
            finally
            {
                Marshal.ReleaseComObject(automator);
            }
        }
    }
}
```

Listing 4-18: Sample late binding code

The console application is a stand-alone application that runs outside of the Office application. Notice how the code creates an instance of the appropriate Office Word object. Once this is accomplished, all the functionality of Word can be leveraged through the reference to the object. In our instance, we simply show the clipboard in the taskbar.

Listing 4-22 contains exception handling for the case where the automation is attempted and an instance of Microsoft Word is not running. In that case, you can either exit with an appropriate message or attempt to start an instance of Microsoft word. The code calls the `ReleaseComObject` method to ensure that resources are cleaned up correctly. This call is necessary because the example illustrates regular COM Interop.

A Smart Tag Example

If you have worked with Office XP and subsequent versions for a while, you will notice that Microsoft Word tags some words in the document. When you hover over these words, an icon appears containing a drop-down menu that enables more functionality associated with these particular words. Figure 4-9 shows smart tagging in action.

Documents that support smart tags must have that feature that enables for smart tagging to function correctly. In an Office Word application, turn on the feature by selecting **Tools** ⇨ **AutoCorrect Options**, select the **Smart Tags** tab, and click the "Label text with smart tags" checkbox, as shown in Figure 4-11. You may want to consider turning this feature off if you are not particularly impressed with the smart tag functionality.

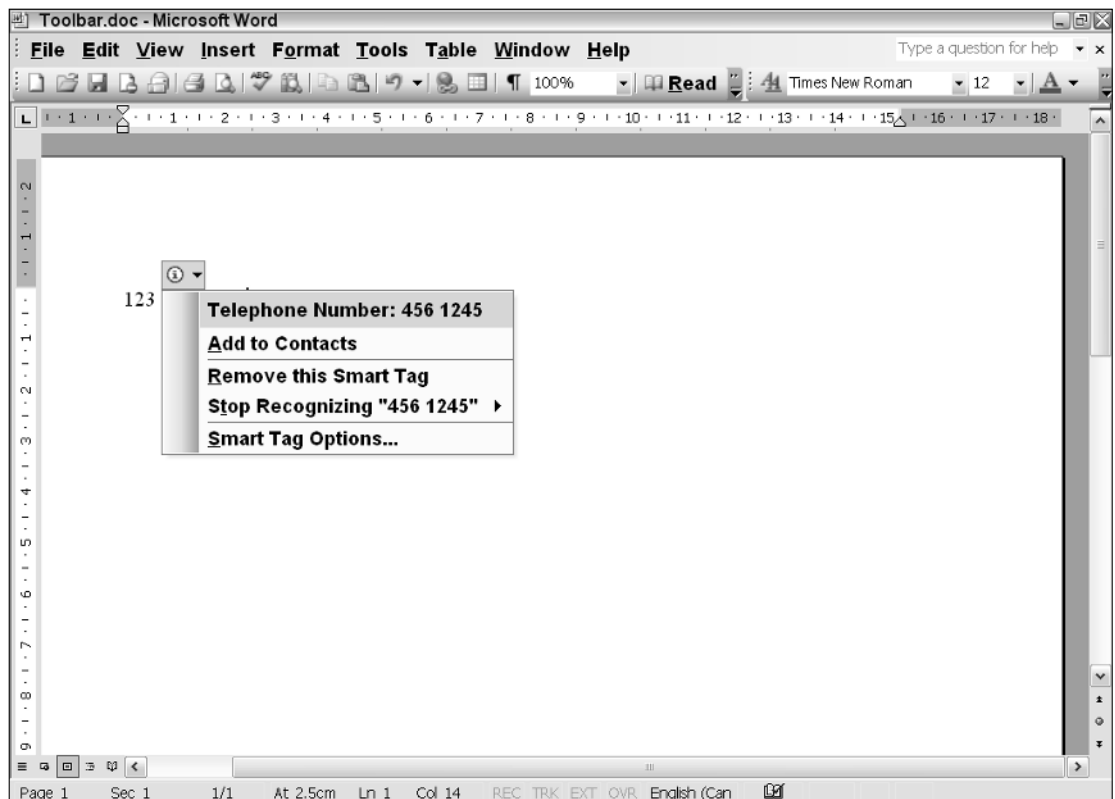


Figure 4-9

There are a couple of ways to create smart tags in Office documents. One approach creates tags at design time using XML, the other creates smart tags at runtime. Let's consider the design-time approach first. After examining the XML fragment shown in Listing 4-19, copy it into Notepad or download the sample from the book's website at www.wrox.com. Save the file as `Books.xml` in the default location that Office uses to load smart tag functionality: `C:\Program Files\Common Files\Microsoft Shared\Smart Tags\Lists`. The action creates a Microsoft Office Smart Tag List (MOSTL) file that defines the behavior of smart tags in the active document.

```
<FL:smarntaglist xmlns:FL="urn:schemas-microsoft-com:smarctags:list">
  <FL:name>Book Index</FL:name>
  <FL:lcid>1033,0</FL:lcid>
  <FL:description>Available Books</FL:description>
  <FL:updateable>>false</FL:updateable>
  <FL:updatefrequency>10080</FL:updatefrequency>
  <FL:autoupdate>>true</FL:autoupdate>
  <FL:smarntag type="urn:schemas-microsoft-com:office:smarctags#BooksIndex">
    <FL:caption>Book Indices</FL:caption>
    <FL:terms>
<FL:termlist>VSTO,OWC,.NET</FL:termlist>
      </FL:terms>
      <FL:actions>
<FL:action id="BookId">
        <FL:caption>Further Reading</FL:caption>
        <FL:url>http://{TEXT}.amazon.com</FL:url>
      </FL:action>
    </FL:actions>
  </FL:smarntag>
</FL:smarntaglist>
```

Listing 4-19: Sample XML file

In order for the smart tag to work in Word, you must save the file with an XML extension in the smart tag directory given above. Then, select `Tools` ⇨ `AutoCorrect Options`. Click the `Smart tags` tab. Finally, click the `Recheck Document` button and accept the confirmation dialog that pops up. Figure 4-10 shows the functionality in action based on our XML file. You may want to consider using design-time smart tagging as the opportunities present themselves.

In some cases, design-time smart tagging may not be feasible so let's consider a runtime implementation. Our example simply implements the first approach of Listing 4-14. Listing 4-20 has the code.

Visual Basic

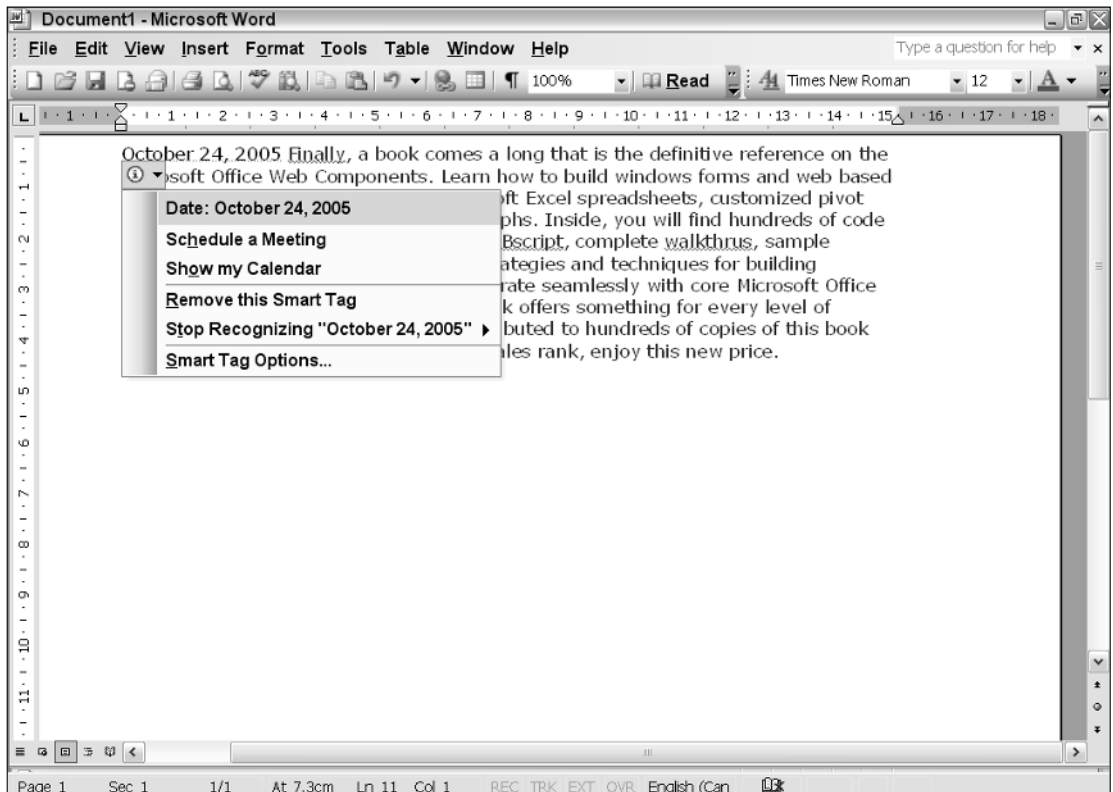
```
Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
  Dim myTag As WordTools.SmartTag = New
WordTools.SmartTag("http://www.lulu.com/owc#OWC", "OWC")
  myTag.Terms.Add("OWC")
  myTag.Terms.Add("VSTO")
  myTag.Terms.Add(".NET")
  Dim act As WordTools.Action = New WordTools.Action("B&ooks///&Go")
  AddHandler act.Click, AddressOf act_Click
  myTag.Actions = New WordTools.Action() {act}
  VstoSmartTags.Add(myTag)
```

C#

```

private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    WordTools.SmartTag myTag = new
WordTools.SmartTag("http://www.lulu.com/owc#OWC", "OWC");
    myTag.Terms.Add("OWC");
    myTag.Terms.Add("VSTO");
    myTag.Terms.Add(".NET");
    WordTools.Action act = new WordTools.Action("B&ooks///&Go");
    act.Click += new
Microsoft.Office.Tools.Word.ActionClickEventHandler(act_Click);
    myTag.Actions = new WordTools.Action[] { act };
    VstoSmartTags.Add(myTag);
}

```

Listing 4-20: Showing runtime smart tag implementation**Figure 4-10**

The code is fairly simple. A new smart tag object is created that points to a particular URI. Notice that the URI must contain two parts separated by the # character otherwise a runtime exception is thrown. The terms are then added to the smart tag object collection so that the smart tag feature can recognize the specified text in the document proper and take appropriate action. That's really all there is to smart tagging.

You can see that it is easy to get creative with that type of functionality because it is possible to hook up events to the tag click event, as Listing 4-20 shows. That option is not available in the XML approach, so you may consider using the runtime approach when smart tags need to be linked with complex resources or when some action or event needs to occur.

Another option is to download and install the Microsoft Smart Tag SDK from the Microsoft Office website. The SDK comes complete with sample implementation and quick start guide. For this reason, an example is not provided here. Be aware though that the SDK creates smart tags through COM Interop. VSTO has support for smart tags built in so you should prefer the VSTO approach to the SDK to avoid the overhead of COM Interop.

If you care to explore smart tags in more detail, you will find that the technology also supports regular expression filtering. Regular expressions enable complex use cases and are well outside the scope of this book. A good resource for regular expressions is www.regexlib.com. Figure 4-11 shows part of the `time.xml` file that is available with Microsoft Office 2003. The `time.xml` MOSTL contains its fair share of regular expressions for parsing time tags in an office document.

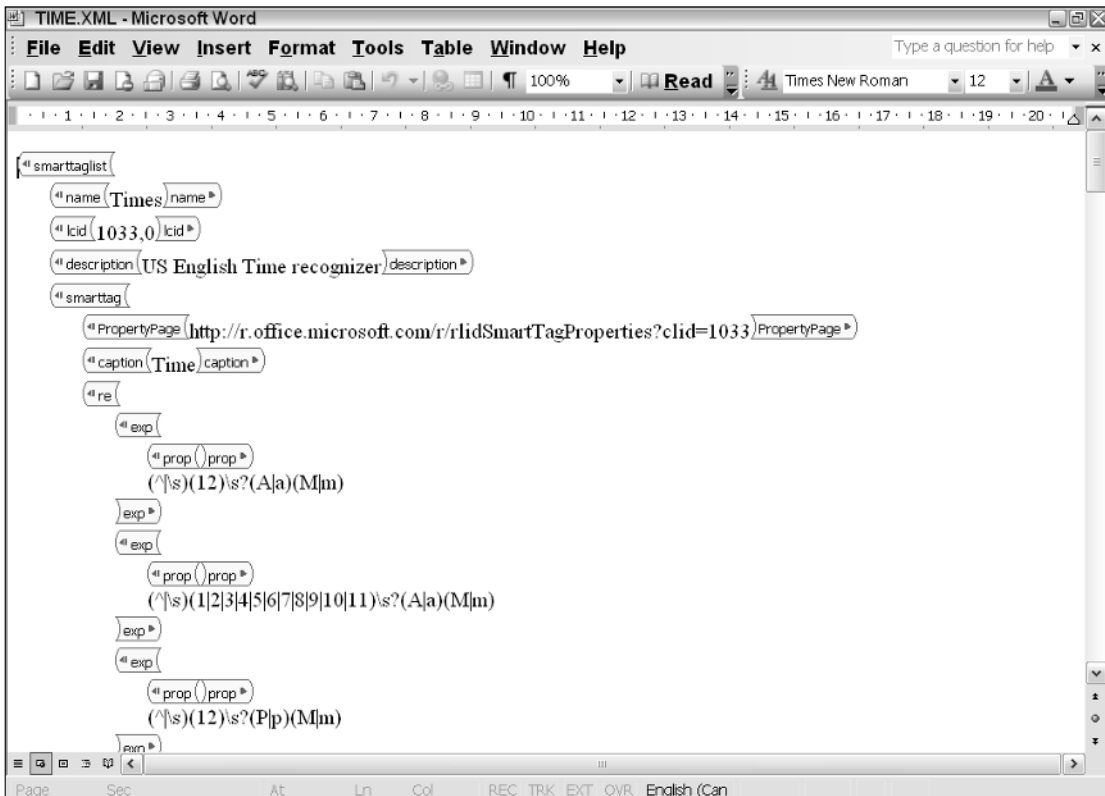


Figure 4-11

Manipulating Office Controls

You will find several examples of manipulating controls in VSTO in this book. More so, the code listed for control manipulation in the Excel chapters is applicable here as well. However, there are a few interesting controls that have surfaced in VSTO that deserve some special attention, so let's consider two of these.

`ActionsPane` controls and `backgroundWorker` controls are new for Visual Studio Tool for Office System. The `backgroundWorker` control may be added to the form from the toolbox. The control is designed to perform long-running tasks in the background. One reason for this control is that it allows the user interface to remain responsive while a resource intensive task is being performed. Symptoms of these long-running tasks usually involve a screen that appears to be frozen, unresponsive, or improperly painted during application execution.

Consider the conceptually simple example in Listing 4-21, which performs a long-running task.

Visual Basic

```
Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    backgroundWorker1.RunWorkerAsync()
End Sub
Private Sub ThisDocument_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs)
End Sub
Private Sub backgroundWorker1_DoWork(ByVal sender As Object, ByVal e As
System.ComponentModel.DoWorkEventArgs)
    System.Threading.Thread.Sleep(2400)
End Sub
```

C#

```
Private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    backgroundWorker1.RunWorkerAsync();
}
Private void ThisDocument_Shutdown(object sender, System.EventArgs e)
{
}
Private void backgroundWorker1_DoWork(object sender,
System.ComponentModel.DoWorkEventArgs e)
{
    System.Threading.Thread.Sleep(2400);
}
```

Listing 4-21: Long-running task execution

This example assumes that a `backgroundWorker` control has been dropped onto the document. At the start of the application, the `backgroundWorker` control starts the long running task with a call to `RunWorkerAsync` method. This call fires the `DoWork` event handler where the code simulates a long-running task. At this point, there are two separate paths of execution. The first execution path represents the main application, and the other represents the long running task.

Chapter 4

The laws that govern the way a `backgroundWorker` control operates with the main application are certainly not trivial and many books have been written about this complex relationship. However, there isn't space in this book to discuss this topic. Certainly, the introduction of this control necessarily trivializes the relationship to the point that it may introduce instability to an otherwise well-written application.

Let's proceed with an example to illustrate our point, in Listing 4-22. Hopefully, it will make things clearer.

Visual Basic

```
Imports System
Imports System.Data
Imports System.Drawing
Imports System.Windows.Forms
Imports Microsoft.VisualStudio.Tools.Applications.Runtime
Imports Word = Microsoft.Office.Interop.Word
Imports Office = Microsoft.Office.Core
Namespace WordActionsPane
Public partial Class ThisDocument
Private Sub New_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
backgroundWorker1.RunWorkerAsync()
End Sub
Private Sub New_Shutdown(ByVal sender As Object, ByVal e As System.EventArgs)
End Sub
Private Sub backgroundWorker1_DoWork(ByVal sender As Object, ByVal e As
System.ComponentModel.DoWorkEventArgs)
Dim i As Integer
For i = 1 To 10 Step 1
System.Threading.Thread.Sleep(500)
this.ActionsPane.Visible = Not this.ActionsPane.Visible
Next
End Sub
End Class
End Namespace
```

C#

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
namespace WordActionsPane
{
public partial class ThisDocument
{
private void ThisDocument_Startup(object sender, System.EventArgs e)
{
backgroundWorker1.RunWorkerAsync();
}
private void ThisDocument_Shutdown(object sender, System.EventArgs e)
{
}
}
```

```

private void backgroundWorker1_DoWork(object sender,
System.ComponentModel.DoWorkEventArgs e)
{
    for (int i = 1; i < 10; i++)
    {
        System.Threading.Thread.Sleep(500);
        this.ActionsPane.Visible = ! this.ActionsPane.Visible;
    }
}
}
}
}

```

Listing 4-22: Background control threaded interaction

Listing 4-22 includes a `backgroundWorker` control on a form. The method `DoWork` is responsible for executing the long-running task. In the example, the long-running task is a loop simulation that causes the application to sleep for 500 milliseconds each time the loop executes. On each pass of the loop, the application will toggle the `ActionsPane` visibility. The visibility property does not actually hide the actions pane, it merely hides the controls that are embedded in the `ActionsPane`. However, this activity is just what is needed.

If you execute the code in Listing 4-22, before long you should receive a runtime exception, indicating that a background thread cannot modify a UI component. But then again, on some Windows platforms, the code may actually execute without issue. In both cases, the outcome is disastrous albeit more so in the latter case, since this bug can lie undiscovered until the code is deployed in production.

The `backgroundWorker` control is actually a wrapper for a separate thread of execution — a mini-program or task of sorts — that is allowed to execute while the main application is running. In essence, there are two paths of execution, one for the main application that fires the `Startup` routine and another path that fires the long-running task through the `backgroundWorker` control.

The long-running task, or child task, is forbidden to manipulate any UI object belonging to the parent task. In Listing 4-22, the UI component is the `ActionsPane`, and it belongs to the parent task. Had we used a button, text box, or any Windows control, the result would be the same because these Windows controls are owned by the parent task.

If the child task needs to modify a UI object owned by the parent task, it must formerly ask the parent task to do the modification for it. Consider the example of this formal request in Listing 4-23.

Visual Basic

```

Imports System
Imports System.Data
Imports System.Drawing
Imports System.Windows.Forms
Imports Microsoft.VisualStudio.Tools.Applications.Runtime
Imports Word = Microsoft.Office.Interop.Word
Imports Office = Microsoft.Office.Core
Namespace WordActionsPane
Public partial Class ThisDocument
Private Sub New_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
backgroundWorker1.RunWorkerAsync()
End Sub

```



```
Private Sub New_Shutdown(ByVal sender As Object, ByVal e As System.EventArgs)
End Sub
Private Sub backgroundWorker1_DoWork(ByVal sender As Object, ByVal e As
System.ComponentModel.DoWorkEventArgs)
    Dim i As Integer
    For i = 1 To 10- 1 Step i + 1
        System.Threading.Thread.Sleep(500)
        If (ActionsPane.InvokeRequired = true) then
            ActionsPane.Invoke(new MethodInvoker(AddressOf ActionsPane.Hide()))
        Else
            ActionsPane.Hide()
        EndIf
    Next
End Sub
End Class
End Namespace
```

C#

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
namespace WordActionsPane
{
    public partial class ThisDocument
    {
        private void ThisDocument_Startup(object sender, System.EventArgs e)
        {
            backgroundWorker1.RunWorkerAsync();
        }
        private void ThisDocument_Shutdown(object sender, System.EventArgs e)
        {
        }
        private void backgroundWorker1_DoWork(object sender,
System.ComponentModel.DoWorkEventArgs e)
        {
            for (int i = 1; i < 10; i++)
            {
                System.Threading.Thread.Sleep(500);
                If (ActionsPane.InvokeRequired)
                    ActionsPane.Invoke(new MethodInvoker(ActionsPane.Hide()));
                Else
                    ActionsPane.Hide();
            }
        }
    }
}
```

Listing 4-23: UI manipulation by background controls

In Listing 4-26 you can see that the child task (long-running task) explicitly asks the parent task, who is the owner of the control, to do some work on its behalf. And the work is done without triggering an exception. The request is handled using the `Invoke` line of code. That sort of formal communication is the only way a child task can manipulate a control on the parent task. Remember, you shouldn't expect the code to actually hide the actions pane control. The `hide` method call hides the controls embedded in the actions pane. To be certain, there are many other rules that come into effect with background controls so it is worthwhile to spend some time poring through MSDN and other resources to come up to speed on thread technology.

Word Event Model

The `OfficeCodeBehind` class containing the code behind the Office document model defines at least two events for the Word object. The `ThisDocument_Open` fires when either a document or a template opens. The `thisDocument_Close` fires when either a document or a template closes. You may use both these events to perform document initialization or cleanup as appropriate.

VSTO makes it easy to wire up additional events in your applications. Listing 4-24 is an example of this.

Visual Basic

```
Imports System
Imports System.Data
Imports System.Drawing
Imports System.Windows.Forms
Imports Microsoft.VisualStudio.Tools.Applications.Runtime
Imports Word = Microsoft.Office.Interop.Word
Imports Office = Microsoft.Office.Core

Namespace WordEvnts
Public partial Class ThisDocument
private void void New_Startup(Object sender, System.EventArgs e)
End Sub
private void void New_Shutdown(Object sender, System.EventArgs e)
End Sub
#region VSTO Designer generated code
private void InternalStartup()
    Me.Startup += New System.EventHandler(ThisDocument_Startup)
    Me.Shutdown += New System.EventHandler(ThisDocument_Shutdown)
    Me.BeforeClose += New
        System.ComponentModel.CancelEventHandler(ThisDocument_BeforeClose)
End Sub
void void New_BeforeClose(Object sender, System.ComponentModel.CancelEventArgs e)
    Throw New Exception("The method or operation is not implemented.")
End Sub
#End Region
End Class
End Namespace
```

C#

```
using System;
using System.Data;
using System.Drawing;
```

```
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
namespace WordEvnts
{
    public partial class ThisDocument
    {
        private void ThisDocument_Startup(object sender, System.EventArgs e)
        {
        }
        private void ThisDocument_Shutdown(object sender, System.EventArgs e)
        {
        }
        #region VSTO Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// </summary>
        private void InternalStartup()
        {
            this.Startup += new System.EventHandler(ThisDocument_Startup);
            this.Shutdown += new System.EventHandler(ThisDocument_Shutdown);
            this.BeforeClose += new
                System.ComponentModel.CancelEventHandler(ThisDocument_BeforeClose);
        }
        void ThisDocument_BeforeClose(object sender, System.ComponentModel.CancelEventArgs
            e)
        {
            throw new Exception("The method or operation is not implemented.");
        }
        #endregion
    }
}
```

Listing 4-24: Event hookup code

It's fascinating to see how simple code can have unexpected results. Inside the `InternalStartup()` method, you can see the `Startup` and `Shutdown` events are wired to their respective handlers. Following this pattern, we add our own event `BeforeClose`. Visual Studio .NET IDE magic automatically creates the handler for us and even implements a bare-bones handler complete with a single line of code that throws an exception when the document begins its closing process. However, if you accept the defaults, run the application and close the document, the application will not behave as expected.

If you expected the application to stop because an exception was thrown, you would be pleasantly surprised. The exception has already occurred but the application simply did not stop. Instead, the document closes and the exception is simply ignored. This behavior is by design. When exceptions occur, the exceptions do not interfere with the document functionality. One of the application requirements for VSTO is that the document must continue to function even in the presence of a misbehaving .NET assembly.

Printing Documents from Word

Like Microsoft Excel, print functionality is simple. Consider code that prints the current document in a Word application, shown in Listing 4-25.

Visual Basic

```
private sub PrintDocument()  
    Me.PrintOut()  
End Sub
```

C#

```
private void PrintDocument()  
{  
    this.PrintOut(  
        ref missing, ref missing, ref missing, ref missing, ref missing, ref missing,  
        ref missing, ref missing, ref missing, ref missing, ref missing, ref missing,  
        ref missing, ref missing, ref missing, ref missing, ref missing, ref missing);  
}
```

Listing 4-25

From Listing 4-25, printing functionality can't possibly get any easier than this. If you choose to call the method as presented in Listing 4-25, the application prints out the active document without any modifications, using the default settings.

The `PrintOut` method can print all or part of the document to include embedded shapes. The printing can occur asynchronously or the print can be customized to print completely before executing the subsequent lines of code. The `Print` routine can also impose page numbers on the printed document and instruct the printer to collate the document accordingly. For more fine-grained control, simply set the appropriate options of the `PrintOut` method.

Considerations for Word Development

The common denominator of all Microsoft Office applications is the Office COM server being targeted by the executing code. Visual Basic for Applications, unmanaged COM, and VSTO are simply different approaches to targeting the underlying server. When calls are funneled through to the Office COM server, a few things must occur before the request can be processed successfully. These initialization steps differ, depending on the programming approach used.

PIAs and RCWs for Office Word

For unmanaged COM and VBA, there is relatively little overhead. The requests are simply channeled to the COM server for processing and the results are returned. However, for the .NET approach, the CLR requires an adapter to intercept the managed call and translate it into a request that unmanaged code can understand.

The CLR uses a specially generated Runtime Callable Wrapper (RCW). This wrapper is an Interop Assembly and is used automatically for you if you have downloaded and installed the Primary Interop

Assemblies. If you choose not to download and install the Primary Interop Assemblies, one will be created for you automatically. Once the translation occurs, the call can be funneled on to the unmanaged COM server for processing.

It is evident that this overhead causes a performance hit each time a request is made from a managed application. This overhead cannot be avoided. Once the interoperation is complete or the application has terminated, the managed resources are cleaned up automatically by the CLR. However, the CLR does not have knowledge about the unmanaged resources created to facilitate communication between the managed and unmanaged word. These resources must be cleaned up, but the CLR is unable to do so. For that reason, the .NET code must explicitly make a call to release the unmanaged resources.

Assemblies and Deployment

Another area that requires some attention is the customization assembly that runs behind the document. One common question developers have centers around associating the .NET assemblies with the Word documents. Typically, every Word document targets its own assembly. However, the .NET assembly running behind a document is not set in stone. It is both easy and relatively painless to associate a document with a different assembly or to multiple assemblies. Listing 4-26 shows some code that associates a server document with a different .NET assembly.

Visual Basic

```
Dim servDoc as ServerDocument = new ServerDocument("file path")
servDoc.AppManifest.Dependency.AssemblyPath = "file path"
servDoc.Save()
servDoc.Close()
```

C#

```
ServerDocument servDoc = new ServerDocument("file path");
servDoc.AppManifest.Dependency.AssemblyPath = " file path ";
servDoc.Save();
servDoc.Close();
```

Listing 4-26: Assembly and document association

The code is fairly straightforward. First, an object of type `ServerDocument` is created. Next, the dependent assembly is changed to point to a new assembly given by the file path. The `ServerDocument` is then saved and closed. From this point onward, the `ServerDocument` fires the code in the new assembly. That sort of approach comes in handy in some deployment scenarios where different documents must be based on the same assembly.

Custom Actions Panes

There are several controls that aren't covered in this book. Most of these controls are intuitive to use and require little more than a passing knowledge of their existence. However, some controls such as the task pane control require a little more thought. Microsoft Office 2003 first introduced the smart document feature. This feature allows developers to create custom document task panes that are associated with Office applications. This smart document feature has been refined to provide the customized task pane in the most recent version of VSTO.

Most developers and users are familiar with task panes, so we spare the details for another section. However, the internal plumbing deserves more attention. The task pane is a generic Windows forms control container that is able to host other controls. You may add controls to the task pane directly through code. However, the task pane is only visible in a running application if it contains controls. These controls also are not printed with the document when a print operation is requested. This makes it an excellent strategy for documents that must be generated in a print friendly manner.

The actions pane is another generic control container that sits inside the task pane control. The actions pane can be used to host other containers and controls. These controls may be wired to events as well. There are some disadvantages of actions panes that must be noted. Actions panes cannot be hidden through code and cannot be repositioned either. The reason for this is that the actions pane is embedded in the task pane. One workaround is to simply reposition the task pane.

Summary

The VSTO Word API is surprisingly powerful yet simple to use. The available functionality spans the entire range of functionality available with Microsoft Office today. Some of the functionality is exposed through a few key objects such as the `ThisApplication` object. Objects such as these provide global hooks where calling code can conveniently access internal objects, data, and the document functionality of the Microsoft Office application.

The `Range` object is the key to most of that functionality. The `Range` object wraps formatting and addressing functionality behind a simple interface. In addition to the in-memory object, the `Range` object also presents itself as a bona fide user interface control that is fully accessible from the `OfficeCodeBehind`.

Another potent control makes an appearance in VSTO. The `backgroundWorker` control is designed to be used with Windows forms and User interface–related code. The control brings the power of threading to the design surface in a way that has never been done before. The control is not without issues. Unfortunately, the control cannot be used in applications such as windows services. Also, there are certain restrictions on controls that are accessed from the `backgroundWorker` control.

.NET and VSTO have certainly made an effort to bring threading to the masses through the `backgroundWorker` control and the associated threading classes. It is still not a move that promotes a warm and fuzzy feeling to seasoned developers, since these implementations trivialize the complexity of these approaches. Threading issues are extremely difficult to discover and often do not surface until it is too late. Still, if you care to learn the ropes first and have the stomach for the steep learning curve, such an approach can realize a significant performance boost to your applications. However, you must proceed with every bit of caution and afford the technique the due diligence it so rightly deserves.

This chapter also covered tables in some detail. Tables are made up of rows and columns as well as functionality and formatting. Formatting can be applied through table styles. When adding styles programmatically, the table style overwrites any custom styles that are already present. If you need to maintain custom styles, add the supported style to the table object first, and then add your custom style.

Another interesting control worth noting is the actions pane. Controls embedded in the `ActionsPane` control do not print by default. Normally, a print operation prints the document and the embedded controls they may contain. The only way to prevent these embedded objects from printing is to set their visible property to `false`. The `ActionsPane` control neatly sidesteps this issue.

5

Outlook Automation

Four hundred million people use Microsoft Office today. It is fair to assume that most of these users do some sort of email task at some point during the day, because Microsoft Outlook is the most popular product of the Microsoft Office suite. Since Microsoft Outlook is the dominant email application for corporations, Microsoft is fairly keen on maintaining that large customer base. One key way to attract and keep customers is to improve the quality and availability of applications built on Microsoft Office technology.

VSTO brings this technology home by providing Outlook add-ins that allows applications to hook into the Microsoft Office Outlook platform. Using add-ins, it is possible to customize or improve the appearance and functionality of the standard toolbar in an Outlook application.

This chapter will focus on programming tasks associated with Microsoft Outlook. The approach is somewhat different because Outlook automation is achieved through the use of add-ins. The code that you write to implement programming requirements is compiled into a .NET managed assembly. This assembly is loaded as a managed add-in into Microsoft Outlook when the Outlook application first initializes. From this point, your code is executed in response to Outlook events. Add-ins created this way are global in nature and are used continually unless unloaded by the user or through an unhandled exception.

Notice that this is different from an Excel spreadsheet application. Excel applications are based on a set of templates that provide functionality to the Excel object, and the templates are not global in scope. In any case, the material here will focus on developing add-ins. In addition, the code will show you how to target the event model of Outlook because add-ins typically respond to events raised by the Outlook Object Model (OOM). For instance, your code may take some action based on new email notifications.

A large majority of the chapter is dedicated to manipulating emails messages, updating address books, managing contacts, scheduling meetings, and implementing calendar functionality. These approaches are all presented from a data-driven perspective. For instance, a new email in the Inbox triggers a new email event notification. The event fires code in your custom add-in that examines the data present in the email. Based on the contents of that data, some action is performed.

We will also examine the key objects responsible for driving the user interface functionality. It's important to understand the relationships among these intrinsic objects if you intend to customize the user interface. One key object in the Outlook namespace is the MAPI object. We reserve a more detailed examination of Messaging Application Programming Interface (MAPI) for later but for now, you should note that MAPI is the glue that holds the Outlook application together.

Finally, we spend some time examining the updated security model. The improvement to the security model helps mitigate the risk of malware that may be spread through Outlook emails. Email integration and connectivity has also been enhanced. Code can now take advantage of new email objects and events that improve the programming experience for Outlook developers.

Configuring VSTO Outlook

VSTO ships with a Microsoft Outlook add-in loader component that allows managed code to use managed as well as unmanaged add-ins. To begin writing code that targets Microsoft Outlook, you will first need to configure Visual Studio. The installation instructions are provided in Chapter 1. From this point, the Microsoft Outlook add-in should appear in the Visual Studio IDE task pane, as shown in Figure 5-1.

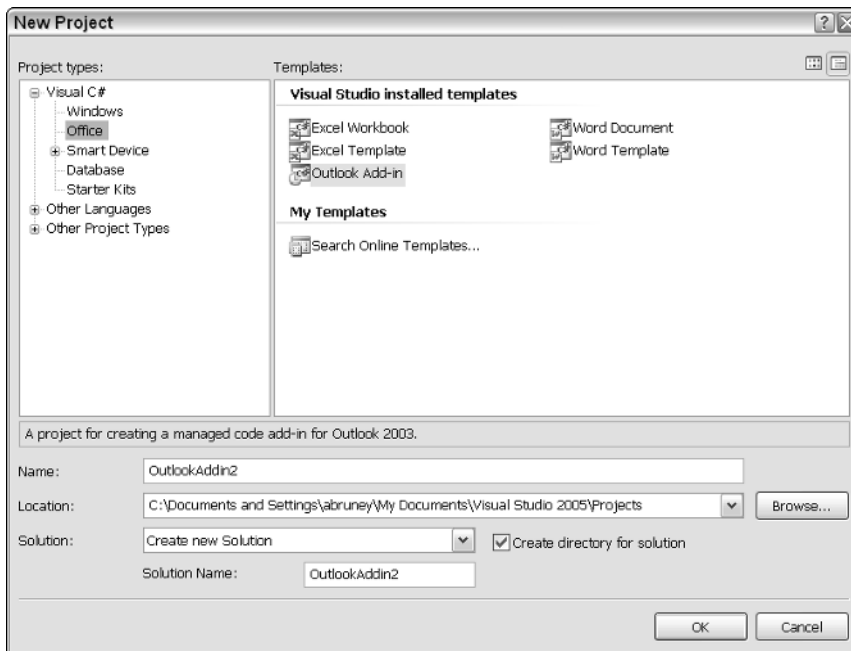


Figure 5-1

You should notice that there are no template options for Microsoft Outlook. The reason for this absence is that Microsoft Outlook is an email application. On the other hand, template-based projects such as Microsoft Excel and Microsoft Word are applications that are centered on document creation and manipulation. Templates are provided in these environments to ease the document creation process.

If you select the Outlook add-in option and create a new project, the Visual Studio IDE opens up to the `OfficeCodeBehind` file, as shown in Figure 5-2.

Notice also that Visual Studio has added a setup project in the property pages. This project contains all the dependent assemblies that your application will require to function correctly. Also, a new primary output subfolder is available, with a few configurable properties. These properties allow the developer to customize the installer package to ease the burden of installation and deployment. For instance, the new option allows you to detect new installation versions automatically through the `detectnew installation` property. To see the list of available configuration properties, right-click the Outlook setup node in the property pages hierarchy tree and choose Properties.

During application development, it is often necessary to purge the Outlook add-ins that are loaded because these add-ins can significantly slow down the Outlook initialization process.

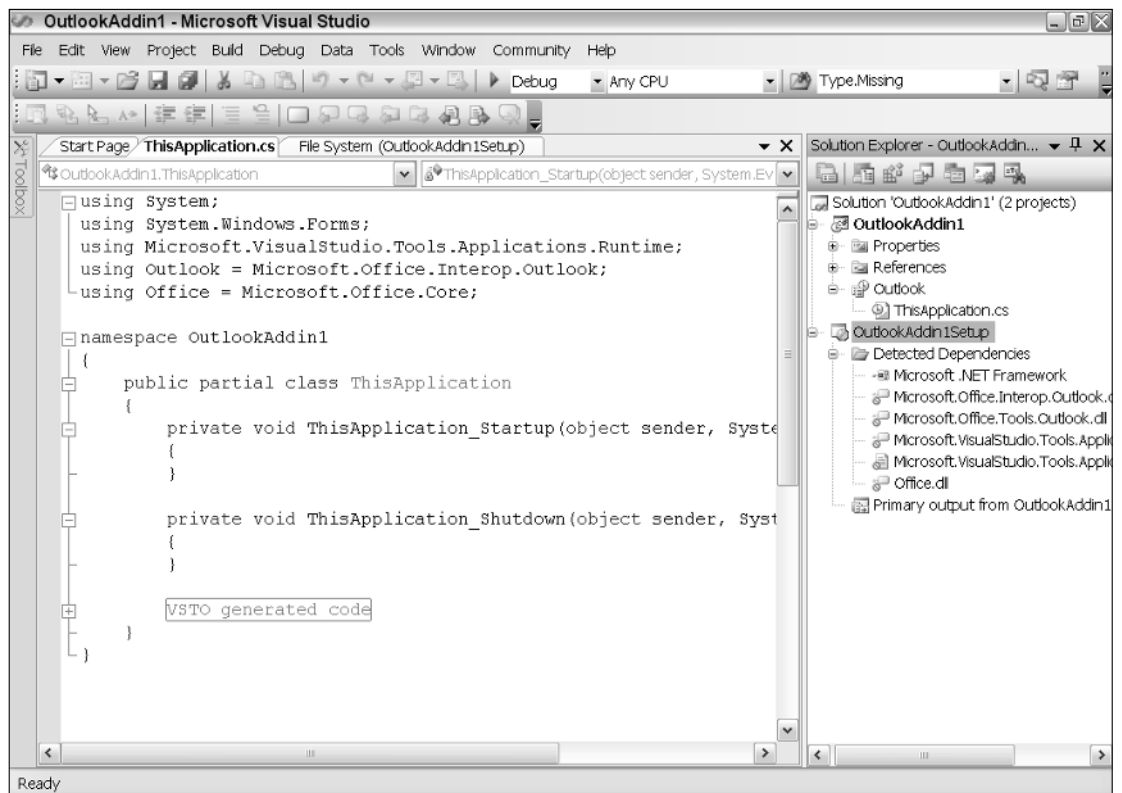


Figure 5-2

Press F5 to run the application without making any changes to `OfficeCodeBehind`. If you haven't configured Microsoft Outlook, the configuration appears. You may use it to configure or add the address to the Microsoft Exchange server. If you do not know the address of the Microsoft Exchange server, ask your network administrator. If Microsoft Exchange server is already configured, simply accept the defaults and proceed.

After configuration is complete, Microsoft Outlook 2003 will be launched, as shown in Figure 5-3.

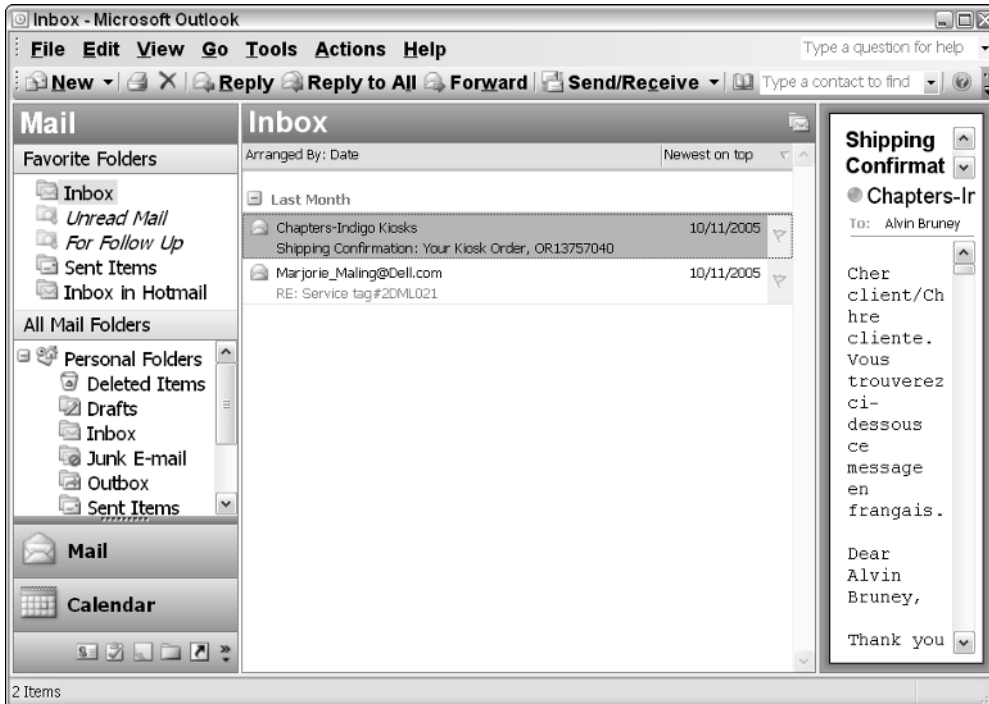


Figure 5-3

Microsoft Outlook defines certain internal objects that represent user interfaces in Outlook that you need to be familiar with in order to build robust, performant add-ins. For instance, the `toolbar` collection displays the standard toolbar buttons that are part of every component in the Microsoft Office VSTO suite. You can use the code you saw in Chapter 2 and Chapter 4 to customize the toolbar for Microsoft Outlook. Although Outlook differs from Excel and Word applications, the approach to programming the toolbars and the menus in VSTO remains consistent across each component.

Another example of an important object is the `Advanced Find` dialog that is used for searching. The dialog is generated from an `Outlook Inspector` object. Most toolbar functionality events will launch the `Inspector` object that, in turn, generates an `Inspector` dialog. A more detailed discussion of `Inspector` objects follows in the next section.

When Microsoft Outlook first launches as in Figure 5-4, an `Explorer` object creates the initial Outlook view. The view may be divided into one or more panes, depending on the end-user configuration. `Explorer` objects are also generated when the user selects `File` → `New` from the Outlook menu bar.

Key Outlook Objects

The Outlook API is spread wide. However, most of the functionality is contained within a few central objects. Once you are familiar with these objects, it's easy to customize Outlook. Before we dive into the objects, it is helpful to gain some perspective on the underlying protocol that Microsoft Outlook uses for email communication. An appreciation of the technology can go a long way when new features, such as extended email management or custom user forms, need to be implemented on top of the Microsoft Outlook platform. Custom user forms are presented later in the chapter.

What Is MAPI?

Messaging Application Programming Interface (MAPI) is not an object. Rather, it is a protocol that allows different Windows-based applications to interact with email applications. MAPI is platform neutral and can facilitate data exchange across multiple systems. MAPI was originally designed and built by Microsoft and has gone through several revisions. There are other protocols for messaging that are equally popular, such as Simple Mail Transfer Protocol (SMTP) and Internet Message Access Protocol (IMAP). However, these protocols are not covered in this book.

MAPI brings certain advantages to the table. For instance, MAPI allows you to manipulate views. MAPI allows you to add custom features to the options menu in Outlook. MAPI also provides performance improvements as compared to implementations that run without MAPI. This is certainly not an exhaustive list, but it is sufficient to provide you with a general idea of MAPI's effectiveness as a protocol.

The data contained in Outlook is organized into Outlook folders. MAPI provides access to the data stored in these folder structures through the `Application` object. The actual data for Microsoft Outlook is stored in files with a `.pst` extension for accounts configured without Microsoft Exchange. Accounts configured with Microsoft Exchange store files with an `.ost` extension. MAPI simply provides a way to access this data through a valid session. Listing 5-1 provides an example.

Visual Basic

```
If Me.Session.Folders.Count > 0 Then
    Dim folder As Outlook.MAPIFolder = Me.Session.Folders(1)
    If Not folder Is Nothing Then
        MessageBox.Show("The " & folder.Name & " folder contains " &
folder.Items.Count & " items.")
    End If
End If
```

C#

```
if (this.Session.Folders.Count > 0)
{
    Outlook.MAPIFolder folder = this.Session.Folders[1];
    {
        MessageBox.Show("The " + folder.Name + " folder contains " +
folder.Items.Count + " items.");
    }
}
```

Listing 5-1: MAPI folder manipulation

The code in Listing 5-1 obtains a reference to the current MAPI session. A valid MAPI session is created when Microsoft Outlook first initializes. A MAPI session is required before calling code can retrieve data from the MAPI subsystem. The code then extracts the folder name and the number of items available in that folder for display. Notice that the architecture is designed to be seamless; calling code is not burdened with the implementation details of the underlying subsystem.

It's important to point out a few sticky issues with this code. Notice that the code uses an index as opposed to an enumeration to retrieve the folder. The reason for this is that folders that do not exist result in an array bounds access exception. Listing 5-2 shows the better approach because it uses a named constant instead of an index. The approach guarantees that the folder being accessed is available without the potential for exceptions.

When folders are constructed, Outlook imposes a type on the contents of the folder, so calling code can assume that all the objects in one folder are necessarily of the same type. It is not possible to add two objects of different types in one folder. With that assumption, calling code may avoid an unnecessary casting step while iterating the contents of the folder. For instance, the default type of the folder in Listing 5-1 is given by `folder.DefaultItemType`.

Application Object

The `Application` object is at the root of the Outlook object hierarchy. Immediately underneath this root object, you will find the `Explorers` and `Inspectors` objects. These objects are described in more detail in the next section.

The `Application` object in Microsoft Outlook houses an instance of the executing application. You may use this object to gain access to deeper levels of the Outlook Object Model (OOM). The `Application` object also contains two key events. The `onstart` event is used to initialize any application variables that the application may use. The `onshutdown` event is used to clean up any application variables that may have been created by the application, such as database connections.

Explorer Object

The `Explorer` object represents the folder hierarchy. The `Explorer` object is roughly equivalent to Windows Explorer, whose views map out the folder hierarchy in Microsoft Windows. Although the `Explorer` window is associated with the default view of Microsoft Outlook, an instance of Microsoft Outlook can be fired without opening an `Explorer` window. For instance, hand-held devices that connect to Outlook must automate Outlook but do not necessarily invoke an instance of `Explorer`.

Figure 5-5, shows an `Explorer` window with its default views. The `Explorer` window can only be created from an existing `Explorer` window by selecting a folder and choosing `New` from the `File` menu. It's important to understand this, especially if calling code must interact with the methods, properties, and events of the `Explorer` object.

Listing 5-2 shows an example of some `Explorer` code.

Visual Basic

```
Private Sub ThisApplication_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    Dim folder As Outlook.MAPIFolder
    folder =
Me.Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderContacts)
    Dim explorerWindow As Outlook.Explorer = Me.Explorers.Add(folder,
Outlook.OlFolderDisplayMode.olFolderDisplayNormal)
    explorerWindow.Activate()
End Sub
```

C#

```
private void ThisApplication_Startup(object sender, System.EventArgs e)
{
    Outlook.MAPIFolder folder =
this.Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderContacts);
    Outlook.Explorer explorerWindow = this.Explorers.Add(folder,
Outlook.OlFolderDisplayMode.olFolderDisplayNormal);
    explorerWindow.Activate();
}
```

Listing 5-2: Explorer object manipulation

The code in Listing 5-2 first obtains a reference to the default `Contacts` folder in Microsoft Outlook. That folder is then passed as an argument to the `Add` method of the `Explorers` collections object. The `Add` method creates an `Explorer` object based on the folder type specified. With the call to `Activate`, the window is displayed onscreen, as you can see back in Figure 5-3.

Assuming that you compile the code, you may notice that the `Activate` method fires a compiler warning for ambiguity in C#. Visual Basic is able to resolve the conflict behind the scenes. You may safely ignore this warning in C#. This is another advantage for using Visual Basic over C#.

When you execute the code in Listing 5-2, you should notice that an `Explorer` window fires that opens immediately to a `Contact` window view, as shown in Figure 5-4. The code is handy and very flexible so that any one of the 16 folders may be passed in as an argument to have the `Explorer` window load the default view.

Inspector Object

The `Inspector` object represents the selected item in the `Explorer` window. You may use that object to inspect items in the `Explorer`. For instance, if you double-click on the `Inbox` folder that sits in the `Explorer` window, an `Inspector` window will be launched containing information about the selected item.

Consider this piece of code.

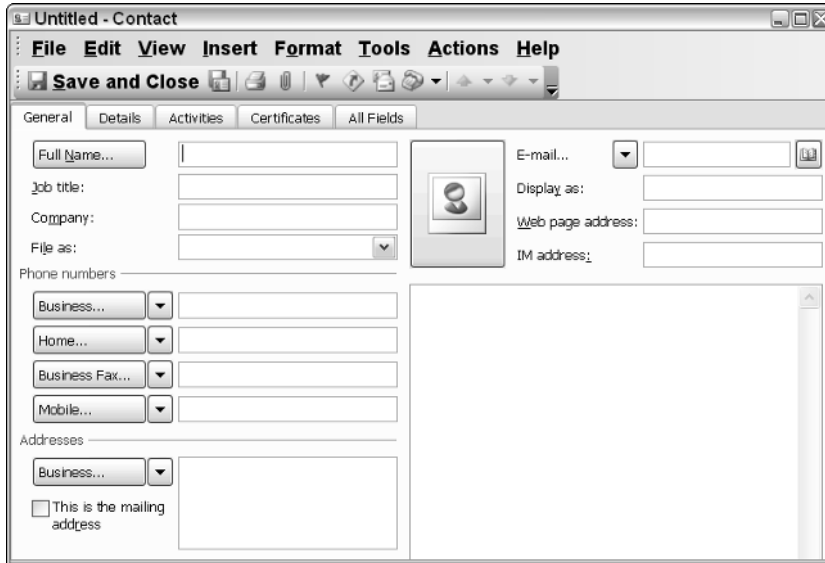


Figure 5-4

Visual Basic

```

Dim ol As Outlook.Application = New
Microsoft.Office.Interop.Outlook.Application()
Dim NewMail As Outlook.MailItem
NewMail =
ol.CreateItem(Microsoft.Office.Interop.Outlook.OlItemType.olMailItem)
Dim inspectorWindow As Outlook.Inspector
inspectorWindow = Me.Inspector.Add(NewMail)
inspectorWindow.Activate()

```

C#

```

Outlook.Application ol = new
Microsoft.Office.Interop.Outlook.Application();
Outlook.MailItem NewMail =
ol.CreateItem(Microsoft.Office.Interop.Outlook.OlItemType.olMailItem) as
Outlook.MailItem;
Outlook.Inspector inspectorWindow = this.Inspector.Add(NewMail);
inspectorWindow.Activate();

```

Listing 5-3: Inspector object manipulation

In Listing 5-3, a reference to the Outlook application is retrieved and stored in variable `ol`. A new mail object is created and added using the `ol` variable. Next, a new Inspector window is created containing the new mail object and displayed to screen. The new mail is displayed empty by default because the code in Listing 5-3 does not set any properties on the new object.

It's important to know that both the `Inspector` and `Explorer` objects are housed in parent collections. A collection is necessary because there may be zero or more objects of each type at any given point in application execution. In fact, a count of these respective collections can usually indicate the state the Microsoft Outlook application is in for cleanup and initialization purposes. Also, a collection container makes it easy for events to be wired to the objects as a whole.

Outlook exposes the `ActiveExplorer()` and `ActiveInspector()` methods that return an instance of the active object at a given point in time. The active object is the window that contains the current focus. In Microsoft Windows, only one window can have the current focus at any point in time.

Email Integration

Every day, billions of pieces of email move across the World Wide Web. For corporate entities running on Windows platforms, the email application of choice is Microsoft Outlook. Microsoft Outlook was specifically designed to manage emails. The most common tasks centered on email management are creating, sending, searching, sorting, and deleting emails. You should expect to provide at least that basic layer of functionality into any email application that you build. Your new functionality will be built and incorporated into Outlook as an add-in.

If Microsoft Outlook encounters an error or exception in your add-in that is not handled, Outlook will try to disable your add-in. If you notice that your application has mysteriously stopped working, you should first determine whether or not your add-in is enabled. If it is disabled, then you will need to review your code for unhandled exceptions.

VSTO exposes a rich interface for email management. VSTO allows emails to be created, sent, searched, sorted, and deleted. In addition to the basic manipulation, it is also possible to perform more functionality with emails. In this section, we take a look at the machinery that Microsoft Outlook exposes to enable code to manage emails.

Creating Emails

The process of creating emails is straightforward enough. Since every email is essentially an object of type `MailItem`, you simply need to create an object of the required type. Here is some code.

Visual Basic

```
Public Sub CreateAndSendEmail()  
    Dim ol As Outlook.Application = New Microsoft.Office.Interop.Outlook.Application()  
    Dim NewMail As Outlook.MailItem  
    NewMail =  
    ol.CreateItem(Microsoft.Office.Interop.Outlook.OlItemType.olMailItem)  
    NewMail.Subject = "This is my email subject"  
    NewMail.To = "vapordan@hotmail.com"
```



```
NewMail.Importance =  
Microsoft.Office.Interop.Outlook.OlImportance.olImportanceNormal  
NewMail.HTMLBody = "This <u>is</u> truly <b>amazing</b>"  
NewMail.BodyFormat =  
Microsoft.Office.Interop.Outlook.OlBodyFormat.olFormatHTML  
NewMail.Send() End  
Sub
```

C#

```
Public void CreateAndSendEmail()  
{  
    Outlook.Application ol = new Microsoft.Office.Interop.Outlook.Application();  
        Outlook.MailItem NewMail =  
    ol.CreateItem(Microsoft.Office.Interop.Outlook.OlItemType.olMailItem) as  
    Outlook.MailItem;  
        NewMail.Subject = "This is my email subject";  
        NewMail.To = "vapordan@hotmail.com";  
        NewMail.Importance =  
    Microsoft.Office.Interop.Outlook.OlImportance.olImportanceNormal;  
        NewMail.HTMLBody = "This <u>is</u> truly <b>amazing</b>";  
        NewMail.BodyFormat =  
    Microsoft.Office.Interop.Outlook.OlBodyFormat.olFormatHTML;  
        NewMail.Send();  
}
```

Listing 5-4: Code to create an email

The code in Listing 5-4 creates an object of type `MailItem`. Once the object is created, the code sets the appropriate properties, such as the `To` and `Subject` fields. Notice that the email object supports formatting types so that it is possible to send an email with the body in HTML. For email readers that support it, such as Microsoft Outlook express, the importance flag is set on the mail item. If the mail you are sending is critical, consider changing the importance to a more appropriate one so that the mail will receive due attention.

If you compare Listing 5-4 and Listing 5-3, there are some key differences. Listing 5-3 creates a new piece of mail and displays it on the screen. The user is responsible for filling in the information such as body, subject, and `To` field. Once this is done, the user can click the `Send` button to send the mail. By contrast, Listing 5-5 creates a new piece of mail and fills in the information. Then, the code attempts to send the piece of mail. At this point, Outlook security will prevent this action. A security dialog will inform the user that a piece of mail is about to be sent on his or her behalf. The user has the option to prevent the mail from being sent.

We reserve a more detailed analysis of this process for later in the chapter. For now, we note that the security module in Outlook has been revised to prevent abuse when code executes on behalf of a user. We also present strategies to override this behavior later in the chapter.

Finally, you should note that the VB portion of the code in Listing 5-4 is not strictly equivalent to the C# code. The C# code uses the `as` operator as a safe cast to prevent exceptions when casting is not possible. The VB code would need to use the `TryCast` method to ensure that the code does not throw an exception when the cast fails.

Valid emails must contain a To field and a From field. The item in the To field must resolve correctly for the email to be delivered successfully. Due to security concerns, there is no programmatic way to set the From field.

Manipulating Email Messages

Today, there are hundreds of millions of active email inboxes. With such a large number of email inboxes, it is easy to see why application software that manipulates emails is common. These software applications all contain the same driving logic; find the inbox and iterate its contents looking for a specific item.

With this driving logic in mind, you can build your own custom add-in to provide some rudimentary processing. Consider this application requirement that monitors for emails with attachments. Once the application finds an email that contains an attachment, some action is taken. Consider the code in Listing 5-5.

Visual Basic

```
Private Sub BlockAttachments()
    Dim inbox As Outlook.MAPIFolder =
    Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderInbox)

    If inbox.Items.Count > 0 Then
        Dim virus As Object
        For Each virus In inbox.Items
            If Not virus Is Nothing Then
                If Not virus.Attachments Is Nothing Then
                    Dim forbidden As System.Collections.ArrayList
                    forbidden = New
                    System.Collections.ArrayList(CType(virus.Attachments.Count, Integer))
                    Dim attachObject As Outlook.Attachment
                    For Each attachObject In virus.Attachments
                        Dim obj As Object = attachObject.MAPIOBJECT
                        If Not obj Is Nothing Then
                            attachObject.Delete()
                            forbidden.Add(attachObject.DisplayName)
                        End If
                    Next
                    If forbidden.Count > 0 Then
                        MessageBox.Show("Some emails were blocked because they
contain attachments")
                    End If
                End If
            End If
        Next
    End If
End Sub
```

C#

```
private void BlockAttachments()
{
    Outlook.MAPIFolder inbox =
    Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderInbox);

    if (inbox.Items.Count > 0)
```

```
    {
        foreach (object item in inbox.Items)
        {
            Outlook.MailItem virus = item as Outlook.MailItem;
            if (virus != null)
            {
                if (virus.Attachments != null)
                {
                    System.Collections.ArrayList forbidden = new
System.Collections.ArrayList(virus.Attachments.Count);
                    foreach (Outlook.Attachment attachObject in
virus.Attachments)
                        {
                            object obj = attachObject.MAPIOBJECT;
                            if (obj != null)
                            {
                                attachObject.Delete();
                                forbidden.Add(attachObject.DisplayName);
                            }
                        }
                    if (forbidden.Count > 0)
                    {
                        MessageBox.Show("Some emails were blocked because
they contain attachments");
                    }
                }
            }
        }
    }
}
```

Listing 5-5: Iterating the inbox folder

The code in Listing 5-5 iterates the items collection to find out if there are items of type `Outlook.MailItem`. As soon as valid items are found, the code iterates the attachments collection. The attachment collections hold attachments for each piece of mail. The code promptly deletes these attachments and stores the name of the deleted item for use later.

As a matter of courtesy, the application informs the user of the violation of corporate policy. Minus a few bells and whistles, this is a common approach for most corporate email filters used to scan email inboxes for suspect mail. As Listing 5-5 demonstrated, it is easy to build an application once you understand the basics.

The VB portion of the code in Listing 5-5 can be optimized to use a `TryCast` method call for the loop iteration code. This approach allows the developer to take advantage of late binding while setting the `Option strict` flag.

Appointments and Meetings

Appointments are an integral part of the Outlook model. Figure 5-5 shows an Outlook appointment. Creating this appointment in code is easy enough. First, you need to be aware that an appointment is represented in Outlook by an `AppointmentItem` object. Essentially, your code will instantiate a new `AppointmentItem` object and set up some interesting properties. Once this is complete, your appointment will show up in Outlook. Figure 5-5 shows an Outlook appointment.

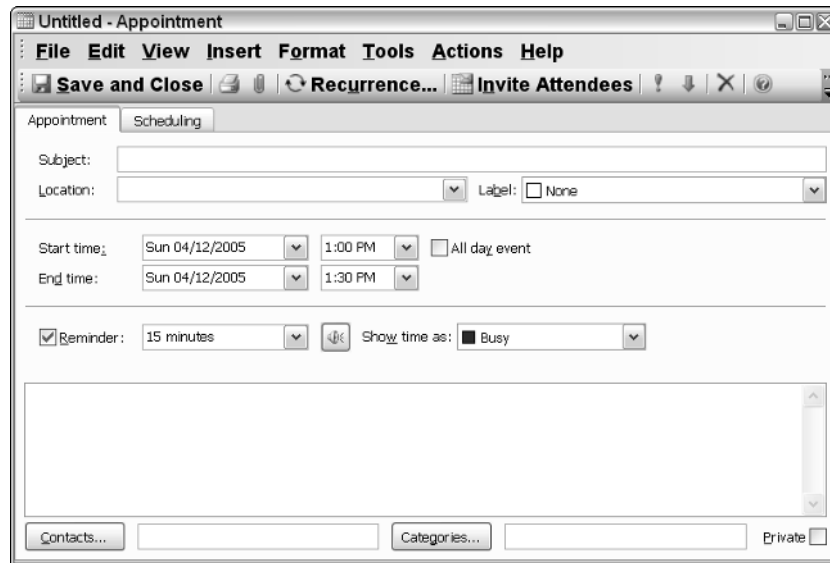


Figure 5-5

The code in the next few sections shows how to manipulate appointments in Microsoft Outlook applications.

Creating and Scheduling Appointments

With Microsoft Outlook, there is a great deal of flexibility when creating and manipulating appointments. For instance, appointments may be created locally or on a server with a shared calendar, either by yourself or someone else with the appropriate permissions. In every case, the create routine is identical assuming that the executing code has permissions to access shared resources.

Let's examine the code to create an appointment. We assume that the create method executes on a local drive and the executing code has the appropriate permissions. Once you grasp this basic concept, it's easy to add bells and whistles.

Visual Basic

```
Imports System
Imports System.Windows.Forms
Imports Microsoft.VisualStudio.Tools.Applications.Runtime
Imports Outlook = Microsoft.Office.Interop.Outlook
Imports Office = Microsoft.Office.Core

Namespace OutlookAddin1
    Public partial Class ThisApplication
        Private Sub New_Startup(ByVal sender As Object, ByVal e As
System.EventArgs) handles Me.Startup
            'creat the outlook appointment
            Dim appItem As Outlook.Application
            appItem = New Microsoft.Office.Interop.Outlook.Application()
            Dim appointment As Outlook.AppointmentItem =
appItem.CreateItem(Outlook.OlItemType.olAppointmentItem)

            'set up some custom features
            appointment.Subject = "This is my new appointment subject"
            appointment.Body = "This is my new appointment body"
            appointment.AllDayEvent = True
            appointment.BusyStatus = Outlook.OlBusyStatus.olBusy
            appointment.Start = DateTime.Now
            appointment.End = DateTime.Now.AddMinutes(30)
            appointment.ReminderSet = True
            appointment.Importance =
Microsoft.Office.Interop.Outlook.OlImportance.olImportanceHigh

            'activate the appointment
            appointment.Save()
        End Sub

        Private Sub New_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Shutdown
        End Sub
    End Class
End Namespace
```

C#

```
using System;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;
using Office = Microsoft.Office.Core;

namespace OutlookAddin1
{
    public partial class ThisApplication
    {
        private void ThisApplication_Startup(object sender, System.EventArgs e)
        {
            //creat the outlook appointment
            Outlook.Application appItem = new
Microsoft.Office.Interop.Outlook.Application();
```

```

        Outlook.AppointmentItem appointment =
appItem.CreateItem(Outlook.OlItemType.olAppointmentItem) as
Outlook.AppointmentItem;

        //set up some custom features
appointment.Subject = "This is my new appointment subject";
appointment.Body = "This is my new appointment body";
appointment.AllDayEvent = true;
appointment.BusyStatus = Outlook.OlBusyStatus.olBusy;
appointment.Start = DateTime.Now;
appointment.End = DateTime.Now.AddMinutes(30);
appointment.ReminderSet = true;
appointment.Importance =
Microsoft.Office.Interop.Outlook.OlImportance.olImportanceHigh;

        //activate
appointment.Save();
    }

private void ThisApplication_Shutdown(object sender, System.EventArgs e)
{
}

#region VSTO generated code

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InternalStartup()
{
    this.Startup += new System.EventHandler(ThisApplication_Startup);
    this.Shutdown += new System.EventHandler(ThisApplication_Shutdown);
}

#endregion
}
}

```

Listing 5-6: Creating appointments in code

The code first creates an Outlook application object and then uses this application object to create an appointment item object. The appointment item object is then customized. For instance, an appointment body and subject are added as well as some other properties that are normally associated with appointments.

You should notice the direct cast of the appointment item in the C# snippet (`TryCast` in VB). The cast is necessary because the `createitem` call returns an object instance. Finally, to activate the appointment, the code must explicitly call the save method so that the appointment can be made visible. Figure 5-6 shows the code in action.



Figure 5-6

Notice that you can click the appointment in the Outlook calendar to display its Inspector object window containing the body of the message. You can set every item in the dialog through code as well as some other properties that are not visible.

Let's examine property setting in some more detail because it may not be as trivial as it sounds. Consider the situation where iterating code must set some property on the current object. If the type of the object is not known, the code cannot simply set the property because an exception will be thrown. However, there is a more elegant way to interrogate the object to determine whether or not it supports a specific property. Consider the code in Listing 5-7.

Visual Basic

```

If Me.Inspectors.Count > 0 Then
    Dim note As Object = Me.Inspectors(1).CurrentItem
    Dim type As Type = note.GetType()
    Dim propertyMapper As String
    propertyMapper = type.InvokeMember("MessageClass", BindingFlags.Public
Or BindingFlags.GetField Or BindingFlags.GetProperty, Nothing, note,
Nothing).ToString()
    If propertyMapper = "IPM.Note" Then
        If Not note Is Nothing Then
            If Not note.IsConflict Then
                Dim subject As String = note.To
                MessageBox.Show("The email is addressed to " + subject)
            End If
            System.Runtime.InteropServices.Marshal.ReleaseComObject(note)
        End If
    End If
End If

```

```

C#
if (this.Inspectors.Count > 0)
{
    object item = this.Inspectors[1].CurrentItem;
    Type type = item.GetType();
    string propertyMapper = type.InvokeMember("MessageClass",
    BindingFlags.Public | BindingFlags.GetField |
BindingFlags.GetProperty, null, item, new object[] { }).ToString();
    if (propertyMapper == "IPM.Note")
    {
        if (!note.IsConflict)
        {
            Outlook.Conflicts conflicts = note.Conflicts;
            if (conflicts.Count < 1)
            {
                string subject = note.To;
                MessageBox.Show("The email is addressed to " +
subject);
            }

            System.Runtime.InteropServices.Marshal.ReleaseComObject(note);
        }
    }
}

```

Listing 5-7: Accessing properties dynamically

The code in Listing 5-7 shows how to retrieve properties dynamically. Before running the code you will need to import the `System.Runtime.InteropServices` and the `System.Reflection` namespaces. The code uses reflection to retrieve the properties during application execution. First, an index is used to obtain a reference to the first object in the `Inspectors` collection. When this code executes, there really is no way to retrieve the correct type. In fact, a call to `GetType()` will simply return an object of type `ComObject`. The only approach is to probe the object through reflection to find out if it contains the property that we are interested in. The `InvokeMember` method call does the object interrogation for us.

The VB line of code in Listing 5-7:

```

propertyMapper = type.InvokeMember("MessageClass",
BindingFlags.Public Or BindingFlags.GetField Or
BindingFlags.GetProperty, Nothing, note, Nothing).ToString()

```

can be optimized to:

```

propertyMapper = CallByName(note, "MessageClass",
CallType.Get).ToString()

```

The argument `MessageClass` indicates the type of object that we are currently handling. In this case, we register our interest in objects of type `MailItem`. If we find an object of the correct type, we briefly check the `IsConflict` property to determine if it is safe to proceed. Items that are in conflict often present problems for retrieving properties. In the case that the `IsConflict` property returns `true`, you will need to retrieve the objects in the `conflicts` collection to find the reason for the conflict.

Finally, we clean up by calling the `ReleaseComObject` on the current instance. You should note that the code is actually using ugly Interop code to perform the interrogation and retrieve an instance of the `object.NET` does not consistently handle the clean up of COM objects correctly. VSTO applications based on .NET 2.0 framework can still leak memory, especially when interfacing with COM objects. You can ensure a better behaved application by handling the clean up on the COM side through a call to `ReleaseComObject`. The actual implementation details of this cleanup is outside the scope of this text.

Let's discuss another issue before moving on. If the code is placed inside the startup event handler method, the code will not be called, since the Inspector window at that point contains no objects. The reason for this is that the startup code has not explicitly asked for any details of items in the Explorer.

One workaround is to subscribe to the `newinspector` event that fires when an Inspector window is called. You can place this code in the startup event handler. Then open an instance of Microsoft Outlook, or run the Outlook add-in from Inside Visual Studio .NET. Double-click an email item to view the details of the email. You will note from earlier reading that this action launches the `Inspector` object window with details on the selected email. At this point in time, our code is fired successfully.

Deleting Appointments

Deleting appointments in Outlook is a handy feature and pretty straightforward. The basic approach involves finding the item and deleting it. Listing 5-8 shows a full implementation.

Visual Basic

```
Imports System
Imports System.Windows.Forms
Imports Microsoft.VisualStudio.Tools.Applications.Runtime
Imports Outlook = Microsoft.Office.Interop.Outlook
Imports Office = Microsoft.Office.Core

Namespace DeleteAppointments
    Public partial Class ThisApplication
        Public Sub DeleteAppointments(ByVal item As String)
            Dim oCalendar As Outlook.MAPIFolder =
                Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderCalendar)
            Dim retVal As String = String.Empty

            Dim oResult As Outlook.AppointmentItem
            For Each oResult In oCalendar.Items
                If Not oResult Is Nothing Then
                    Dim type As Type = oResult.GetType()
                    retVal = CType(type.InvokeMember("Subject",
                        System.Reflection.BindingFlags.Instance Or System.Reflection.BindingFlags.Public Or
                        System.Reflection.BindingFlags.GetProperty, Nothing, oResult, Nothing, Nothing),
                        String)

                    If retVal = item Then
                        oResult.Delete()
                    End If
                End If
            Next
        End Sub

        Private Sub New_Startup(ByVal sender As Object, ByVal e As
            System.EventArgs)
```

```

        DeleteAppointments("This is my new appointment subject")
    End Sub

    Private Sub New_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs)
        End Sub

    End Class
End Namespace

```

C#

```

using System;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;
using Office = Microsoft.Office.Core;

namespace DeleteAppointments
{
    public partial class ThisApplication
    {
        public void DeleteAppointments(string item)
        {
            Outlook.MAPIFolder oCalendar =
Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderCalendar);
            string retVal = string.Empty;

            foreach (Outlook.AppointmentItem oResult in oCalendar.Items)
            {
                if (oResult != null)
                {
                    retVal = (string)oResult.GetType().InvokeMember("Subject",
System.Reflection.BindingFlags.Instance | System.Reflection.BindingFlags.Public |
System.Reflection.BindingFlags.GetProperty, null, oResult, null, null);
                    if (retVal == item)
                    {
                        oResult.Delete();
                    }
                }
            }
        }

        private void ThisApplication_Startup(object sender, System.EventArgs e)
        {
            DeleteAppointments("This is my new appointment subject");
        }

        private void ThisApplication_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}

```

Listing 5-8: Deleting appointments from outlook

Appointments are stored in the Microsoft Outlook calendar. The code in Listing 5-8 iterates the `appointments` collection searching for an item where the subject matches the keyword. Notice that the subject property is examined using reflection through the use of the `InvokeMember` method call. In this specific scenario, the reflection approach is not strictly necessary because the property is immediately accessible through the `oResult` object. For instance, the `InvokeMember` line of code can be replaced with `oResult.Subject`. However, this approach has been provided as an alternative. You should consider using it when properties are not directly exposed through the object. Bear in mind also that reflection is more expensive to the runtime than direct property access.

One issue you need to be aware of is that the delete call can fail. For instance, if the specific item is in use, it cannot be deleted and a runtime exception will occur usually with a cryptic error message. To that end, it's usually proper programming technique to wrap the delete call in exception-handling code. Notice that if an exception is thrown and is unhandled, your add-in will crash, possibly leaving Outlook and other add-ins in an unknown state.

VSTO does load every managed add-in in its own application domain to reduce the vulnerability of misbehaving add-ins on the health of the Outlook application in general. However, these niceties do not extend to add-ins that are built with unmanaged code. The end result is that Outlook is still vulnerable to add-in crashes.

Creating Meetings

An Outlook meeting is a type of appointment. Meetings are also associated with the Microsoft Outlook calendar implicitly. The code that creates meetings can be constructed from the code that creates an appointment with a minor modification. Here is an example of the required code change:

```
appointment.MeetingStatus =  
Microsoft.Office.Interop.Outlook.OlMeetingStatus.olMeeting
```

Additionally, meetings typically involve more than one participant, so some more code needs to be added to ensure that the meeting is set up correctly. In this case, some method of notification must occur to inform the recipients of the meeting. Fortunately, this is already built in to Outlook. Using the same code in Listing 5-1, first remove the `appointment.Save()` method line of code and then replace it with the code in Listing 5-9.

Visual Basic

```
Dim attendees As Outlook.Recipients  
Attendees = appointment.Recipients  
'add meeting participants  
Dim i As Integer  
For i = 0 To 10 - 1 Step i + 1  
    Dim developer As Outlook.Recipient = attendees.Add(i.ToString())  
    developer.Type = CType(Outlook.OlMeetingRecipientType.olRequired, Integer)  
Next  
'activate  
appointment.Send()
```

C#

```
Outlook.Recipients attendees = appointment.Recipients;  
//add meeting participants  
for(int i = 0; i < 10; i++)
```

```

{
    Outlook.Recipient developer = attendees.Add(i.ToString());
    developer.Type = (int) Outlook.OlMeetingRecipientType.olRequired;
}
//activate
appointment.Send();

```

Listing 5-9: Sending meeting requests

Continuing from Listing 5-8, the appointment is converted to a meeting by simply changing the meeting status to one of three available statuses. Then, a `for` loop is used to create some bogus attendees and the meeting request is sent out. However, if you attempt to run the code in Listing 5-9, things start to go terribly wrong.

Firstly, the bogus data created by the `for` loop is guaranteed to fail because these names do not resolve to valid email addresses in the Outlook address book. This results in an exception. As noted in previous chapters, the error message is quite confusing and borders on the side of useless. To fix that portion of code, you will simply need to replace the `i.ToString()` variable with valid email addresses.

Creating and Scheduling Outlook Items

Using the approach from the previous section, you should be able to modify the code to create contacts, tasks, distribution lists, reports, notes, and document item objects, since the items are all related in some way. Consider Listing 5-10, an example that creates a new Outlook note.

Visual Basic

```

Private Sub New_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    Dim ol As Outlook.Application
    ol = New Microsoft.Office.Interop.Outlook.Application()
    Dim NewNote As Outlook.NoteItem
    'Create a new note
    NewNote =
ol.CreateItem(Microsoft.Office.Interop.Outlook.OlItemType.olNoteItem)
    NewNote.Body = "This is my new note body"
    NewNote.Categories = "Memo"
    NewNote.Display()
End Sub

```

C#

```

using System;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;
using Office = Microsoft.Office.Core;

namespace Note
{
    public partial class ThisApplication
    {
        private void ThisApplication_Startup(object sender, System.EventArgs e)
        {

```

```
        Outlook.Application ol = new
Microsoft.Office.Interop.Outlook.Application();
        Outlook.NoteItem NewNote;
        //Create a new note
        NewNote =
ol.CreateItem(Microsoft.Office.Interop.Outlook.OlItemType.olNoteItem) as
Outlook.NoteItem;
        NewNote.Body = "This is my new note body";
        NewNote.Categories = "Memo";
        NewNote.Display(false);
    }
}
```

Listing 5-10: Note item creation

The code in Listing 5-10 creates a new Note object. Once the new Note is created, the body is set to some text and it is assigned to a category in Microsoft Outlook. Since the `categories` property is a string, it may be assigned any value. However, you must be disciplined enough to create categories that are useful. It's always a good idea to stick to the Outlook-provided categories rather than create your own. Finally, the new note is displayed to the screen.

If you haven't noticed by now, every time Outlook is opened, it runs the code contained in the add-in. Outlook is configured to load add-ins on startup, so it makes no difference if you remove the appointments from the calendar. The next time Outlook is opened, the appointments magically show up.

To prevent this behavior, simply instruct Outlook to avoid loading this add-in. Here is how to accomplish this. Select Tools ⇨ Options. The Options dialog will be displayed. Click the Other tab and click the Advanced Options button. The Advanced Options dialog box will be displayed. Click the Com-Add-ins button at the bottom of the dialog. The Add-in dialog box should now be displayed with a list of add-ins that Outlook will load on startup. Figure 5-7 shows the Add-in dialog box.

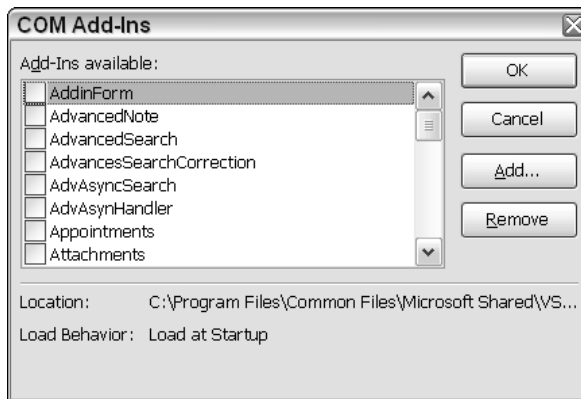


Figure 5-7

Select the add-in Note. The COM add-in path points to the directory where the project is located. For VSTO add-ins, the path points to the common VSTO runtime folder. Notice also that the load behavior is set to startup. If you select Remove, the add-in will not be available for loading. Instead, uncheck the add-in to disable the load. Disabled add-ins are still available to the Outlook application at a later point in time.

You can also add add-ins via this dialog box. However, only COM add-ins may be added in this way. Add-ins built using VSTO will fail with an appropriate message to the user. VSTO add-ins are loaded automatically by the VSTO installation application sometime before add-in execution.

To add COM add-ins, simply click Add, then navigate to the COM add-in on disk. At startup, Outlook will load the new add-in. You should be aware that each add-in that needs to be loaded will incur some overhead to Outlook during the initial load process. Also, you may notice that VSTO add-ins that are built from managed code are loaded using the COM add-in mechanism. The reason for this quirk is that Microsoft Outlook must be fooled into believing that it is loading a regular COM add-in. The actual implementation details of this process are outside the scope of the text.

Different types of customizations exist for different item types. For instance, you may be able to set the background color and category of each note item that you create. That type of functionality will vary based on the particular item created. Examine the exposed properties with IntelliSense to determine the range of customizations that are possible. Listing 5-11 is a more involved example of note manipulation.

Visual Basic

```
Private Sub ThisApplication_Startup(ByVal sender As Object, ByVal e As
System.EventArgs)
    Dim ol As Outlook.Application = New
Microsoft.Office.Interop.Outlook.Application()
    Dim NewNote As Outlook.NoteItem
    'Create a new note
    NewNote = ol.CreateItem(CType(as Outlook.NoteItem,
Microsoft.Office.Interop.Outlook.OlItemType.olNoteItem)
NewNote.Body = "Holiday 2morrow"
NewNote.Categories = "Business"
NewNote.Height = NewNote.Height \ 2
NewNote.Color = Outlook.OlNoteColor.olBlue
NewNote.Display(True)
End Sub
```

C#

```
private void ThisApplication_Startup(object sender, System.EventArgs e)
{
    Outlook.Application ol = new
Microsoft.Office.Interop.Outlook.Application();
    Outlook.NoteItem NewNote;
    //Create a new note
    NewNote =
ol.CreateItem(Microsoft.Office.Interop.Outlook.OlItemType.olNoteItem) as
Outlook.NoteItem;
    object modal = true;
    NewNote.Body = "Holiday 2morrow";
    NewNote.Categories = "Business";
```

```
NewNote.Height %= 2;  
NewNote.Color = Outlook.OlNoteColor.olBlue;  
NewNote.Display(modal);  
}
```

Listing 5-11: Advanced note manipulation

The example in Listing 5-11 creates a note and sets the body and category appropriately. The size and color of the note are adjusted accordingly and the note is made to display as a modal dialog. Although the `Categories` method accepts a string, you should try to use the predefined categories of Microsoft Outlook and resist the temptation to create your own. The predefined categories may be viewed by selecting the `Categories` option from the note context menu. Figure 5-8 shows the predefined categories located in Microsoft Outlook.

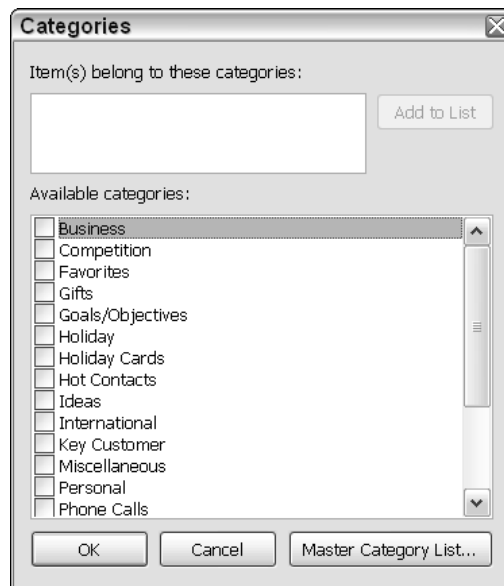


Figure 5-8

Depending on the item being created, some functionality may or may not be available. For instance, notes do not expose events so a note cannot be made to respond to a hyperlink click if that link appears inside the note document. However, you may be able to wire your handler code to the Outlook document hyperlink event method as a workaround.

Folder Manipulation

Recall that Microsoft Outlook organizes data into folders. Look back to Figure 5-3 to see some of the default folders that are available.

Outlook's design and architecture allows developers to create generic code to iterate these different folders. Only a few minor modifications are required as you navigate from one folder to another. Consider Listing 5-12.

Visual Basic

```
Dim itemsBox As Outlook.MAPIFolder =
Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderDeletedItems)
    If itemsBox.Items.Count > 0 Then
        Dim item As Object
        For Each item In itemsBox.Items
            Dim oItem As Outlook.MailItem = TryCast(item, Outlook.MailItem)
            If oItem IsNot Nothing Then
                'code to process items
            End If
        Next
    End If
```

C#

```
Outlook.MAPIFolder itemsBox =
Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderDeletedItems);
    if (itemsBox.Items.Count > 0)
    {
        foreach (object item in itemsBox.Items)
        {
            Outlook.MailItem oItem = item as Outlook.MailItem;
            if (oItem != null)
            {
                //code to process items
            }
        }
    }
}
```

Listing 5-12: Generic code to manipulate folders

In case you were wondering, the `Outlook.OlDefaultFolders` enumeration contains all the default Outlook folders that are available to calling code (16 default folders). Microsoft Outlook designates certain folders in the collection as the default folder. The designation occurs when Outlook detects the inbox and calendar in a particular folder.

Address Book Manipulation

You are probably already familiar with the Outlook address book shown in Figure 5-9. In fact, we have presented several examples of functionality that are implicitly hooked to the address book. For instance, Listing 5-12 showed that code may fail if an email address cannot be resolved.

The purpose of the address book is to store contact information. Microsoft Outlook exposes access to this list through the `AddressList` collection. The collection provides object-oriented access to its internals through a host of methods, properties, and events. The collection is also protected by Microsoft Outlook to keep harmful applications such as viruses from planting seeds of evil on your computer. We'll certainly talk more about the steps that Microsoft Outlook has taken to mitigate such risks, but for now we concentrate only on address manipulation.

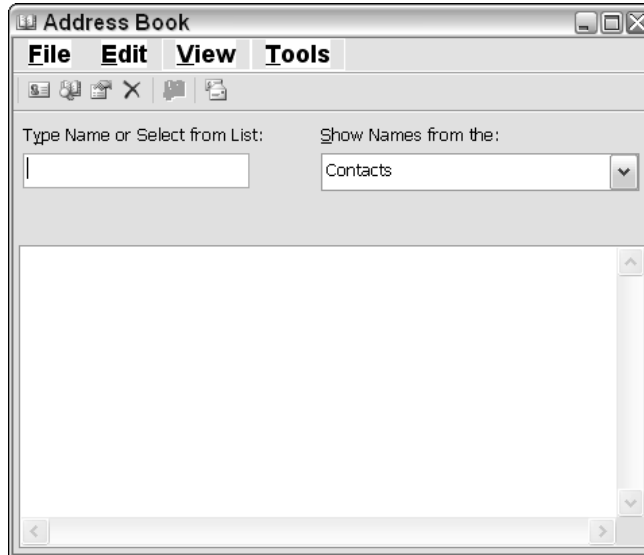


Figure 5-9

Consider Listing 5-13, which manipulates the address book.

Visual Basic

```

Private Function GetInfoFromAddressBook(ByVal contact As String, ByVal name As
String) As Outlook.AddressEntry
    If name = Nothing OrElse name.Trim().Length = 0 Then
        Return Nothing
    End If
    If contact = Nothing OrElse contact.Trim().Length = 0 Then
        contact = "Contacts"
    End If

    contact = contact.Trim()
    name = name.Trim()

    If Not Session.AddressLists Is Nothing And Session.AddressLists.Count > 0
Then
        Dim addressLists As Outlook.AddressLists
        addressLists = Session.AddressLists
        Dim address As Outlook.AddressList
        For Each address In addressLists
            If address.Name.Trim() = contact Then
                Dim enTry As Outlook.AddressEntry
                For Each enTry In address.AddressEntries
                    If enTry.Name.Trim() = name Then
                        Return enTry
                    End If
                Next
            End If
        Next
    End If
Next

```

```

End If
Return Nothing
End Function

```

C#

```

private Outlook.AddressEntry GetInfoFromAddressBook(string contact, string name)
{
    if (name == null || name.Trim().Length == 0)
        return null;
    if (contact == null || contact.Trim().Length == 0)
        contact = "Contacts";

    contact = contact.Trim();
    name = name.Trim();

    if (Session.AddressLists != null && Session.AddressLists.Count > 0)
    {
        Outlook.AddressLists addressLists = Session.AddressLists as
Outlook.AddressLists;
        foreach (Outlook.AddressList address in addressLists)
        {
            if (address.Name.Trim() == contact)
            {
                foreach (Outlook.AddressEntry entry in
address.AddressEntries)
                {
                    if (entry.Name.Trim() == name)
                    {
                        return entry;
                    }
                }
            }
        }
    }
    return null;
}

```

Listing 5-13: Outlook address book probe routine

There's quite a lot going on in the code so let's take it slowly. First, the code performs some rudimentary sanity checks with `if` statements. Then, an active session is used to grab the address lists. We talked about MAPI sessions at the beginning of the chapter so you should be familiar with the concept. An address book may contain one or more address lists that each contains unique identifiers. Figure 5-10 shows the default address list available in the right drop-down box. It is this list that is returned through the `AddressList` collection object.

The code begins an iterative dive into the `AddressList` collection. The address objects are exactly equal to the values that appear in the drop-down box in Figure 5-13. Finally, each address object is made up of one or more address entry objects. These address entry objects contain the actual address information that we are after. If a match is found, we simply return the match. Otherwise, we continue searching.

The code is only constructed to find a single instance of the address. However, for real-world applications, there may be duplicate addresses so the code should be amended to behave correctly. An implementation is trivial and best left as an exercise to the reader.

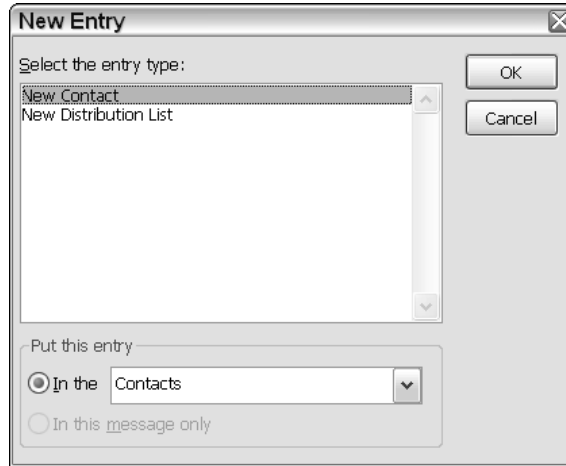


Figure 5-10

Events

Recall from earlier in the chapter that most Microsoft Outlook features are built from a few objects such as the `Application`, `Inspector`, and `Explorer` objects. As it turns out, these objects also expose numerous events. For instance, the application object exposes the `startup` and `shutdown` events. If you examine the default `OfficeCodeBehind`'s VSTO-generated code, you will notice that these events are wired to event handlers. The `ThisApplication_Startup` handler may be used to initialize application-level objects, while the `ThisApplication_Shutdown` handler may be used to cleanup application-level objects.

Consider the code in Listing 5-14.

Visual Basic

```
Public Class ThisApplication

    Private Sub ThisApplication_MAPILogonComplete() Handles Me.MAPILogonComplete
        MessageBox.Show("Welcome " &
            Me.GetNamespace("MAPI").CurrentUser.Name.ToString())
    End Sub

    Private Sub ThisApplication_Startup(ByVal sender As Object, ByVal e As
        System.EventArgs) Handles Me.Startup

    End Sub

    Private Sub ThisApplication_Shutdown(ByVal sender As Object, ByVal e As
        System.EventArgs) Handles Me.Shutdown

    End Sub

End Class
```

C#

```
using System;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;
using Office = Microsoft.Office.Core;

namespace MAPLogon
{
    public partial class ThisApplication
    {
        private void ThisApplication_Startup(object sender, System.EventArgs e)
        {
            this.MAPILogonComplete += new
Microsoft.Office.Interop.Outlook.ApplicationEvents_11_MAPILogonCompleteEventHandler
(ThisApplication_MAPILogonComplete);
        }

        void ThisApplication_MAPILogonComplete()
        {
            MessageBox.Show("Welcome " +
this.GetNamespace("MAPI").CurrentUser.Name.ToString());
        }
    }
}
```

Listing 5-14: Event handling code to display a welcome message prompt

The code is straightforward enough so that an explanation is not necessary. However, Visual Basic programmers will note that the `MAPILogonComplete` event must be explicitly added in the VSTO designer. Once the handler appears in the `OfficeCodeBehind` file, you may then place the code in the event handler. You should note also that the `GetNamespace` method call presented in Listing 5-13 is an alternative approach to retrieving a MAPI session.

The Explorer window drives the folder navigation functionality of Microsoft Outlook. The events that are exposed through the Explorer window relate in part to these folders and the data they may contain. If you would like to apply functionality to the Explorer, it's a good idea to examine the events that it exposes.

The `Inspector` object displays data based on selections in the Explorer window. For instance, double-clicking on a folder in the Explorer window launches the Inspector window. As part of the Inspector launch, several events are fired. To apply functionality, you might consider subscribing to some of these events.

For both of these objects, the approach is the same. You will need to wire up your handlers to the events that you are interested in and place code in the event handler so that it may be executed when the event handler is called. For Visual Basic applications, you will need to add the event handlers by clicking on the event drop-down list.

Since VSTO-based application code is executed as an add-in, your calling code will simply react to Microsoft Outlook events. For this reason, Microsoft Outlook add-ins typically follow a predefined format of subscribing to Microsoft Outlook events and handling these events appropriately. Consider the code in Listing 5-15.

Visual Basic

```
Public Class ThisApplication
    Private Declare Auto Function PlaySound Lib "winmm.dll" _
        (ByVal lpszSoundName As String, ByVal hModule As Integer, _
        ByVal dwFlags As Integer) As Integer

    Private Sub ThisApplication_NewMail() Handles Me.NewMail
        PlaySound("C:\program files\messenger\newalert.wav", 0, &H20000)
    End Sub

    Private Sub ThisApplication_Startup(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Startup
    End Sub

    Private Sub ThisApplication_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Shutdown

    End Sub

End Class
```

C#

```
using System;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;
using Office = Microsoft.Office.Core;

using System.Runtime.InteropServices;

namespace YouGotMail
{
    public partial class ThisApplication
    {
        [DllImport("WinMM.dll")]
        public static extern bool PlaySound(byte[] filename, int typeOfSound);

        private void ThisApplication_Startup(object sender, System.EventArgs e)
        {
            this.NewMail += new
Microsoft.Office.Interop.Outlook.ApplicationEvents_11_NewMailEventHandler(ThisAppli
cation_NewMail);
        }

        void ThisApplication_NewMail()
        {
            [DllImport("winmm.dll", EntryPoint = "PlaySound", CharSet =
CharSet.Auto)]
            private static extern int PlaySound(String pszSound, int hmod, int falgs);
            static void Main(string[] args)
            {
                PlaySound(@"C:\program files\messenger\newalert.wav", 0, 0x0000);
            }
        }
    }
}
```

```

    }

    private void ThisApplication_Shutdown(object sender, System.EventArgs e)
    {
    }
}
}

```

Listing 5-15

This application simply subscribes to the new mail event. Once new mail is received in Microsoft Outlook, the event handler is called. The code in the event handler plays a sound file for each piece of mail received. For Visual Basic, you will need to add the `NewMail` event from the events drop-down list.

Microsoft Outlook exposes a number of events that calling code may subscribe to in an application. The basic code for event hook up is essentially the same. However, the events exposed are not as rich as the Microsoft Word or Microsoft Excel.

Data Manipulation

Microsoft Outlook is designed to process large amounts of data to include raw text as well as files in the form of attachments. Eventually, end users may need to sift through these documents in search of data. The find functionality is a common tool used to search for pieces of text inside folders. The find functionality works by passing the text search string into the Outlook API. While the Outlook API executes the search request, the user interface remains fairly responsive. Figure 5-11 shows the Outlook search dialog.

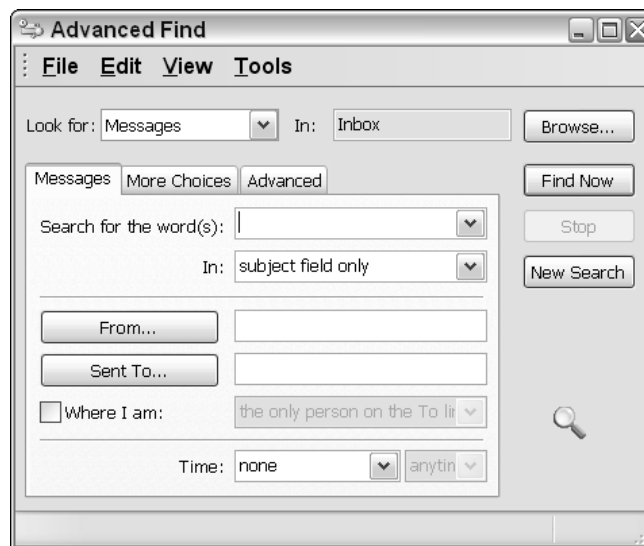


Figure 5-11

Searching for Data

There are several options to implementing search functionality in Microsoft Outlook. We examine these in terms of complexity. If you take the course of least resistance, the simplest approach to searching is to show the find dialog in Microsoft Outlook. However, Outlook does not easily expose any of the common dialogs that are found in Microsoft Word. So, the next best thing is to build a simple loop to probe for the contents, as in Listing 5-16. The behavior roughly replicates the end-user supported Find dialog provided by Microsoft Outlook.

Visual Basic

```
Private Sub SearchRoutine(ByVal searchToken As String)
    Dim inbox As Outlook.MAPIFolder
    inbox = Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderInbox)
    Dim count As Integer
    count = inbox.Items.Count
    Dim foundItem As String
    foundItem = String.Empty

    Dim found As Object
    For Each found In inbox.Items
        foundItem = CallByName(note, "Subject", CallType.Get).ToString()
    If foundItem = searchToken Then
        MessageBox.Show("Found a suspicious item")
    End If
    Next
End Sub
```

C#

```
public void SearchRoutine(string searchToken)
{
    Outlook.MAPIFolder oInbox =
Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderInbox);
    int count = oInbox.Items.Count;
    string retVal = string.Empty;

    foreach (object oResult in oInbox.Items)
    {
        retVal = (string)oResult.GetType().InvokeMember("Subject",
System.Reflection.BindingFlags.Instance | System.Reflection.BindingFlags.Public |
System.Reflection.BindingFlags.GetProperty, null, oResult, null, null);
        if (retVal == searchToken)
        {
            MessageBox.Show("Found a suspicious item");
        }
    }
}
```

Listing 5-16: Simple iterative loop to find data

The code retrieves a reference to the inbox folder and begins an iterative descent into the items collection searching for items with the subject matching the keyword “virus.” I’ll concede that though the approach is simple, the C# code is ugly. Part of the eyesore is the reflection inside the `for` loop. The VB code is a

lot more presentable because the reflection code is wrapped in the `CallByName` method. C# offers no such luxury. The reflection expense is also costly.

For efficiency reasons, you should avoid using a `for` loop. You can improve the performance if you choose another approach. Consider Listing 5-17.

Visual Basic

```
Dim inbox As Outlook.MAPIFolder
    inbox = Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderInbox)
    Dim oResult As Object
    oResult = inbox.Items.Find("[Subject] = ""virus"")
    If Not oResult IsNot Nothing Then
        Dim data as Outlook.MailItem = DirectCast(oResult, Outlook.MailItem)
    If data IsNot Nothing Then

        End If
    End If
```

C#

```
//OutlookSearch.Properties.
    Outlook.MAPIFolder oInbox =
Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderInbox);
    object oResult = oInbox.Items.Find(@"[Subject] = ""virus"");

    if (oResult != null)
    {
        Outlook.MailItem data = (Outlook. MailItem)oResult;
        if (data != null)
        {
        }
    }
}
```

Listing 5-17: Search functionality code using the Find method

The code first obtains a reference to the inbox folder and invokes the `Find` method for the `Items` collection. Notice that the `Find` method accepts a filter string parameter. If the find is successful, the returned object contains the items matching the filter. This filter is fast and more efficient than writing a loop to iterate the contents of the collection so its use is preferred over the `for` loop approach.

The filter parameter passed in roughly resembles the .NET filter mechanism commonly found in dataset manipulation. The property to be searched is marked with square brackets, and the actual search request item must be placed inside double quotation marks.

Filters are specific to the folder being searched so passing in an invalid filter results in an exception. To find all the available filters for a specific folder, it's easier to let Microsoft Outlook display a valid list for you. Click the Inbox folder in the left pane of Microsoft Outlook. Select Arrange by from the Tools Menu. Select Fields and a dialog should appear, as shown in Figure 5-11. Notice that this dialog allows you to add custom fields that in turn may be used in your calling code. The mechanism is very flexible.

However, one limitation exists. The find returns only the first item matching the filter. If other items exist, they are simply ignored. So let's amend the code to get Listing 5-18.

Visual Basic

```
Dim oInbox As Outlook.MAPIFolder =
Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderInbox)
Dim oResult As Object = oInbox.Items.Find("[Subject] = " & "virus")
Do
    oResult = oInbox.Items.FindNext()
Loop While Not oResult Is Nothing

Dim data As Outlook.MailItem
If Not oResult Is Nothing Then
    MessageBox.Show("Found")
    data = CType(oResult, Outlook.MailItem)
End If
```

C#

```
Outlook.MAPIFolder oInbox =
Session.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderInbox);
object oResult = oInbox.Items.Find(@"[Subject] = " & "virus");
do
{
    oResult = oInbox.Items.FindNext();
} while (oResult != null);

Outlook.MailItem data;
if (oResult != null)
{
    MessageBox.Show("Found");
    data = (Outlook.MailItem)oResult;
}
```

Listing 5-18: Search functionality code for multiple terms

The extra code is simply a `do` loop that fires the `FindNext` method. The `FindNext` method executes the parameters of the previous find call; this is the reason for the empty parameter list. The loop executes until all items matching the filter criteria is found. This implementation is still much faster than iterating the items collection with a home grown `for` loop.

The VB code snippet shows the `CType` operator being used to perform the cast. Strictly speaking, `CType` is a conversion operator. For improved efficiency, you should consider using a specialized casting operator such as `DirectCast` or `TryCast`, since these operators do not use Visual Basic's runtime helper routines for conversion.

While the `Find` method is certainly efficient for probing the items collection, it isn't without limitations. For instance, the filter parameter cannot grow in sophistication. It is limited to simple `AND`, `OR`, or `NOT` constructs. Consider these limits of filtering:

```
oInbox.Items.Find(@"[Subject] = " & "virus" & " AND [To] = " & "Microsoft@hotmail.com")
oInbox.Items.Find(@"[Subject] = " & "virus" & " OR [To] = " & " Microsoft@hotmail.com")
oInbox.Items.Find(@"[Subject] = " & "virus" & " AND NOT [To]= " & "Microsoft@hotmail.com")
```

The `AND`, `OR`, and `NOT` operators retain their logical meaning and an explanation is not necessary at this point.

Advanced Data Search

The filter parameter used in the `Find` method cannot execute partial matches. A match for a keyword `virus` in the subject field will return no results if the items being searched contain the word `viruses`. In some use cases, that lack of flexibility is a significant deterrent. To address use cases like this, a more powerful search approach must be adopted. Listing 5-19 shows an example.

Visual Basic

```
Private Sub ThisApplication_Startup(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Startup
    Dim scope As String = "'Inbox'"
    Dim filter As String = """"subject"" LIKE '%vir%'"
    Dim tag As String = "uniqueTag"
    Dim subFolders As Object = False

    Dim retVal As Outlook.Search = Me.AdvancedSearch(scope, Filter, subFolders,
tag)

    If Not retVal Is Nothing Then
        System.Threading.Thread.Sleep(200)
        If retVal.IsSynchronous = False Then
            retVal.Stop()
        End If
    End If
End Sub
```

C#

```
private void ThisApplication_Startup(object sender, System.EventArgs e)
{
    string scope = @"'Inbox'";
    string filter= @""subject"" LIKE '%vir%';
    string tag = "uniqueTag";
    object subFolders = false;

    Outlook.Search retVal = this.AdvancedSearch(scope, filter, subFolders,
tag);

    if (retVal != null)
    {
        System.Threading.Thread.Sleep(200);
        if (retVal.IsSynchronous == false)
            retVal.Stop();
    }
}
```

Listing 5-19: Advanced search method

You will notice that Listing 5-19 essentially replicates the search functionality coded in Listing 5-17. However, the search filter has been extended to include text matching. The search should return items that contain partial matches for the keyword `vir`. For instance, there will be a successful match for `viruses`.

It may surprise you to know that this code may fail more often than necessary. In fact, for large folders, `retVal` may almost always be equal to `null` even though matches exist. The reason for this quirky behavior is that the `AdvancedSearch` method executes asynchronously; that is, it occurs on a separate thread.

Chapter 5

To correct this behavior, we need to account for the synchronicity. Listing 5-20 shows how you should proceed.

Visual Basic

```
Dim scope As String = "'Inbox'"
Dim filter As String = """"subject"" LIKE '%vir%'"
Dim tag As String = "uniqueTag"
Me.AdvancedSearch(scope, filter, False, tag)

'event handler code
Private Sub ThisApplication_AdvancedSearchComplete(ByVal SearchObject As
Microsoft.Office.Interop.Outlook.Search) Handles Me.AdvancedSearchComplete

    If Not SearchObject Is Nothing Then
        Dim item As Outlook.MailItem = ActiveInspector ().CurrentItem
        Dim conflicts As Outlook.Conflicts = item.Conflicts
        If Not conflicts Is Nothing And conflicts.Count > 0 Then
            'perform some analysis
            Dim parent As Outlook.MailItem = conflicts.Parent
            If Not Parent Is Nothing Then
                If Parent.Importance > Outlook.OlImportance.olImportanceNormal
Then
                    MessageBox.Show("A piece of mail with subject 'virus'
contains a conflict and cannot be deleted")
                Else
                    item.Delete()
                End If
            End If
        End If
    End If
End Sub
```

C#

```
string scope = @"'Inbox'";
string filter= @""""subject"" LIKE '%vir%'"';
string tag = "uniqueTag";
object subFolders = false;

this.AdvancedSearchComplete += new Microsoft.Office.Interop.Outlook
.ApplicationEvents_11_AdvancedSearchCompleteEventHandler(ThisApplication_
AdvancedSearchComplete);
this.AdvancedSearch(scope, filter, subFolders, tag);

//event handler code
void
ThisApplication_AdvancedSearchComplete(Microsoft.Office.Interop.Outlook.Search
SearchObject)
{
    if (SearchObject != null)
    {
        Outlook.MailItem item = ActiveInspector().CurrentItem as
Outlook.MailItem;
        Outlook.Conflicts conflicts = item.Conflicts;
        if (conflicts != null && conflicts.Count > 0)
```

```

        {
            //perform some analysis
            Outlook.MailItem parent = conflicts.Parent as Outlook.MailItem;
            if (parent != null)
            {
                if (parent.Importance >
                    Outlook.OlImportance.olImportanceNormal)
                {
                    MessageBox.Show("A piece of mail with subject 'virus'
contains a conflict and cannot be deleted");
                }
                else
                    item.Delete();
            }
        }
    }
}

```

Listing 5-20: Advanced Search Method

To run this code, you will need to place the first four lines of code in a routine other than the startup routine. The reason for this is that there are no `Inspector` objects that are available at startup, therefore `CurrentItem` will always return a null value. One simple workaround is to place the first four lines of code in a double-click event handler for an `Inspector` window. At that point, the `CurrentItem` will contain a valid object and the code will execute correctly.

This is much better. After setting up some plumbing, we hook into the `AdvancedSearchCompleted` event. When the asynchronous search completes, it will invoke our event handler. The event handler simply displays the filter that was passed in. You can get more creative with the routine to analyze the search results. But this is enough to give you the basic idea.

When the `AdvancedSearchCompleted` handler is called, you will notice that the `ActiveInspector` object is used to obtain the `CurrentItem`. Before processing the `CurrentItem`, the code checks to make sure that the item is not in conflict. If all is well, the item is deleted, otherwise an appropriate message is displayed.

The `conflicts` collection is made up of `conflict` objects. `Conflict` objects represent items that are in some sort of conflict with another Microsoft Outlook item. The `conflicts` collection allows code to iterate the collection. In this manner, it is possible to find the complete details for all the conflicts.

This code fix does solve our previous problems, but it is still buggy. Since Microsoft Outlook can fire searches through an end-user request, our custom code will be fired even if we did not explicitly invoke the search method. This is really distasteful especially if the `AdvancedSearchCompleted` method performs some functionality that wasn't called for by the user.

The architects implementing VSTO decided to fix this annoyance by providing an optional `Tag` parameter. In Listing 5-21, we set this parameter to some unique string. When `AdvancedSearchCompleted` fires, we can write a simple if statement to check the value of the tag property to see if it matches what

we are expecting. If the condition evaluates to true, we can proceed since we are guaranteed that the search was invoked from our calling code. Consider Listing 5-21:

Visual Basic

```
Private Sub ThisApplication_AdvancedSearchComplete(ByVal SearchObject As
Microsoft.Office.Interop.Outlook.Search) Handles Me.AdvancedSearchComplete

    If Not SearchObject Is Nothing AndAlso compSearch.Tag =
"uniqueTag" Then
        MessageBox.Show(compSearch.Filter.ToString())
    End If
End Sub
```

C#

```
void
ThisApplication_AdvancedSearchComplete(Microsoft.Office.Interop.Outlook.Search
SearchObject)
{
    if (SearchObject!= null && SearchObject.Tag == "uniqueTag")
    {
        MessageBox.Show(compSearch.Filter.ToString());
    }
}
```

Listing 5-21: Advanced search correction

Although search routines are very handy, they are resource hogs especially if the target contains a number of items. It is quite common for a busy executive to contain several hundred messages in their inbox for one day. Some optimizations can be performed during the search so that the routine will complete faster and only examine data specific to the search.

Object Model Guard

Microsoft Outlook security has been redesigned. The redesigned includes more restrictions on parts of the OOM to reduce the surface area of attack due to malicious viruses spreading through email applications. The change also extends down into the Collaboration Data Objects (CDO) model so that code cannot circumvent Outlook security by routing through the CDO layer. Even MAPI has been reviewed to be more robust and resistant to attacks. Although most of these changes fall outside the scope of this text, we review key parts of the Object Model Guard (OMG) since it plays an influential role in the overall security of Microsoft Outlook.

The OMG is a layer of security imposed on some Outlook objects. The layer restricts access to certain parts of the object model. The OMG assumes that executing code runs untrusted and must obtain the permission of the end user in order to carry out certain types of functionality. For instance, the code in Listing 5-4 earlier in the chapter first creates an email and then sends it to a specified address. When the send method of the mail object is called, the OMG displays a security dialog prompt. At this point, the end user can choose to allow the email to be sent or to cancel the process. If you need to suppress the security dialog, you must use the trusted object that Outlook provides. Listing 5-22 shows an example.

Visual Basic

```
Dim NewMail As Outlook.MailItem
NewMail
Me.CreateItem(Microsoft.Office.Interop.Outlook.OlItemType.olMailItem)
NewMail.Subject = "This is my email subject"
NewMail.To = "vapordan@hotmail.com"
NewMail.Send()
```

C#

```
Outlook.MailItem NewMail =
this.CreateItem(Microsoft.Office.Interop.Outlook.OlItemType.olMailItem) as
Outlook.MailItem;
NewMail.Subject = "This is my email subject";
NewMail.To = "vapordan@hotmail.com";
NewMail.Send();
```

Listing 5-22: Overriding the OMG

In this modified example, notice that the `CreateItem` is invoked from the current object on the stack. This trusted object is the application object passed by Outlook to the custom add-in that you created.

If you revert back to the code presented in Listing 5-4, the security dialog will be displayed once more because the code calls the `CreateItem` method using a reference obtained from the untrusted `Application` object. The untrusted `Application` object is not the same object as the trusted object provided by Outlook. It is important to understand the difference between these two approaches. For more flexibility, you can retrieve a reference to the trusted application object by accessing the special `ThisApplication` object.

The security situation is complicated somewhat when Microsoft Exchange forms part of the picture. If Microsoft Outlook is configured to run without Microsoft Exchange, an administrator can disable the OMG. Once the OMG has been disabled, the code approach presented in Listing 5-4 will execute without displaying a security dialog. If Microsoft Outlook is configured to run with Microsoft Exchange, an administrator can configure the OMG to apply security restrictions to certain add-ins. The behavior of the OMG itself can be modified as well. For obvious reasons, only knowledgeable persons should perform that type of application audit.

So, you can see that it is quite possible for the code in Listing 5-22 to fail. Consider the case where Microsoft Outlook is configured to run with Microsoft Exchange and an administrator has configured all managed add-ins to deny access to email. In this scenario, the trusted object behavior will be overridden and the email will not be sent. Instead, a `ComException` error will be thrown.

Working with Office Controls

Microsoft Outlook exposes limited functionality for customizing Outlook itself. This is particularly unnerving if you are considering making the switch to VSTO and expect that type of support. In addition, the workaround to implement such functionality is ugly at best. Still, if you absolutely require that type of functionality, the next few sections outline a plan of attack.

Before diving into code, you should note that Microsoft Outlook behaves very differently from Microsoft Excel and Word. And, its programming surface is not as rich as these other environments. For instance, the concept of the Office task pane does not apply to Microsoft Outlook. Although it is possible to build a control that resembles the Office task pane in Microsoft Outlook, it really is unnecessary. Rather, your design should be rethought in terms of functionality that can be provided through a toolbar or a menu. If you still cannot refine your design to fit into this mold, your add-in is probably not suited for Microsoft Outlook.

Toolbar Customization

The Microsoft Outlook application object does not expose the familiar `CommandBars` object collection. Recall from earlier chapters that this collection allowed access to and manipulation of the command bars in an Office document. Without this basic access, you will need to use another approach to adding items to the Microsoft Office toolbar. We consider these approaches next.

Adding Buttons to the Toolbar

Our workaround involves using the `Explorer` object to gain access to the `CommandBars` object. Listing 5-23 shows how to proceed.

Visual Basic

```
public class ThisApplication
    Dim NewCommandBar As Office.CommandBar
    Dim NewCBarButton As Office.CommandBarButton

    Private Sub ThisApplication_Startup(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Startup
        NewCommandBar = Me.ActiveExplorer.CommandBars.Add("myButton",
Office.MsoBarPosition.msoBarTop)
        Dim NewCBarButton As Office.CommandBarButton =
CType(NewCommandBar.Controls.Add(Office.MsoControlType.msoControlButton),
Office.CommandBarButton)
        NewCBarButton.Caption = "my new button"
        NewCBarButton.FaceId = 563
        NewCommandBar.Visible = True
        AddHandler NewCBarButton.Click, AddressOf Me.NewCBarButton_Click
    End Sub

    Private Sub ThisApplication_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Shutdown

    End Sub
    Private Sub NewCBarButton_Click(ByVal ctrl As
Microsoft.Office.Core.CommandBarButton, ByRef Cancel As Boolean)
        MessageBox.Show("Hello, World")
    End Sub
End Class
```

C#

```
using System;
using System.Windows.Forms;
```

```

using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;
using Office = Microsoft.Office.Core;

namespace OutlookToolbars
{
    public partial class ThisApplication
    {
        Office.CommandBar NewCommandBar;
        Office.CommandBarButton NewCBarButton;
        private void ThisApplication_Startup(object sender, System.EventArgs e)
        {
            Explorers.NewExplorer += new
Microsoft.Office.Interop.Outlook.ExplorersEvents_NewExplorerEventHandler(Explorers_
NewExplorer);
        }

        void Explorers_NewExplorer(Microsoft.Office.Interop.Outlook.Explorer
Explorer)
        {
            Outlook.Explorer explorer = this.ActiveExplorer();
            Office.CommandBars newCommandBar = explorer.CommandBars;

            object val = false;
            NewCommandBar = newCommandBar.Add("myButton",
Office.MsoBarPosition.msoBarTop, false, false);
            NewCBarButton =
(Office.CommandBarButton)NewCommandBar.Controls.Add(Office.MsoControlType.msoContro
lButton, missing, missing, missing, false);
            NewCBarButton.Caption = "button1";
            NewCBarButton.FaceId = 563;
            NewCommandBar.Visible = true;
            NewCBarButton.Visible = true;
            NewCBarButton.TooltipText = "My new button";
            NewCBarButton.Click += new
Microsoft.Office.Core._CommandBarButtonEvents_ClickEventHandler(NewCBarButton_Click
);
        }

        void NewCBarButton_Click(Microsoft.Office.Core.CommandBarButton Ctrl, ref
bool CancelDefault)
        {
            MessageBox.Show("Hello, World!");
        }

        private void ThisApplication_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}

```

Listing 5-23: Adding buttons to the toolbar in Microsoft Outlook

The essential difference from previous code provided earlier to manipulate toolbars is that an `Explorer` object is used to obtain a reference to the `CommandBars` object collection. With this reference, we add our new button using the `add` parameter. Figure 5-12 shows the custom button added to the toolbar.

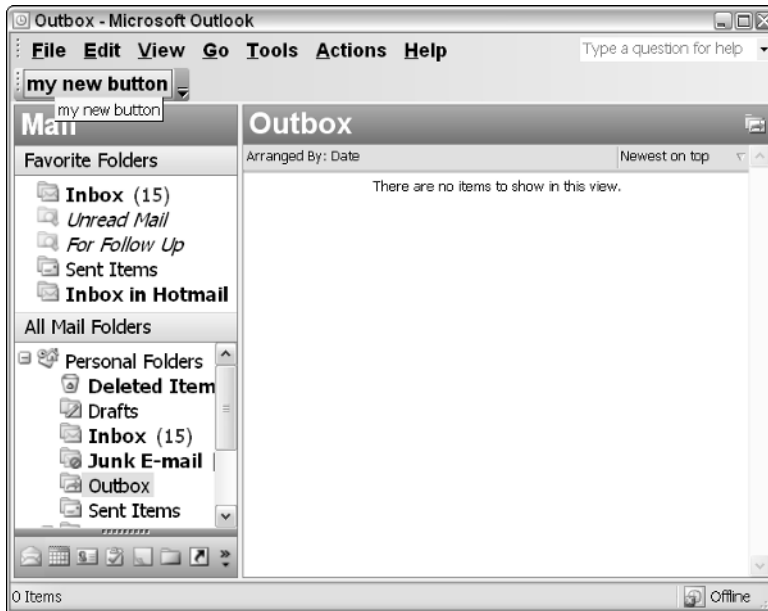


Figure 5-12

Although Visual Basic can perform automatic casting in some instances, you will need to perform an explicit cast when manipulating menus and toolbars. Otherwise, the code will compile but fail with a runtime exception, indicating that the array value is out of bounds.

Once the button is created, we set up the usual captions and icon. In this case the icon is loaded at index 563, that happens to be a calculator image. Finally, the button is hooked to a click event so that we can take some action when it is clicked. You should be familiar with the event wiring code because it was described fully in Chapter 2.

When building toolbar buttons, it's always a good idea to first find out if there is an existing button on the toolbar that matches yours. This simple check can prevent duplicates. If a possible duplicate is detected, you can simply remove the existing button before adding your own button. The code to remove a toolbar button is provided in the next section.

Removing Buttons from the Toolbar

Buttons may also be removed from the Microsoft Outlook toolbar. You will notice that the key difference is that the toolbar is being accessed through the `Explorer` object. The rest of the code remains virtually identical to code in the previous chapters. For clarity's sake, let's visit a delete example. Consider Listing 5-24.

Visual Basic

```

Dim explorer As Outlook.Explorer = Me.ActiveExplorer           Dim NewCommandBar
As Office.CommandBars = explorer.CommandBars

    'do not replicate command bar objects
    Dim cmdBar As Office.CommandBar
    For Each cmdBar In NewCommandBar
        If cmdBar.Name.Equals("ShortcutBar") Then
            cmdBar.Delete()
            Exit For
        End If
    Next

```

C#

```

Outlook.Explorer explorer = this.ActiveExplorer;           Office.CommandBars
newCommandBar = explorer.CommandBars;

    //do not replicate command bar objects
    foreach (Office.CommandBar cmdBar in newCommandBar)
    {
        if (cmdBar.Name.Equals("ShortcutBar"))
        {
            cmdBar.Delete();
            break;
        }
    }

```

Listing 5-24: Code to delete CommandBar objects in Microsoft Outlook

Again, an Explorer object is used to gain access into the CommandBar object collection. Next, the code simply iterates the collection looking for the `shortcutbar` keyword. If one is found, the object is deleted. While it isn't likely for this code to fail, it is good programming practice to employ exception handling around the delete call.

There is a good chance that the specific keyword may not find the system `shortcutbar`. Microsoft may change this string identifier at some point in the future. Or, these strings are simply not defined for applications that run in a foreign language context. For this reason, it's best to define the string in a configuration file so that it can be easily modified without recompiling the code.

Menu Customization

Menus are an integral part of Microsoft Office, and Outlook is no different. End users have grown accustomed to menus and expect that functionality to be available in any application based on Microsoft Office. Because of this expectation, code should not remove or suppress default menus. Application requirements that explicitly remove default menus should be viewed with suspicion. Instead, you should consider disabling specific menu items to give the user some inclination that some functionality is not available.

Adding Items to Menus

As with toolbar customization, the application object does not provide a reference to the appropriate toolbar collection in Microsoft Outlook. However, we can easily adapt to this nuisance by using the `ActiveExplorer()` method to obtain a suitable reference. Listing 5-25 shows an example.

Visual Basic

```
public class ThisApplication
    Dim menubar As Office.CommandBar
    Dim cmdBarControl As Office.CommandBarPopup
    Dim menuCommand As Office.CommandBarButton
    Private Sub ThisApplication_Startup(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Startup
        ' Add the menu.
        menubar = Me.ActiveExplorer().CommandBars.ActiveMenuBar
        cmdBarControl = CType (menubar.Controls.Add(Office.MsoControlType
.msoControlPopup, , , menubar.Controls.Count, True), Office.CommandBarPopup)
        If Not cmdBarControl Is Nothing Then
            cmdBarControl.Caption = "&Add-in Tools"
            ' Add the menu command.
            menuCommand = CType (cmdBarControl.Controls.Add(Office.MsoControlType
.msoControlButton, , , , True), Office.CommandBarButton)
            menuCommand.Caption = "&Word Count..."
            menuCommand.Tag = DateTime.Now.ToString()
            AddHandler menuCommand.Click, AddressOf menuCommand_Click
        End If
    End Sub
    Private Sub menuCommand_Click(ByVal Ctrl As
Microsoft.Office.Core.CommandBarButton, ByRef CancelDefault As Boolean)
        MsgBox("Hello, World!")
    End Sub

    Private Sub ThisApplication_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Shutdown
        ?
    End Sub

End Class
```

C#

```
using System;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;
using Office = Microsoft.Office.Core;

namespace OutlookMenu
{
    public partial class ThisApplication
    {
        Office.CommandBar menubar;
        Office.CommandBarPopup cmdBarControl;
        Office.CommandBarButton menuCommand;
        private void ThisDocument_Startup(object sender, System.EventArgs e)
        {
            // Add the menu.
            menubar = this.ActiveExplorer().CommandBars.ActiveMenuBar;
            cmdBarControl = (Office.CommandBarPopup)menubar.Controls.Add(
                Office.MsoControlType.msoControlPopup, missing, missing,
                menubar.Controls.Count, true);
```

```

        if (cmdBarControl != null)
        {
            cmdBarControl.Caption = "&Add-in Tools";
            // Add the menu command.
            menuCommand = (Office.CommandBarButton)cmdBarControl.Controls.Add(
                Office.MsoControlType.msoControlButton, missing, missing, missing,
                true);
            menuCommand.Caption = "&Word Count...";
            menuCommand.Tag = DateTime.Now.ToString();
            menuCommand.Click += new
                Microsoft.Office.Core._CommandBarButtonEvents_ClickEventHandler(menuCommand_Click);
        }
        void menuCommand_Click(Microsoft.Office.Core.CommandBarButton Ctrl, ref
            bool CancelDefault)
        {
            MessageBox.Show("Hello, World!");
        }
    }
}

```

Listing 5-25: Code to add items to Microsoft Outlook menu

If you've been making judicious notes, the menu generation code appears in Chapter 4. All we've done here is attach to the menu object from the `ActiveExplorer` window instead of the `Application` object. A few notes are in order before continuing. Firstly, we point out that the variables to hold the menus must be global in scope; otherwise, they may be garbage collected resulting in menus that no longer work.

Another point worth noting is that some unique value must be given to the tag variable to allow the function to execute correctly. Microsoft Office uses the tag identifier to keep track of event handlers for a specific `commandbarcontrol`. The remainder of the code has been explained already in Chapter 4.

Removing Buttons from the Menu Bar

Removing buttons from the menu bar is equally easy. Listing 5-26 shows how to remove menus.

Visual Basic

```

Private Sub RemoveMenubar(ByVal menuTitle As String)
    'remove the menu
    Try
        Dim foundMenu As Object
        foundMenu =
            Me.ActiveExplorer().CommandBars.ActiveMenuBar.FindControl(Office.MsoControlType.mso
                ControlPopup, Nothing, menuTitle, True, True)
        If Not foundMenu Is Nothing Then
            foundMenu.Delete(True)
        End If
    Catch ex As Exception
        MessageBox.Show(ex.Message)
    End Try
End Sub

```

C#

```
private void RemoveMenubar(string menuItem)
{
    //remove the menu
    try
    {
        Office.CommandBarPopup foundMenu = (Office.CommandBarPopup)
            this.ActiveExplorer().CommandBars.ActiveMenuBar.
            FindControl(Office.MsoControlType.msoControlPopup,
                missing, menuItem, true, true);
        if (foundMenu != null)
        {
            foundMenu.Delete(true);
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Listing 5-26: Code to remove items from a Microsoft Outlook menu

The `removeMenuBar` routine accepts a string parameter. This parameter is used as a key to find the target menu option. The code uses the `FindControl` method of the `ActiveMenuBar` object to internally probe the menu bar collection for a menu matching the unique tag. Once the `FindControl` method has executed, the code simply tests the variable to see if it contains a valid object. If it does, the object is deleted. The code is wrapped in an exception handling block, just in case things go wrong.

Although Visual Basic can perform automatic casting in some instances, you will need to perform an explicit cast when manipulating menus and toolbars. Otherwise, the code will compile but fail with a runtime exception, indicating that the array value is out of bounds.

The `FindControl` call is optimized to search the collection efficiently, but it can only return a single instance. This is not a concern since each control should contain a unique tag. However, if there is a requirement to search for multiple controls, an iterative loop should be used instead of the `FindControl` method. As pointed out previously, an iterative loop is less efficient than the `FindControl` method.

Integrating Windows Applications with Outlook

The code snippets and discussions in this chapter have operated under the assumption that functionality is being added to Microsoft Outlook. Now, we consider the opposite case. The next piece of code demonstrates how to harness Microsoft Outlook functionality in Windows applications. We spend some time considering the case where items are dropped from Microsoft Outlook onto a Windows forms application.

Drag and drop support is another feature that end users have come to expect in any type of windows applications. In fact, end users typically drag objects from the desktop to running applications with the expectation that the application will service the request. However, these types of requests have no automatic implementation. They must be built in.

Drag and drop is implemented with the help of events. When the user initiates a drag operation, the `dragdrop` event for the application is fired. When the end user releases the object being dragged, the `drop` event is fired. Application code simply needs to listen for these two events and handle them appropriately to successfully service the request. Consider the code in Listing 5-27.

Visual Basic

```
Imports System.IO
Imports System.Collections
Imports System.Text

Public Class Form1
    Dim incomingTitle As String = "FileGroupDescriptor"
    Private Sub TreeView1_DragDrop(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles WinTree.DragDrop
        Dim str() As String = e.Data.GetFormats(True)
        Dim supported As Boolean = False
        Dim data As String
        For Each data In str
            If data = incomingTitle Then
                supported = True
                Exit For
            End If
        Next

        If Not supported Then
            'data file format not supported so simply exit
            Return
        End If

        Dim fileNames() As String = New String(5) {}
        Dim sizeByte As String = 512
        ' get the incoming data
        Dim ioStream As System.IO.Stream =
CType(e.Data.GetData("FileGroupDescriptor"), System.IO.Stream)
        Dim contents() As Byte = New Byte(sizeByte) {}
        ioStream.Read(contents, 0, sizeByte)
        ioStream.Close()
        Dim currentNode As TreeNode = WinTree.Nodes(0)
        If Not currentNode Is Nothing Then
            currentNode.Text = currentNode.Text + " 1 item added"
            WinTree.Update()
        End If

    End Sub

    Private Sub TreeView1_DragEnter(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles WinTree.DragEnter
        e.Effect = DragDropEffects.Link
    End Sub
End Class
```

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using MSForms = Microsoft.Vbe.Interop.Forms;
using System.IO;

namespace DragClient
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        string incomingTitle = "FileGroupDescriptor";
        private void WinTree_DragEnter(object sender, DragEventArgs e)
        {
            e.Effect = DragDropEffects.Link;
        }

        private void Form1_Load()
        {
        }

        private void WinTree_DragDrop(object sender, DragEventArgs e)
        {
            string[] str = e.Data.GetFormats(true);
            bool supported = false;
            foreach (string data in str)
            {
                if (data == incomingTitle)
                {
                    supported = true;
                    break;
                }
            }

            if (!supported)
            {
                //data file format not supported so simply exit
                return;
            }

            int sizeByte = 512;
            string[] fileNames = new string[5];
        }
    }
}
```

```

        // get the incoming data
        Stream incoming = (Stream)e.Data.GetData(incomingTitle);
        byte[] fileGroupDescriptor = new byte[sizeByte];
        incoming.Read(fileGroupDescriptor, 0, sizeByte);
        System.Text.StringBuilder sb = new StringBuilder();
        System.IO.Stream ioStream =
        (System.IO.Stream)e.Data.GetData("FileGroupDescriptor");
        byte[] contents = new Byte[sizeByte];
        ioStream.Read(contents, 0, sizeByte);
        ioStream.Close();

        TreeNode currentNode = WinTree.Nodes[0];
        if (currentNode != null)
        {
            currentNode.Text = currentNode.Text + "1 item added";
            WinTree.Update();
        }
    }
}
}
}

```

Listing 5-27: Drag and drop with Outlook emails

To run this code, you will need to create a Windows project. On the default form, drop a `treeview` control. From the property pages, set the `allowdrop` property of the `treeview` control to `true` so that the application can respond to `dragdrop` events. Also, you will need to double-click the `dragdrop` event and `dragenter` events so that event handlers can be created. These are your listening devices that monitor a drag operation to determine when it starts and when it ends. The `dragenter` event handler is not strictly necessary but it adds some flavor to the code in that it simply adds a drag effect so that the end user has some inclination that the drag operation is being initiated.

Add two nodes to the `treeview` as shown in Figure 5-13. Name these nodes `inbox` and `outbox`. Once the project is complete, you may then enter the code shown in Listing 5-27.

Compile and run the application. Figure 5-14 shows the application in action.

The `dragdrop` event handler contains the bulk of the code. The handler fires when the drop operation is complete. The code uses the `Drageventargs e` parameter to retrieve the data being dropped. Applications that support drag operations set the data type differently. The code checks for type because we assume that the data is coming from Microsoft Outlook. If it is data we are interested in, we take some action.

You should realize that the code is fairly basic. The intention is to show that common windows functionality can be incorporated into a VSTO application. The code shown in Listing 5-27 presents opportunities for optimization and code simplification. One such optimization can be made to the `WinTree_Drag` function. For example, the entire code can be replaced with this:

```

if(!e.Data.GetDataPresent(IncomingTitle, false))
return;

```

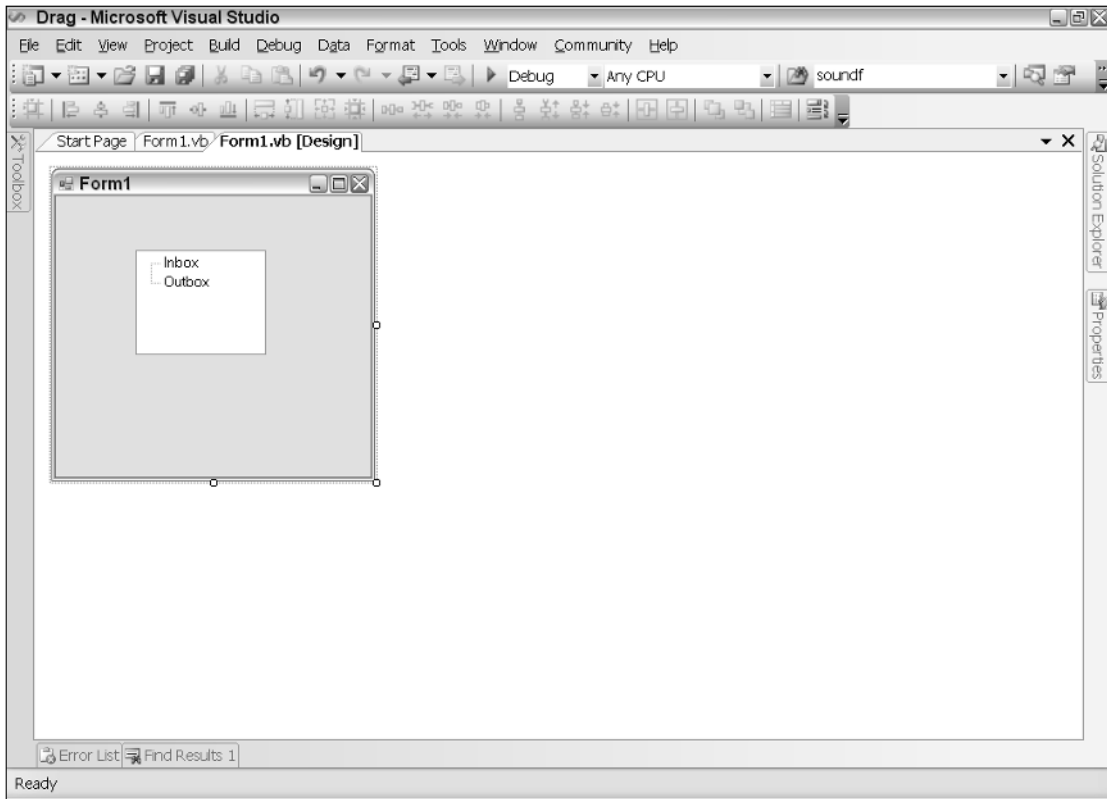



Figure 5-13



Figure 5-14

Now, consider the requirement that needs a Windows form added to an Outlook project. You should note that this is the opposite of the previous requirement. However, both these requirements span the spectrum of Outlook functionality. If you have a basic understanding of how to achieve both these requirements, you will be well on your way toward creating enterprise-level applications.

Create a new VSTO Outlook Add-in following the instructions at the beginning of the chapter. Name the project `AddinForm`. On the project node in the Visual Studio .NET IDE, right-click to view the project menu. A sample menu is shown in Figure 5-15.

Choose Add Windows Forms. The action will display an Add New Item dialog, as shown in Figure 5-16.

Add a text box and a button to the form. You may accept the default names. From this point on, you should notice that this is simply a Windows forms mini-application that may be programmed in the usual way. Our example is simple. When the button is clicked, we write some text to the text box on the form.

To add the code, double-click the button and add some text to the text box's `Text` method. Your bare-bones code should look like Listing 5-28.

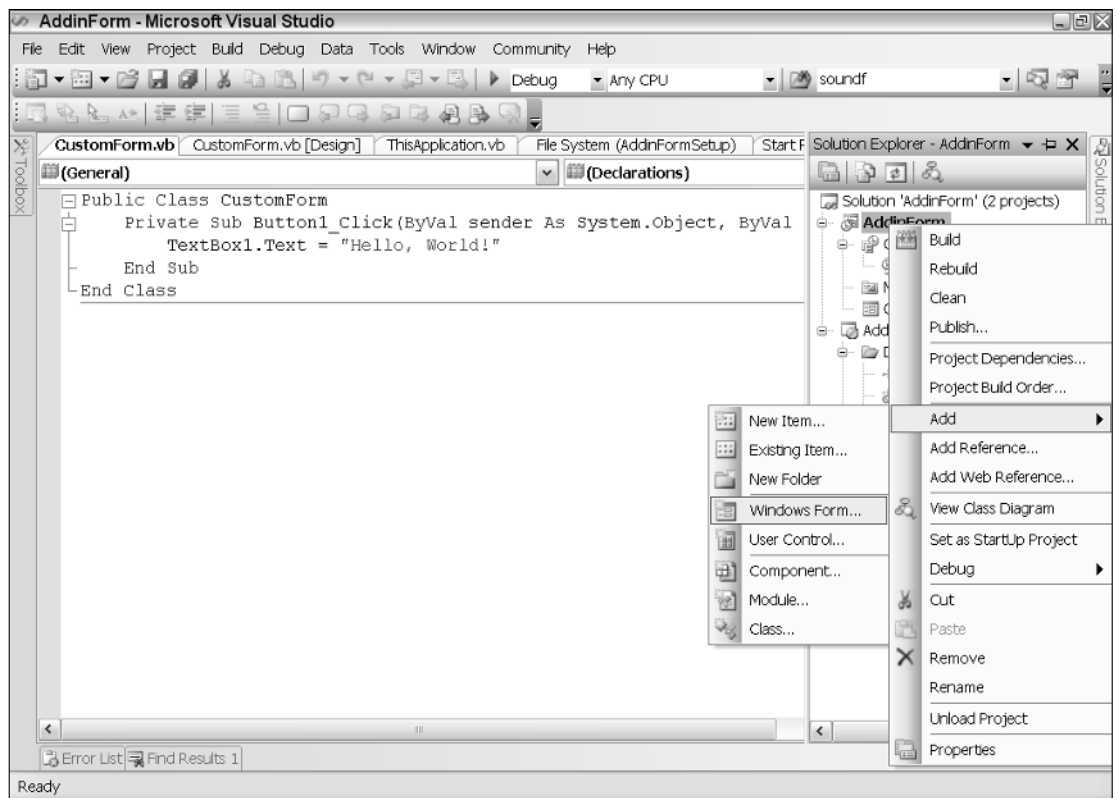


Figure 5-15

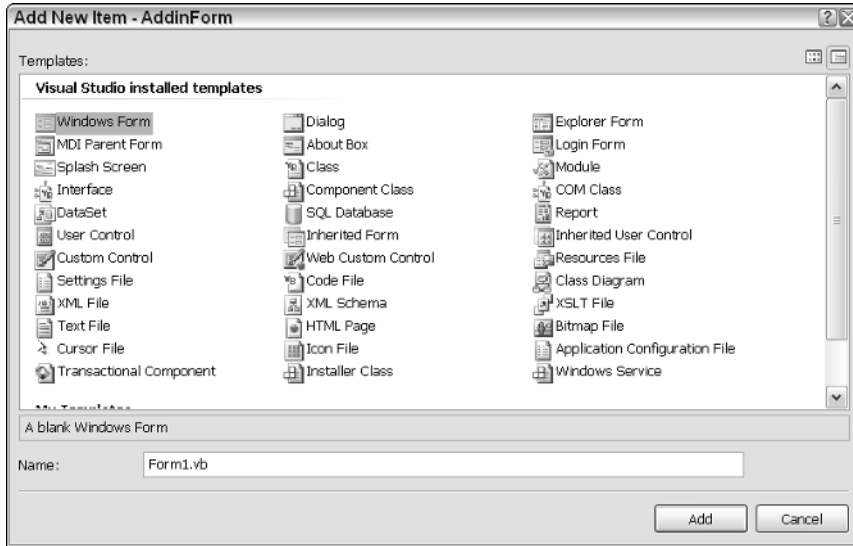


Figure 5-16

Visual Basic

```
Public Class CustomForm
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
        TextBox1.Text = "Hello, World!"
    End Sub
End Class
```

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace AddinForm
{
    public partial class CustomForm : Form
    {
        public CustomForm()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            textBox1.Text = "Hello, World!";
        }
    }
}
```

```

    }
}
}

```

Listing 5-28: Button click example

The form is ready for action at this point. However, the most important piece of the puzzle needs to be added. Microsoft Outlook must be informed that this window needs to be called in order for it to be displayed. There are several approaches to accomplishing this. One of the most common approaches is to subscribe to an Outlook event, as shown in Listing 5-29. When this event is fired, your Windows form will be displayed.

Visual Basic

```

public class ThisApplication
    Private WithEvents explorEvent As Outlook.Explorers

    Private Sub ThisApplication_Startup(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Startup
        explorEvent = Me.Explorers
        AddHandler explorEvent.NewExplorer, AddressOf Explorers_NewExplorer    End
Sub
    Private Sub Explorers_NewExplorer ByVal Inspector As
Microsoft.Office.Interop.Outlook.Explorer) Handles explorEvent.NewExplorer
        Dim myCustomForm As CustomForm
        myCustomForm = New CustomForm()
        myCustomForm.Show()
    End Sub
    Private Sub ThisApplication_Shutdown(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Shutdown

    End Sub

End class

```

C#

```

using System;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;
using Office = Microsoft.Office.Core;

namespace AddinForm
{
    public partial class ThisApplication
    {
        private void ThisApplication_Startup(object sender, System.EventArgs e)
        {
            this.Explorers.NewExplorer += new Microsoft.Office.Interop.Outlook
.ExplorersEvents_NewExplorerEventHandler(Explorers_NewExplorer);
        }

        void Explorers_NewExplorer(Microsoft.Office.Interop.Outlook.Explorer
Explorer)

```

```
        {
            CustomForm myForm = new CustomForm();
            myForm.Show();
        }

        private void ThisApplication_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}
```

Listing 5-29: Calling a custom form

That's it. The code is pretty simple. When a new Explorer window is opened (select a folder, right-click, and choose Open in new window), the `NewExplorer` event fires and calls your code. The code simply creates an instance of your custom-generated form and displays it. If you click the button, some text is written to the text box on the form.

You should note that this example, while conceptually simple, is complete. It shows how to call a custom form from code that is built at design time. But, you can easily modify the application to create and display a form at runtime as well. Also the example demonstrates how Outlook is able to display your form and cause the form's event to work correctly. This is the basis for all windows applications; a form is created, and some action is taken based on the controls that are placed on the form.

Finally, you should note that once outlook calls your code through the event handler, there aren't any restrictions on what the code can do. For instance, you can run another application or write and read files on disk. Also, the events that are fired can be cancelled as well. One advanced use is wiring into the appropriate Explorer window event, calling your customized form, and disabling Outlook's Explorer form from showing by cancelling the event. The possibilities are endless!

Summary

This chapter presented a number of key concepts that enable developers to build enterprise-level applications. The most important objects such as the `Application`, `Explorer`, and `Inspector` objects were presented with some code examples. These objects form the cornerstone of every Outlook-based add-in.

The chapter also provided an overview of MAPI. This email protocol powers the underpinnings of the Outlook application, and it is helpful to understand MAPI if you are to build robust applications.

The bulk of the chapter focused on building applications that use the `Explorer` and `Inspector` objects. These objects provide a number of events that may be useful to calling code. In fact, events are the main way that add-ins interact with Microsoft Outlook applications. Our sample snippets ran the gamut of application requirements that are common in industry.

It is not uncommon to find developers adding functionality to Microsoft Outlook that has nothing in common with emails or message management. One reason for this abuse is that end users constantly seek a single point of access to software-related tasks. Since Outlook is being used by these end users,

it seems justified to graft a spreadsheet-like application to Outlook or to integrate an existing custom mortgage calculator into the Outlook interface, for instance. However, that type of justification is without merit.

The chapter also focused special attention on emails. We learned to create and send emails. When creating emails, it's always a good idea to call the `Resolve` method so that email address can be validated against the Outlook address book for integrity.

One quirk that needs to be pointed out is that Microsoft Outlook add-ins cannot execute with the debugger attached if an instance of Outlook is running on the same machine. The debugger attach process cannot gain access to the running instance of Microsoft Outlook. In order to debug the code, there must be no running instances of Outlook on the desktop initially.

VSTO has eased the burden of application integration with Outlook because it greatly simplifies the process, while providing a solid platform for integration and extensibility. Expect to see more applications bundled in with the Outlook interface in corporate environments.

To be sure, the integration was a last-minute decision. Consequently, the support is provided as an extension to the IDE. However, the internal plumbing does not have a significant impact on the caliber of Microsoft Outlook software that can be built on this platform.

6

The Charting Tool

Today, with a flood of products on the market, it is often difficult to choose an adequate charting package that will suit your budget-conscious needs. To add to the difficulty, most third-party charting packages only incorporate one type of functionality — charts. If you require a mix and match of functionality such as charts with pivot tables or charts with Excel spreadsheets, you will need to purchase and combine different packages. This is a strong case for VSTO-based charting because it is more than a simple chart package. VSTO combines the use of the `Spreadsheet`, `Chart`, `InfoPath`, and `Outlook` object functionality seamlessly under one roof.

This chapter is designed with the required theory and terminology up front. The code is provided after the theoretical portion is complete. You should make every effort to at least be familiar with the charting terms otherwise the going will be rough. For the early parts of the chapter, the code examples will usually be complete, stand-alone applications. However, as we progress, these coding examples will dwindle into code snippets. The chapter will walk you through chart creation and data-loading process. Once you are comfortable putting a basic chart together, the remainder of the chapter will focus on some advance charting aspects to include point manipulation, image and background additions, and legend customizations.

Design-Time Charting

VSTO-based applications can create and render charts that are based on the Microsoft Excel `Chart` object. As previously pointed out, this object can be manipulated at design time as well as runtime. Charts that render at design time allow the developer to get an immediate feel for the data and how it may be displayed ahead of deployment. Such time-saving techniques can actually reduce the time to market for the application. First, I'll spend some time focusing on the design-time aspects.

Chart Wizard

The Chart Wizard is well known to Office developers and end users. It may be less well known to .NET developers. The Chart Wizard is a step-by-step guide to generating charts based on a data source. The Chart Wizard can only source its data from an Excel spreadsheet. The data to be displayed is modeled in real time in the chart engine.

The Chart Wizard is available in the standard toolbar in Microsoft Excel. The Chart Wizard is not available for Microsoft Outlook-based applications or for Microsoft Word applications. Figure 6-1 shows the Chart Wizard icon on the standard toolbar.

Chart Wizard icon

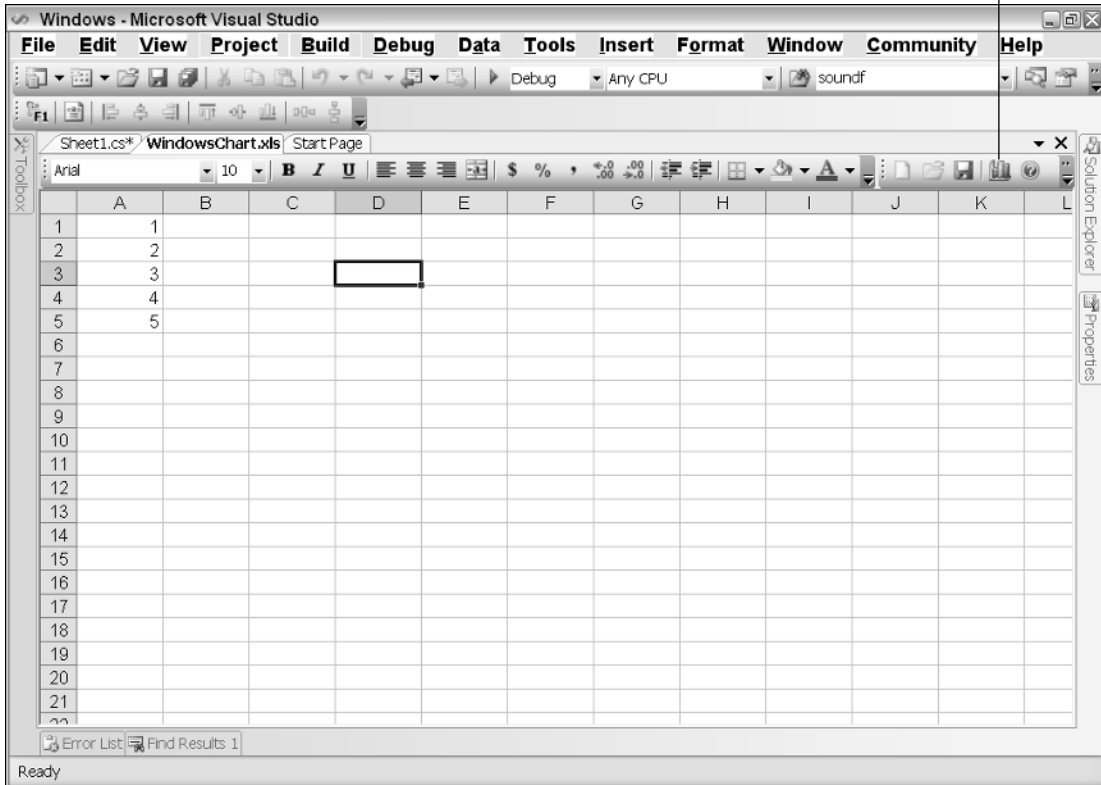


Figure 6-1

The next few examples will focus on manipulating the Chart Wizard, so you can gain a feel for the various objects and the influence they exert on the chart object. Once you are comfortable manipulating the chart in design mode, the rest of the chapter is dedicated to manipulating charts at runtime. Most of the manipulation will exactly reproduce the design-time features. But the runtime manipulation will also add functionality that is not present in the design-time framework.

Charting Nomenclature

Charting terminology can seem foreign at times, since it is based on terms that are unfamiliar to the .NET developer who is without Microsoft Office development experience. However, understanding this terminology is of fundamental importance if you intend to build charting applications. In fact, the Chart API absolutely assumes that the developer has more than a passing knowledge of the nomenclature. If you do not possess this knowledge, you may waste valuable time trying to decipher charting errors. In addition, developing applications with more involved charting requirements can make for an unpleasant experience.

Consider the case where a manager requests that a certain line drawn on a chart application be named appropriately. Without a basic understanding of the charting nomenclature, the developer cannot easily find the object responsible for naming the line. This is, by far, the biggest challenge that developers face; that is, relating charting terminology to the actual objects that compose a chart. By the end of the chapter, you will consider such requests trivial to implement.

Let's begin by creating a new Excel project. See Chapter 2 for the details on creating an Excel project from scratch. Once the project is created, enter numeric data in row 1 for columns A1 through E1, as shown in Figure 6-2. Select the range A1 through E1 before continuing, so that the chart engine knows which cells to use to load data.

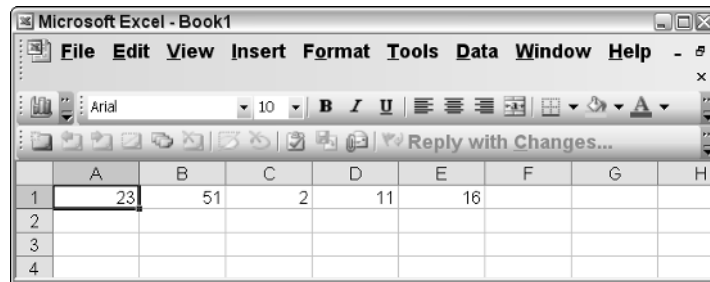


Figure 6-2

Next, open the Chart Wizard by selecting the charting icon on the standard toolbar in the design-time environment of Visual Studio Tools for Office.

The Chart Wizard contains four pages, the first of which is shown in Figure 6-3. Notice the button labeled Press and Hold to View Sample. If you press and hold the button, the Chart Wizard displays a sample column chart based on the column chart selection in the Chart Type window. For these examples, we will retain the default column chart selection. However, you can change the chart type at any time.

Notice also that there are seven subtypes in the right window for the column chart. If you care to do the math, the chart engine can render 75 different chart types straight out of the box. This does not include the custom types that are available in the tab labeled Custom Types in Figure 6-3. Click Next to proceed to page 2, shown in Figure 6-4.

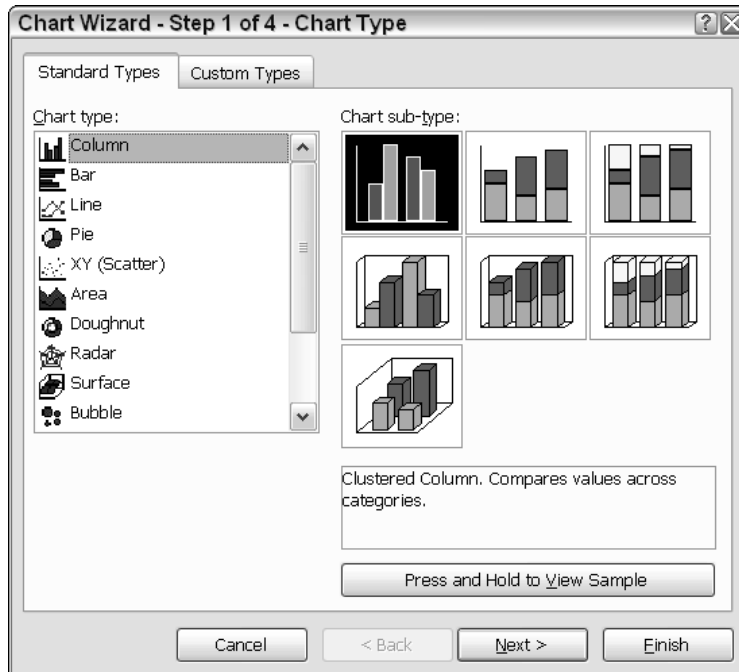


Figure 6-3

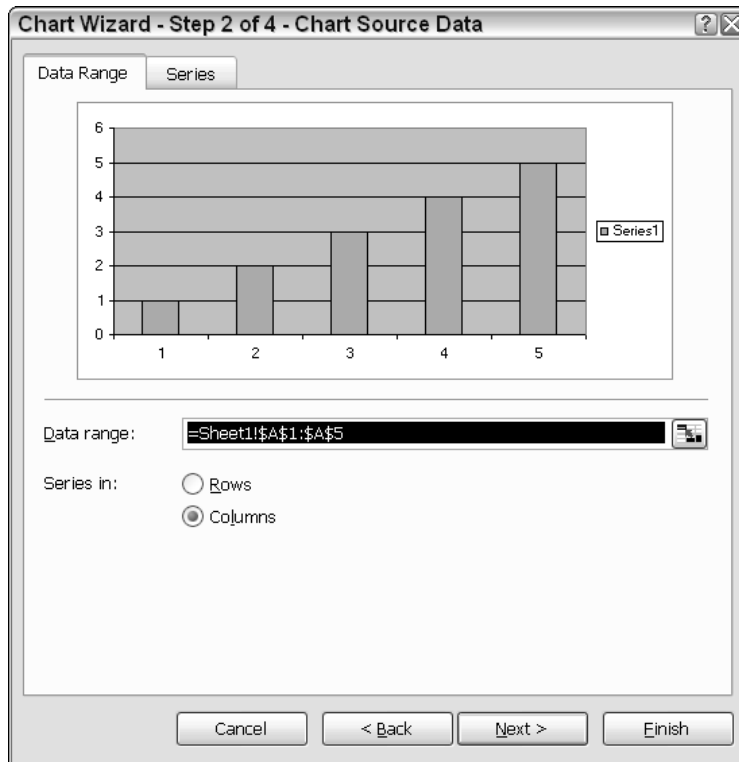


Figure 6-4

Page 2 allows the user to bind the chart data to the cells in the spreadsheet. The Chart Wizard defaults to binding whatever data was selected at the time you clicked the chart button. Notice that the Column radio button is selected. The setting forces the Chart Wizard to use columns as the data source for the Category (X) axes. If you choose the Rows radio button, you will notice that the chart changes to reflect the settings and sources its data from the Category (X) axes from the spreadsheet row. Click Next at the bottom of the property page to proceed to step 3. Page 3 should resemble Figure 6-5.

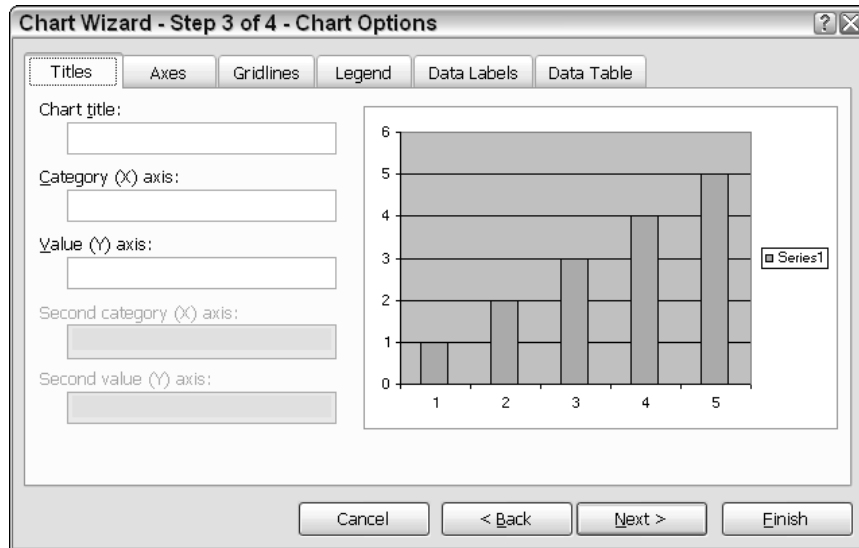


Figure 6-5

These are the important parts of a VSTO-based chart that you must be familiar with in order to develop charting applications. The next few sections explain these terms with practical examples. In the selected titles tab, notice that the rightmost portion contains a column chart displayed inside a rectangular area. The square area is called the chartspace. The chartspace is exposed through a `chartspace` object at runtime. The `chartspace` object holds the chart and the associated objects.

The actual chart is rendered inside the chartspace. This smaller rectangular space is called the chart plot area. The chart plot area represents the actual surface where the chart is being rendered. The chart plot area must necessarily sit inside the `chartspace`. The chart plot area is exposed through the `chartplotarea` object at runtime and is fully customizable. For instance, it is possible to assign different colors or even images to the chart plot area. Several examples of this type of customization will be presented in later portions of the chapter. But for now, let's focus on charting terminology.

Titles

The first tab in Figure 6-5 shows the Title property. The Title property has an associated `title` object in the chart API. Every chart has zero or more titles or captions. The title must be a string identifier. By default, the chart title is displayed prominently at the top of the chart. However, the title can be moved to a different location at runtime. Enter the phrase "This is my Chart Title" in the chart title text box shown in Figure 6-5. Notice that the chart title is updated immediately.

The title you have entered is considered the default title. However, a chart can have as many as five different or similar titles. We will provide details of how to display multiple chart titles on a chart surface later in the chapter.

Axes

The second tab in Figure 6-5 shows the Axes property. A chart must contain at least two axes. The major axes of a chart are the major vertical and horizontal lines displayed in Figure 6-6. In mathematics, the horizontal and vertical chart axes are named X and Y, respectively. However, the Excel charts use a more appropriate naming convention.

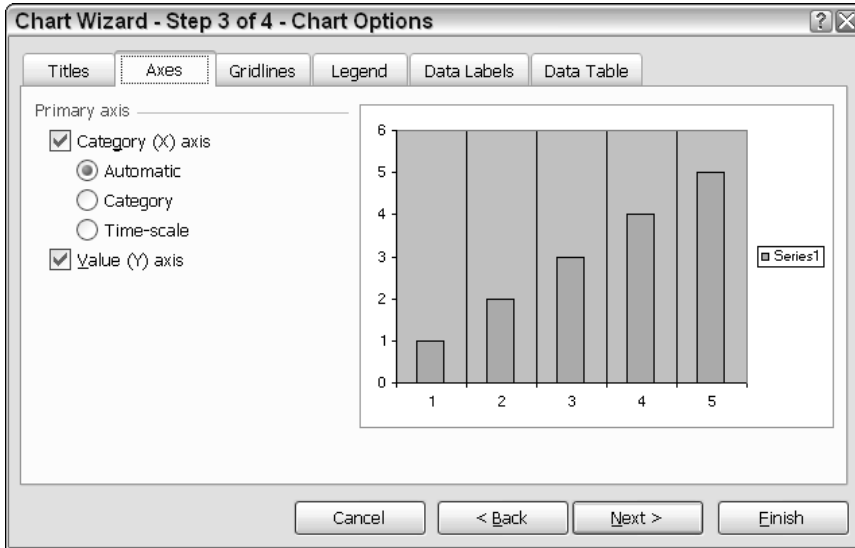


Figure 6-6

The category axis and value axis correspond loosely to the X and Y axes, respectively. The reason for breaking with the mathematical naming convention is that the X and Y axes are necessarily perpendicular to each other. However, Excel can render charts where the axes are not necessarily oriented perpendicularly to each other. For instance, the radar, pie, doughnut, and polar charts do not have conventional X and Y axes. These charts have axes that radiate outward from the center. For these charts, the category and value axes terminology are more appropriate. It may be helpful to examine these charts by clicking the Press and Hold to View Sample button for one of the charts to see the orientation of the axes. We probe these special charts later in the chapter.

You may select and deselect the checkbox shown in Figure 6-6 to see how it affects the chart. Each selection is rendered in real time. Notice that there are three radio buttons. Automatic indicates that the chart will examine the data and try to determine what the data represents before it is used on the category axis. Normally, the data will be numeric. No change is expected for the Category radio button since the data being sourced from the spreadsheet displays automatically to the category axis. If you select Time-scale you will notice that the chart imposes a date/time on the category axes. We talk more about this time-scale feature in a moment.

Time Scaling

Time scaling is a charting feature that implements a time interval on a chart axis. The time-scale algorithm works by first recognizing data input as date/time measurements. Then, the chart makes the appropriate adjustments so that these intervals are suitable to be plotted. Right away, you should observe that quite a lot of data manipulation is involved in the process of adjusting the data to suite specific time intervals. For the most part, and if you are aware of this algorithm, the intervals can generally improve data interpretation and analysis.

Time-scale axes are only available on stock charts, line, column, bar, and area charts. However, the time-scale axes is not available when any of these charts contain category labels that are automatically displayed on more than one line.

Time-scale units are set according to the difference between points on the category axis. It is possible to change this unit to a different unit. Valid units are day, month, and year.

Gridlines

Gridlines represent vertical or horizontal lines that are drawn in the chart plot area in addition to the category and value axis. Gridlines may also be concentric, as in the case of a pie or radar chart. The category axis and the value axis are each considered gridlines as well. Major gridlines differ from minor gridlines in that major gridlines define the major category units on the category axis, as you can see in Figure 6-7. For instance, recall that the chart sourced its data from column A1 through E1. Each column, 1 through 5, forms a major gridline. Minor gridlines fall halfway between each pair of major gridlines. Major gridlines and minor gridlines form an intersecting grid on the chart plot area.

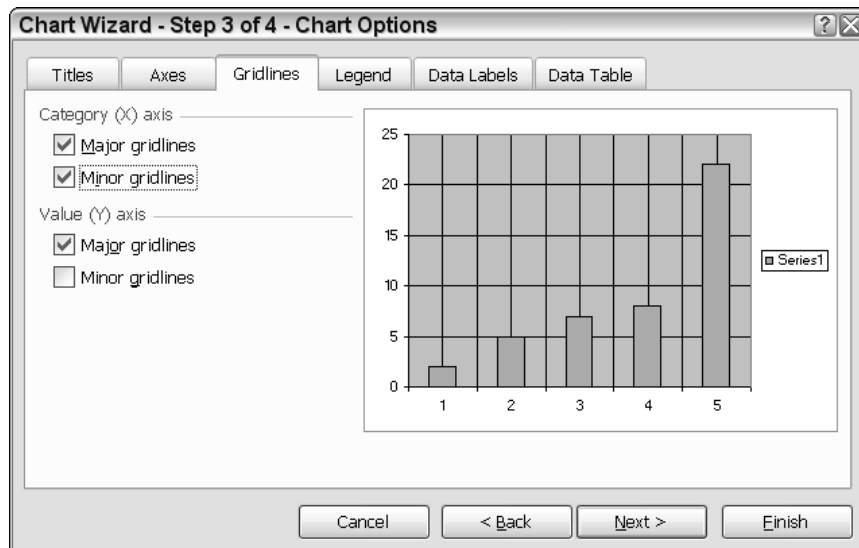


Figure 6-7

You should experiment with the gridlines of a chart by selecting and deselecting the appropriate checkboxes in Figure 6-7. Underneath, gridlines map directly to gridline objects so that it is easy to impose gridlines on or remove them from the chart plot area. However, the gridlines implementation does not allow code to target individual gridlines. Customization applied to the gridline object affects all gridlines on the chart.

Series

A series is a contiguous array of points on a chart. These points represent the data values that are constructed from the category and value axis pairs. When the chart detects that a series of points are available, it immediately assigns a default name to that series and performs some other internal operations on the series that we consider later. The default name or caption represents the title for the series. Later in this chapter, you will learn how to take control of this naming process to impose your own titles on the series.

Each series is represented by a `Series` object. Series objects live in a `Series` collection. In our example, recall that the data being rendered as part of the series was sourced from Row A. On page 2 of the Chart Wizard, Figure 6-4, the series defaulted to columns. This means that there is only one series in the chart represented by row 1. However, if you go back to page 2 of the Chart Wizard and change the setting so that Rows is selected instead of Column, you should see that the chart is updated to show series 1 through 5. The reason for this is that the chart now sources its series data from the columns on the spreadsheet. The five columns containing data are mapped to five series on the chart surface.

Now that you are familiar with the series terminology, you can go back to page 2 of the Chart Wizard, Figure 6-4, and choose the Series tab. Notice that you can add as many series as you want. However, there is a limitation imposed on the number of series on a chart surface. More details are provided at the end of the chapter under “Chart Limitations.” Examine Figure 6-8 for one possible customization.

Before moving on, you should notice that Figure 6-7 contains a data label for category axis. This is simply another one of the five possible titles that a chart can have.

Legend

The legend represents the rectangular area of the chart shown in Figure 6-8 that contains the name `series1`. A legend displays information associated with the lines drawn on the chart plot area. Examine Figure 6-9; by default, the legend is positioned to the right of the chart plot area inside the chartspace.

Notice the various settings for the legend. As you select the different settings, the position of the legend is updated. Observant readers will notice that these positions are 90 degrees apart. It is not possible to position a chart at 15 degrees using the Chart Wizard. However, using code, you can position the chart legend on any part of the chart space.

By default, the legend object holds the default name or legend entry of each series as it is added to the chart plot area. The legend entry contains a key or color-coding that corresponds to the particular series. The legend key is helpful for identifying the different series on the chart. The legend can be customized through code by accessing the `legend` object. A chart can only contain one legend. Even with this limitation, it is possible to draw your own legend on the chart. The advanced section of the chapter will show you how to draw custom objects on the chart surface.

The legend entry and legend key values cannot be assigned directly through the Chart Wizard at runtime or at design time. However, both of these values may be changed after the Chart Wizard has created the chart. An example will be provided later.

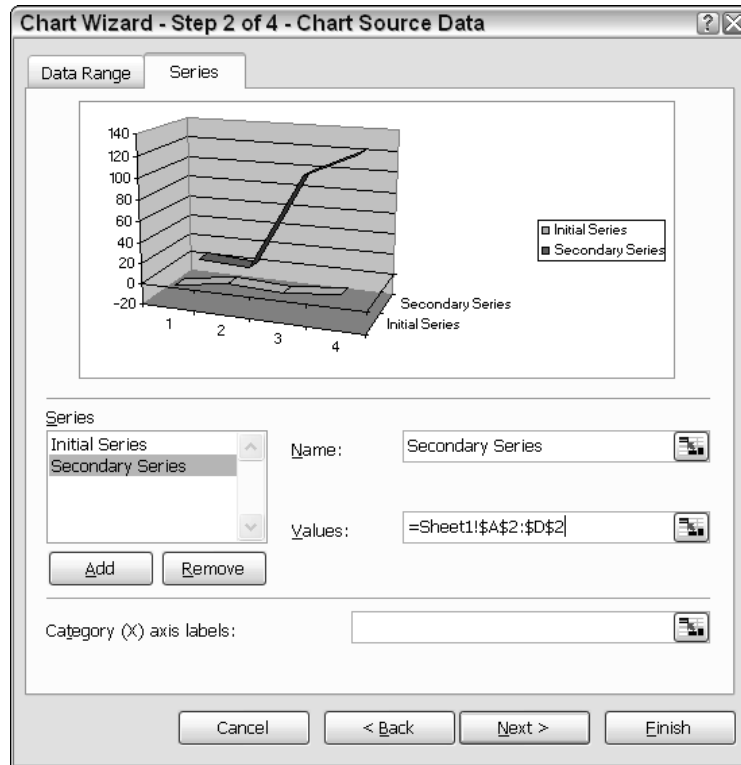


Figure 6-8

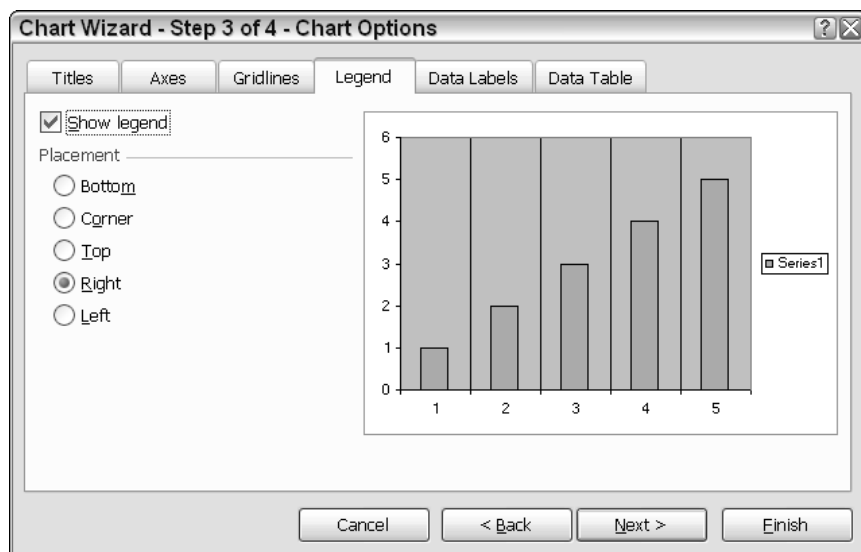


Figure 6-9

Data Labels

Data labels are labels that are imposed on the chart plot area to identify a specific part of the chart. Consider Figure 6-10.

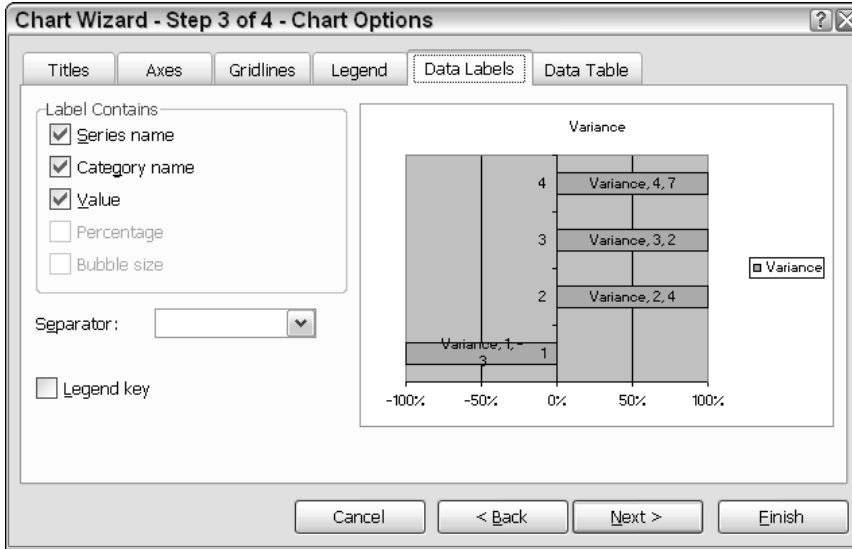


Figure 6-10

As you select each setting, the chart plot area updates to show the default labels that may be displayed. You will notice that with three choices selected, the chart plot area appears cluttered. Information overload through a cluttered chart is just as bad as insufficient information. Always strive to find a good balance with the data being presented on the chart surface.

Data Tables

The last tab on page 3 of the Chart Wizard is the Data Table tab. Consider Figure 6-11 with the selected data table checkbox. Notice that the chart has imposed a table of values immediately beneath the chart.

Data tables provide an easy reference for the data displayed in a chart. Data tables are new for VSTO. For non-VSTO approaches, a data table must be built manually and imposed on the chart surface during the render phase.

The final page of the Chart Wizard allows the user to site the chart either in the existing worksheet or in a new worksheet. All the design-time options are available to runtime code. So let's begin.

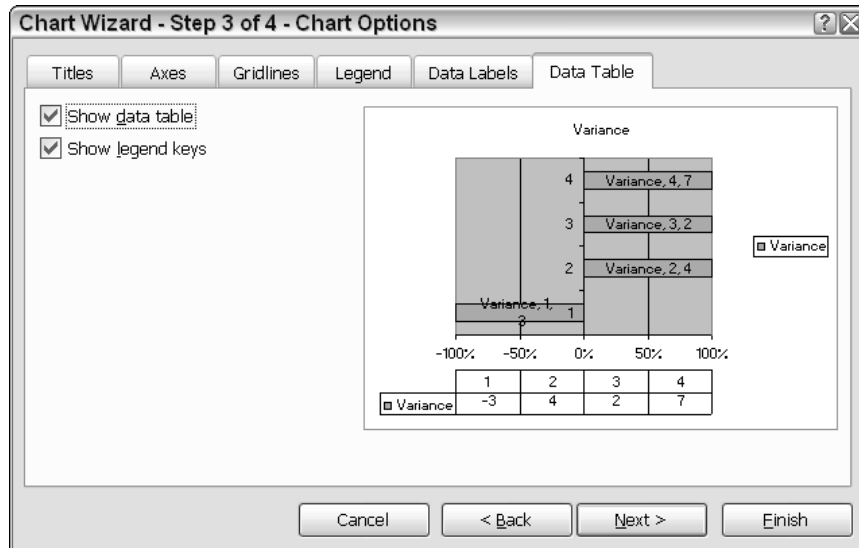


Figure 6-11

Creating VSTO Charts

Charts are implicitly tied to worksheets. In fact, the `Worksheets` object actually exposes a `Chart` object, which represents the default chart in the worksheet. However, you should note that this implicit bond can be broken if you need to generate charts that aren't bound to a worksheet. We provide some strategies for that in a moment. For the vast majority of use cases, charts will be associated with a worksheet.

Whether bound to a worksheet or not, charts are created by adding an object of type `Chart` to the chart collection. The addition is a placeholder and nothing meaningful appears at this point. To finalize the chart display, you'll need to add data to the chart. Let's focus for a moment on the typical case.

Stand-Alone Charts

The `Worksheet` document exposes a `Chart` object. You may create a new chart by calling the `Add` method of the charts collection. Consider Listing 6-1.

Visual Basic

```
Dim xlChart As Excel.Chart = DirectCast(Globals.ThisWorkbook.Charts.Add(),
Excel.Chart)
```

C#

```
Excel.Chart xlChart = (Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing,
missing, missing, missing);
```

Listing 6-1 Creating charts bound to a worksheet

Listing 6-1 shows that the `Add` method accepts four optional parameters. By default, the newly added chart forces the worksheet holding the chart to be the active worksheet. When you create a chart using the `Add` method of the `Charts` collection, the fourth parameter type is implied as `Excel.XlSheetType.xlChart` and you should just pass `missing` to the method call for C#. In fact, specifying the type as `xlChart` will generate an exception. Visual Basic suffers no such inconvenience. As soon as you have added the chart to the `Charts` collection you can begin to customize the chart and load data.

The code in the Visual Basic sections are assumed to run from `sheet1.vb` unless otherwise specified. Visual Basic contains the appropriate references and imports to cause the code to compile successfully. In the case of events, where appropriate, the handlers will be added through code.

Embedded Charts

As the name implies, embedded charts are charts that are created as part of the worksheet. The usual means for creating this type of chart is to use the Chart Wizard. Earlier on in the chapter, we saw that the Chart Wizard can be used to create charts at design time. Listing 6-2 shows how to use the Chart Wizard to create charts at runtime.

Visual Basic

```
Dim ChartObjects as Excel.ChartObjects =
    DirectCast (Me.ChartObjects(), Excel.ChartObjects)
Dim chartObject As Excel.ChartObject = ChartObjects.Add(1,20,250,250)

chartObject.Chart.ChartWizard(Me.Range("A1", "E1"),
    Excel.XlChartType.xl3DColumn, Title:="First Chart")
chartObject.Name = "InitialiChartObject"

ChartObjects = DirectCast (Me.ChartObjects(), Excel.ChartObjects)
Dim chartObject2 As Excel.ChartObject =ChartObjects.Add(251,20,250,250)

chartObject2.Chart.ChartWizard(Me.Range("F1", "K1"),
    Excel.XlChartType.xl3DColumn, Title:="Second Chart")
chartObject2.Name = "SecondaryChartObject"
chartObject2.Activate()
```

C#

```
Excel.ChartObjects ChartObjects =
    (Excel.ChartObjects)this.ChartObjects(missing);
Excel.ChartObject chartObject = ChartObjects.Add(1, 20, 250, 250);

chartObject.Chart.ChartWizard(this.Range["A1", "E1"],
    Excel.XlChartType.xl3DColumn, missing, missing, missing,
    missing, missing, "First Chart", missing, missing, missing);
chartObject.Name = "InitialiChartObject";

ChartObjects = (Excel.ChartObjects)this.ChartObjects(missing);
```

```

Excel.ChartObject chartObject2 = ChartObjects.Add(251, 20, 250, 250);

chartObject2.Chart.ChartWizard(this.Range["F1", "K1"],
    Excel.XlChartType.xl3DColumn, missing, missing, missing,
    missing, missing, "Second Chart", missing, missing, missing);
chartObject2.Name = "SecondaryChartObject";
chartObject2.Activate();

```

Listing 6-2 Creating embedded charts

The type of chart may be set at the series level or at the chart space level. The series level setting overrides the chart space level setting. Some charts, such as the scatter and bubble charts, do not allow you to set the chart type at the series level.

Loading Charts with Data

Earlier on in the chapter, we examined the Chart Wizard. You may recall that the data was loaded and manipulated at design time. However, the chart is capable of loading data at runtime as well. Let's begin by examining some scenarios through code. The first bit of code manipulates the Chart object to provide results similar to those in the exercises that we performed at the beginning of the chapter using the Chart Wizard. The code in Listing 6-3 will load data without the use of the Chart Wizard.

Visual Basic

```

Dim ChartObjects1 As Excel.ChartObjects = DirectCast (Me.ChartObjects(),
Excel.ChartObjects)
    Dim chartObject1 As Excel.ChartObject = ChartObjects1.Add(100, 20, 400,
300)
    Dim rng As Excel.Range
    rng = Me.Application.Range("B1:B5")
    chartObject1.Chart.ChartWizard(rng, Excel.XlChartType.xl3DColumn,
Title:="Custom Chart")
    chartObject1.Name = "myChartObject"
    chartObject1.Interior.Color = ColorTranslator.ToOle(Color.Silver)
    chartObject1.Activate()

```

C#

```

Excel.ChartObjects ChartObjects1 = (Excel.ChartObjects)this.ChartObjects(missing);
Excel.ChartObject chartObject1 = ChartObjects1.Add(100, 20, 400, 300);

chartObject1.Chart.ChartWizard(this.Range["B1", "B5"],
    Excel.XlChartType.xl3DColumn, missing, missing, missing,
    missing, missing, "Custom Chart", missing, missing, missing);
chartObject1.Name = "myChartObject";
chartObject1.Interior.Color = ColorTranslator.ToOle(Color.Silver);
chartObject1.Activate();

```

Listing 6-3 Using the Chart Wizard to render charts

Chapter 6

Listing 6-3 creates a chart that is loaded from data in a spreadsheet range given by B1:B5. If you care to examine the code, you should see that `ChartObjects1` contains a reference to the newly added chart that has been sized appropriately. The size of the `chartspace` area derives its dimensions from the parameters of the `Add` method. However, the dimensions may be easily changed at some later point in time. Note also that if you enter incorrect parameters, a runtime exception will be thrown indicating only that the Chart Wizard method has failed. The error message does not indicate the offending parameter.

The Chart Wizard implicitly uses each column as a series. The setting is equivalent to choosing the first radio button in Figure 6-3. Series terminology will be explained in detail later in the chapter.

At this point, the chart contains no data and will display as an empty square on the spreadsheet. The next step involves invoking the Chart Wizard. The Chart Wizard will handle the manual labor of building the chart from a specified data source. As we mentioned before, the data source is limited to an Excel range. The code in Listing 6-3 also adds a name to the chart object. This name is only available to calling code and does not display on the chart surface.

The Chart Wizard is flexible enough to construct a chart based on an empty range. Notice Figure 6-12 with the black outline around the range B1:B5. That range forms the series in the new chart. As soon as you enter a number, the chart updates to show the new information.

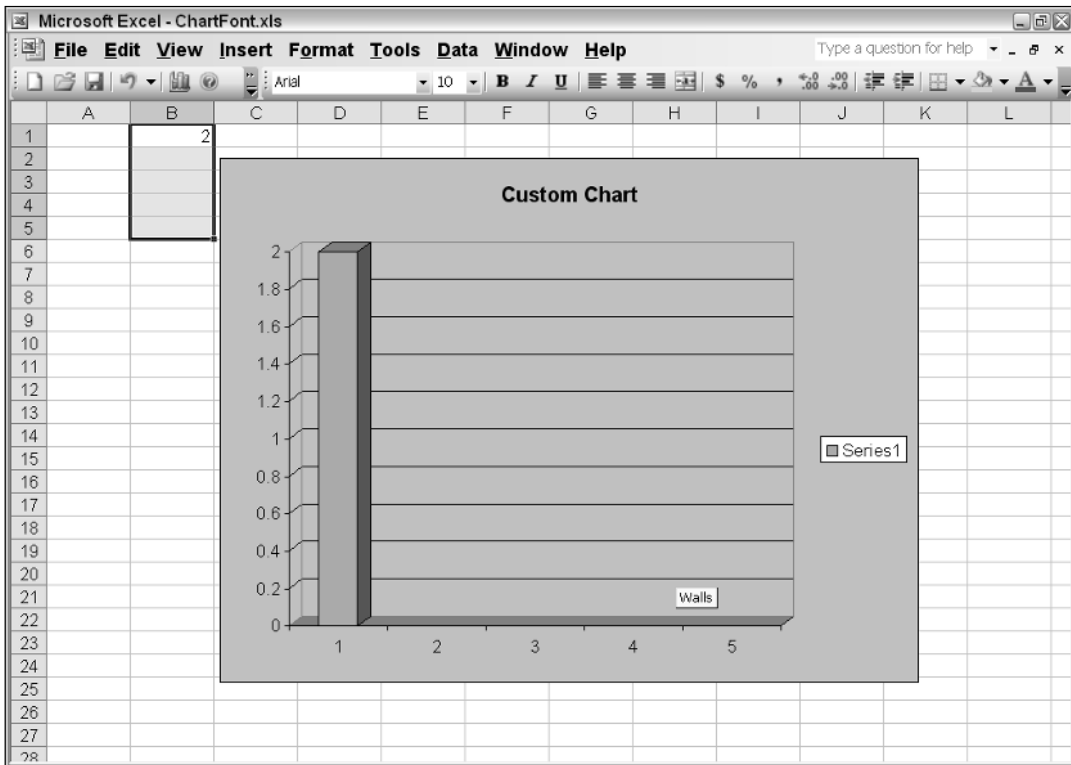


Figure 6-12

If you think about this long and hard, a light bulb should come on. Through this mechanism, the chart is able to respond to data loads in real time. That sort of responsiveness is well suited for real-time applications such as stock market software. And the code to build it is not much different from what is presented here. Simply add a range, and load the data continually into that range. Bind the Chart Wizard to that range, choose an appropriate chart type, and charge the customer an arm and a leg for several months of development effort. At the end of the chapter, you will be able to build that type of application easily.

There is one more quirk here that you should be aware of. The chart simply plots contiguous data that it finds on the spreadsheet. It doesn't necessarily limit its data input to the range that is specified in code. This is certainly problematic because it means that you cannot plot a subset of a range in a spreadsheet; the chart will plot every point. Let's consider a sophisticated example from the dataset here:

A	B	C	D	E
1	11	22	14	4
9	2	7	2	22
6	2	3	3	4
8	4	9	4	3
22	11	7	44	5

Realize that if we simply plot the data by assigning a range A1:E5, we expect to find five series. Each series will contain the points assigned to the specific row. For instance, we expect to find series 1 containing points 1, 11, 22, 14, and 4. Likewise, series 4 should contain points 8, 4, 9, 4, and 3. However, it is possible to exercise some control over the points in this series. Consider Listing 6-4.

Visual Basic

```
Dim xlChart As Excel.Chart = DirectCast(Globals.ThisWorkbook.Charts.Add(),
Excel.Chart)
    Dim cellRange As Excel.Range = Globals.Sheet1.Range("a1", "e5")

    xlChart.SetSourceData(cellRange.CurrentRegion)
    xlChart.ChartType = Excel.XlChartType.xl3DColumn

    Dim oSeries As Excel.Series = DirectCast(xlChart.SeriesCollection(1),
Excel.Series)
    oSeries.XValues = Me.Range("a1", "e1")
    oSeries.Name = "Normal Series 1"

    oSeries = DirectCast(xlChart.SeriesCollection(2), Excel.Series)
    oSeries.XValues = Me.Range("a2", "d2")
    oSeries.Name = "Shortened Series 2"

    oSeries = DirectCast(xlChart.SeriesCollection(3), Excel.Series)
    oSeries.XValues = Me.Range("b1", "b5")
    oSeries.Name = "Awkard"

xlChart.Location(Excel.XlChartLocation.xlLocationAsObject, Me.Name)
```

C#

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace XValue
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
            Excel.Range cellRange = this.Range["a1", "e5"] as Excel.Range;

            xlChart.SetSourceData(cellRange.CurrentRegion, missing);
            xlChart.ChartType = Excel.XlChartType.xl3DColumn;

            Excel.Series oSeries = (Excel.Series)xlChart.SeriesCollection(1);
            oSeries.XValues = this.Range["a1", "e1"];
            oSeries.Name = "Normal Series 1";

            oSeries = (Excel.Series)xlChart.SeriesCollection(2);
            oSeries.XValues = this.Range["a2", "d2"];
            oSeries.Name = "Shortened Series 2";

            oSeries = (Excel.Series)xlChart.SeriesCollection(3);
            oSeries.XValues = this.Range["b1", "b5"];
            oSeries.Name = "Awkard";

            xlChart.Location(Excel.XlChartLocation.xlLocationAsObject, this.Name);
        }

        private void Sheet1_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}
```

Listing 6-4 Series customization from range dataset

The chart dataset is set to A1:E5. However, for series 1, we instruct the chart to source its X values from the range A1:E1. This is the default. We name the series appropriately. For series 2, we instruct the chart to source its X values from the range A2:D2. This is not the default; the range is shorter than the default by one cell, E2. We name the series appropriately. For series 3, we instruct the chart to source its X values from the range B1:B5. This is not the default. We name the series appropriately. The remaining two series are plotted with the default values given by range A4:E4 and A5:E5, respectively.

Unfortunately, for the second series, where the range A2:D2 specifically elides the E2 cell, the chart continues to plot the entire range. The only way to force the chart to respect our wishes is to insert an empty cell at E2. Be aware of this limitation when sourcing from the spreadsheet. It's not particularly pleasant because the action necessitates a change in the original dataset.

For icing on the cake, the last line of code simply locates the chart on the specified sheet. You may use this setting to locate the chart to any other worksheet provided that the worksheet is valid and available. Figure 6-13 shows the code in action.

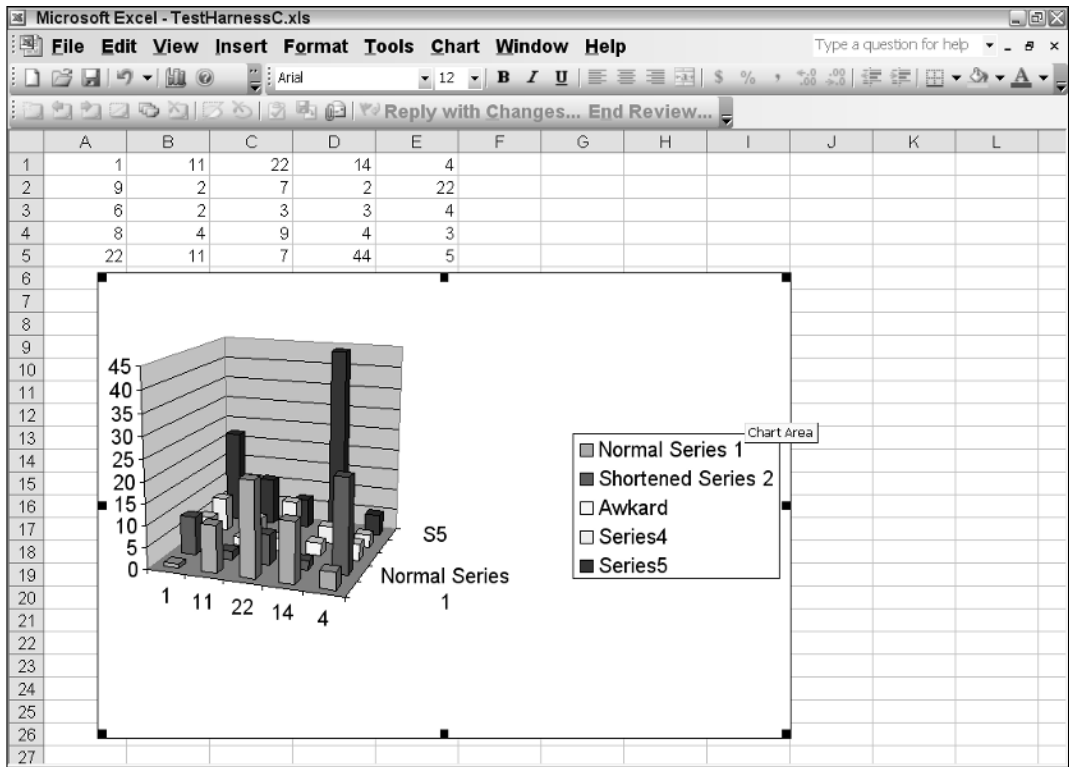


Figure 6-13

Before going on, you should notice that the chart in Figure 6-12 is a 3-D chart. However, there are only two axes. The category and value axes are perpendicular to each other. But you should also notice that the chart has depth that seems to imply a third axis is present. However, this depth is not the result of a third axis. A VSTO chart cannot have three axes. Instead, the perceived depth is automatically created by the chart engine in order to label the series correctly.

Essential Chart Object Hierarchy

Fortunately for the average developer seeking to implement charting functionality in windows applications, only a small part of the `Chart` object hierarchy knowledge is actually required to build robust software. For instance, knowledge of the axes, series, and titles is sufficient to qualify you as a chart expert. The reason behind this claim is that, for a large majority of charting requirement, that is all the knowledge that is actually required. However, it does not mean that you cannot benefit from learning more about the object hierarchy.

Series Manipulation

The `Series` collection implies that a chart may have more than one series. We reserve an in-depth analysis of charts with multiple series for later in the chapter. For now, you should note that the `Series` object may be used to override certain global chart settings. For instance, the chart type that is set at the chart level, may be overridden at the series level so that each series contains its own chart type. In spite of this flexibility, you should be aware that there are limitations to the number of chart types can be imposed on a single chart surface.

The `Series` object also allows for the customization of all things related to the series. For instance, labels, captions, font settings, and colors may be imposed on the series. In addition, trend lines and error bars also form part of the series functionality as well.

You should note that the `SeriesCollection` does not contain a count method. To find the number of series in a chart surface, you will actually need to iterate the collection. This limitation is a VSTO imposed limitation. If you use regular Interop, the series collection does expose a count property. Though the code has been presented to manipulate series in the preceding section, let's formalize it here with the code in Listing 6-5.

Visual Basic

```
Dim oSeries As Excel.Series = DirectCast (xlChart.SeriesCollection(1),  
Excel.Series)  
oSeries.XValues = Me.Range("a1", "e1")  
oSeries.Name = "Normal Series 1"
```

C#

```
Excel.Series oSeries = (Excel.Series)xlChart.SeriesCollection(1);  
oSeries.XValues = this.Range["a1", "e1"];  
oSeries.Name = "Normal Series 1";
```

Listing 6-5 Series manipulation

While other Excel programming surfaces allow a series to be made up of an infinite number of points, VSTO-based series can only contain a finite number of points. This number is smaller for 3-D charts and larger for 2-D charts. The series is key to unlocking the rich feature set of the Excel chart.

Axes and Scaling

A formal definition of a chart axis has already been presented. Code has also shown how to retrieve references to particular axes using an index reference. However, it is also possible to retrieve an axis through a name.

One important reason for using a name instead of an index is that the index may change based on the type of chart and the manipulation of the series. This change may invalidate the index being used. However, a named reference is guaranteed to be correct. One additional benefit is that the code is much more readable. Consider the code in Listing 6-6 that shows how to use a named reference.

Visual Basic

```
Dim axisScale As Excel.Axis =
CType(Globals.ThisWorkbook.ActiveChart.Axes(Excel.XlAxisType.xlValue, Excel.XlAxisGroup.xlPrimary), Excel.Axis)
```

C#

```
Excel.Axis axisScale =
(Excel.Axis)Globals.ThisWorkbook.ActiveChart.Axes(Excel.XlAxisType.xlValue,
Excel.XlAxisGroup.xlPrimary);
```

Listing 6-6 Working with the primary chart axis

In the code snippet presented, the axes parameter supports two parameters. The first parameter indicates the specific axis. For charts with multiple series, the chart code can explicitly request either the primary or the secondary axis.

By default, Microsoft Excel handles unit scaling internally. Consider a data source that spans values from 1 to 10 on the category axis. It's appropriate for each mark to represent a unit, since the width can easily fit on the category axis. However, if the data set includes a range from 0 through 50,000, then some sort of scale must be applied to allow the data to fit on the chart axis.

If you are not content with the internal scaling that Microsoft Excel has imposed, you can easily apply your own unit so that tick marks increment in units of 10,000 for instance. Listing 6-7 shows some code to make this possible.

Visual Basic

```
Dim ChartObjects1 As Excel.ChartObjects = CType(Me.ChartObjects(),
Excel.ChartObjects)
    Dim chartObject1 As Excel.ChartObject = ChartObjects1.Add(100, 20, 400,
300)
    chartObject1.Chart.ChartWizard(Me.Range("A1", "A3"),
Excel.XlChartType.xlLine, Title:="Scale Chart")
    chartObject1.Activate()

    Dim axisScale As Excel.Axis = DirectCast (
chartObject1.Axes(Excel.XlAxisType.xlValue, Excel.XlAxisGroup.xlPrimary),
Excel.Axis)
    If axisScale IsNot Nothing Then
        axisScale.ScaleType = Excel.XlScaleType.xlScaleLogarithmic
        axisScale.MajorUnitIsAuto = False
        axisScale.MajorUnit = 1000
    End If
```

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
    Excel.ChartObjects ChartObjects1 =
(Excel.ChartObjects)this.ChartObjects(missing);
    Excel.ChartObject chartObject1 = ChartObjects1.Add(100, 20, 400, 300);
```

```
chartObject1.Chart.ChartWizard(this.Range["A1", "A3"],
    Excel.XlChartType.xlLine, missing, missing, missing,
    missing, missing, "Scale Chart", missing, missing, missing);
chartObject1.Activate();

Excel.Axis axisScale = (Excel.Axis)
chartObject1.Axes(Excel.XlAxisType.xlValue, Excel.XlAxisGroup.xlPrimary);
if (axisScale != null)
{
    axisScale.ScaleType = Excel.XlScaleType.xlScaleLogarithmic;
    axisScale.MajorUnitIsAuto = false;
    axisScale.MajorUnit = 1000;
}
}
```

Listing 6-7 Scaling automation code

Listing 6-7 first loads data from a `Range` object and sets the type to a line chart. Next, the code retrieves a reference to the primary value axis. In case you were wondering, the `XlAxisGroup` enumeration provides access to the different types of axis that may be found in charts with multiple series. Since we are examining the simple case, the primary axis is returned.

There is one quirk associated with the logarithmic scale. It will tend to distort the values on the value axis. If you consider this an eyesore, one workaround is to set the `MajorUnit` and `MinorUnit` to the same quantity. This will smooth out the scale distortion without affecting the chart accuracy.

With a reference to the value axis, we set the scale type to logarithm. Chart scales may either be linear or logarithmic with the default being linear. When a logarithmic chart is plotted, by default, the value axis will scale in powers of 10. For instance, the units will be equal to 1, 10, 100, 1000, and so on. This is most often appropriate. However, for cases where this isn't appropriate, the code in Listing 6-6 shows how to set the `MajorUnit` to 1000. With that setting applied, every major tick mark will be some multiple of 1000. While it isn't strictly necessary to set the `MajorUnitIsAuto` property to `true`, the code sets it so that you may be aware of this property. By default, when the `MajorUnit` is adjusted through code, the `MajorUnitIsAuto` is automatically set to `false`. You can see the results in Figure 6-14.

The code could have gone on to set the `MinorUnit` as well, but the meaning for such a setting can be derived easily from the `MajorUnit` property. You should note that these settings apply to the value axis. `MinorUnits` applied to the category axis have no effect unless the chart type is set to `timescale`.

Tick Marks

In the typical case, chart data is simply rendered to the chart surface without any sort of customization. While this may work for the majority of cases, there are times when this approach needs to be refined a bit. For instance, consider the case where there are a number of data points to be rendered. By default,

the chart will attempt to fit as many points as necessary on the category axis. If this means increasing the chart width to accommodate the data, then this is performed automatically and no programming intervention is required. However, the end result may be a cluttered chart surface with unreadable data.

One approach to cleaning up the chart surface is to restrict the number of points being rendered. This does not change the accuracy of the chart; it simply allows the chart to plot less axis labels so that the category axis remains uncluttered and readable. The missing values can easily be inferred from the surrounding data.

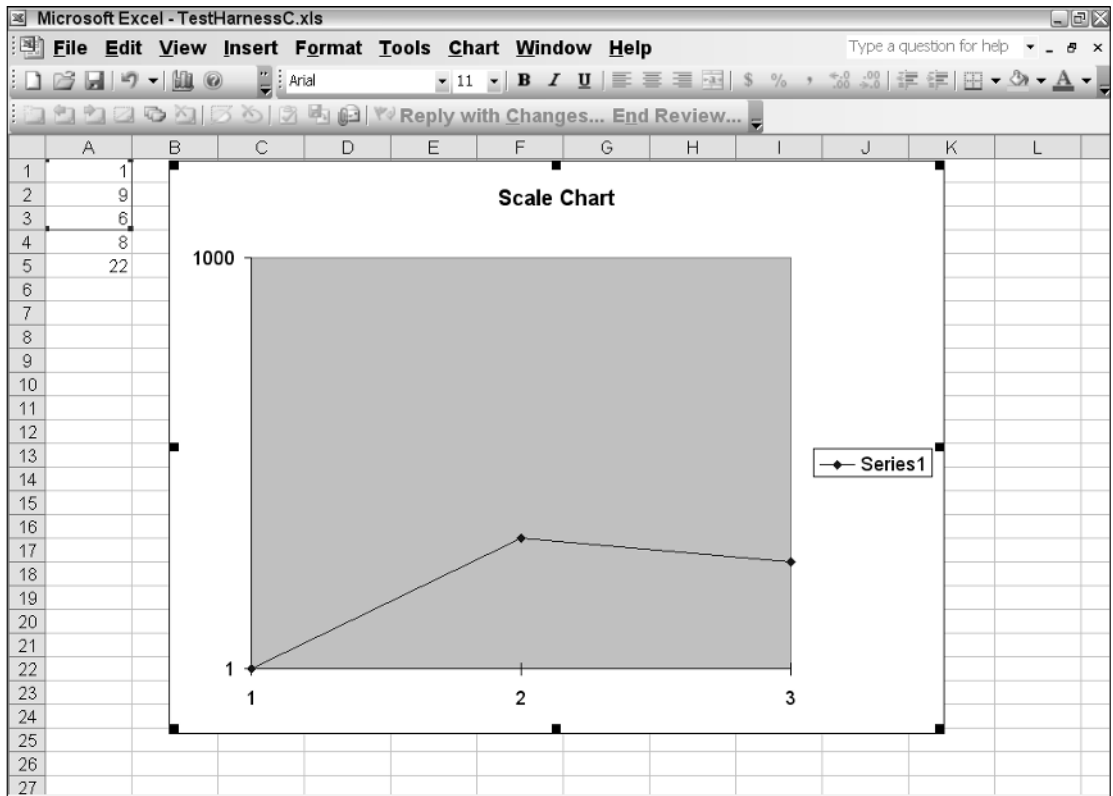


Figure 6-14

An effective way to achieving this customization is to use the tick mark object. A tick mark is a label drawn on a chart to represent the data coordinates being plotted. Figure 6-15, shows an example of tick marks (vertical and horizontal bars on each axis).

Unfortunately, tick marks apply to the chart as a whole. It is not possible to customize an individual tick mark per se. Customizations that apply to the tick mark are fitted to the entire chart. Tick marks can also contain labels, but for the same reasons, these labels apply to the chart as a whole. By default, these labels are derived from the category titles.

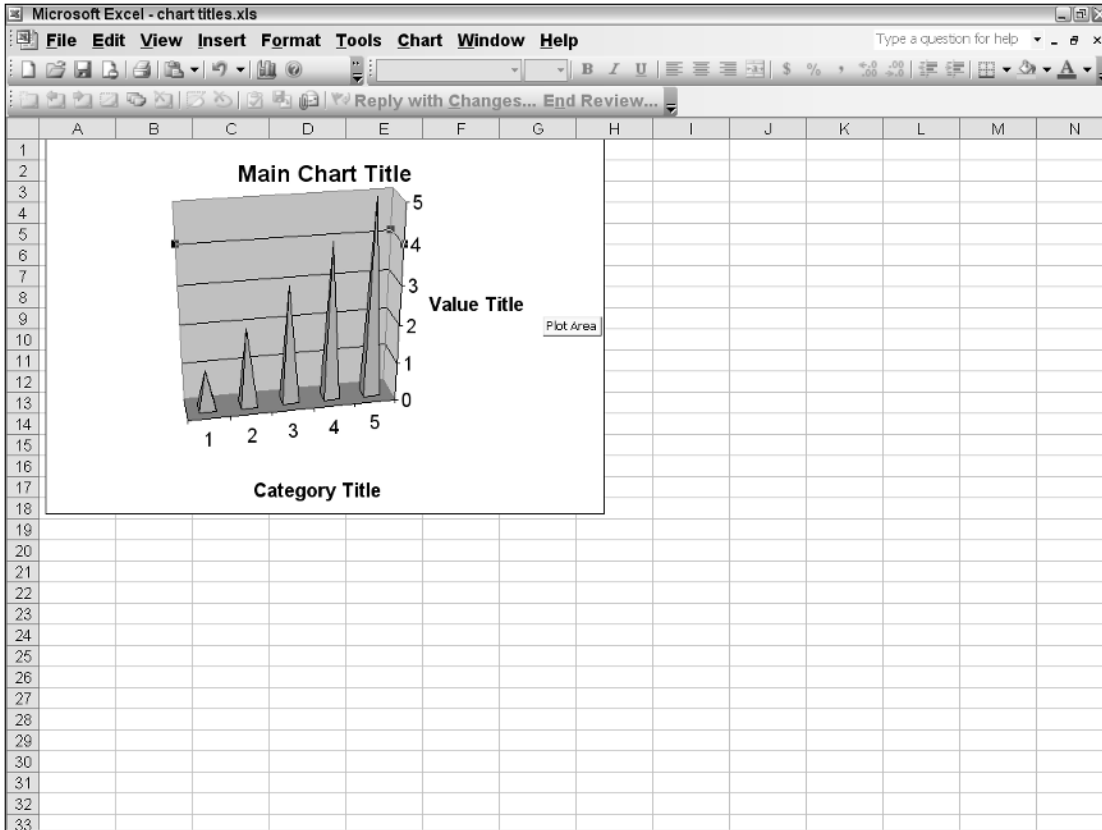


Figure 6-15

Tick marks have an array of associated objects that are usually used in concert to allow for further customization. The `TickLabelSpacing`, `MajorUnit`, `Minimum`, and `Maximum` scale are all used as a singular unit to effect customizations. Listing 6-8 has an example.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
Handles Me.Startup
    Dim ChartObjects1 As Excel.ChartObjects = DirectCast (Me.ChartObjects(),
Excel.ChartObjects)
    Dim chartObject1 As Excel.ChartObject = ChartObjects1.Add(100, 20, 400,
300)
    chartObject1.Chart.ChartWizard(Me.Range("B1", "B5"),
Excel.XlChartType.xlArea, Title:="Tick Chart")
    chartObject1.Activate()

    Dim axisScale As Excel.Axis = DirectCast
(Global.ThisWorkbook.ActiveChart.Axes(Excel.XlAxisType.xlCategory,
Excel.XlAxisGroup.xlPrimary), Excel.Axis)
    If axisScale IsNot Nothing Then
        Dim ticklabels As Excel.TickLabels = CType(axisScale.TickLabels,
Excel.TickLabels)
    End If
End Sub
```

```

        ticklabels.Orientation =
Excel.XlTickLabelOrientation.xlTickLabelOrientationVertical
        ticklabels.Font.Color = ColorTranslator.ToOle(Color.DarkBlue)
        ticklabels.Font.Name = "courier new"
    End If
End Sub

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace Ticks
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.ChartObjects ChartObjects1 =
(Excel.ChartObjects)this.ChartObjects(missing);
            Excel.ChartObject chartObject1 = ChartObjects1.Add(100, 20, 400, 300);
            chartObject1.Chart.ChartWizard(this.Range["B1", "B5"],
                Excel.XlChartType.xlArea, missing, missing, missing,
                missing, missing, "Tick Chart", missing, missing, missing);
            chartObject1.Activate();

            Excel.Axis axisScale =
(Excel.Axis)Globals.ThisWorkbook.ActiveChart.Axes(Excel.XlAxisType.xlValue,
Excel.XlAxisGroup.xlPrimary);
            if (axisScale != null)
            {
                Excel.TickLabels ticklabels =
(Excel.TickLabels)axisScale.TickLabels;
                ticklabels.Orientation =
Excel.XlTickLabelOrientation.xlTickLabelOrientationVertical;
                ticklabels.Font.Color = ColorTranslator.ToOle(Color.DarkBlue);
                ticklabels.Font.Name = "courier new";
            }
        }

        private void Sheet1_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}

```

Listing 6-8 Tick mark manipulation

Chapter 6

Listing 6-7 shows code to customize the tick marks. While the code isn't necessarily complicated, you should notice that a portion of the code reflects some font customization. For instance, the appearance and color may be shaped accordingly. The code in Listing 6-7 also orients the labels vertically on the axis. This approach can be used to prevent clutter on the axes.

The `Microsoft.Office.Interop.Excel.XlTickMark` enumeration allows for some customization on the appearance of tick marks on the chart surface. For instance, the tick marks can be viewed as a cross, placed on the inside of the axis, or removed entirely. By default, the tick marks appear on the outside of the category axis.

To customize the tick marks for the value or series axis, you will need to obtain a reference to the appropriate axis. If you examine the code in Listing 6-7, the `axisScale` variable points to the primary axis. Simply change the reference to point to the required axis to obtain an appropriate reference. The reference will be valid for all axes except the value axis. A runtime exception will be thrown if you attempt to customize the tick marks for the value axis.

Titles and Captions

Earlier material has indicated quite prominently that a chart can have as many as five titles. Let's shed some light on this. Figure 6-16 shows a chart with three titles.

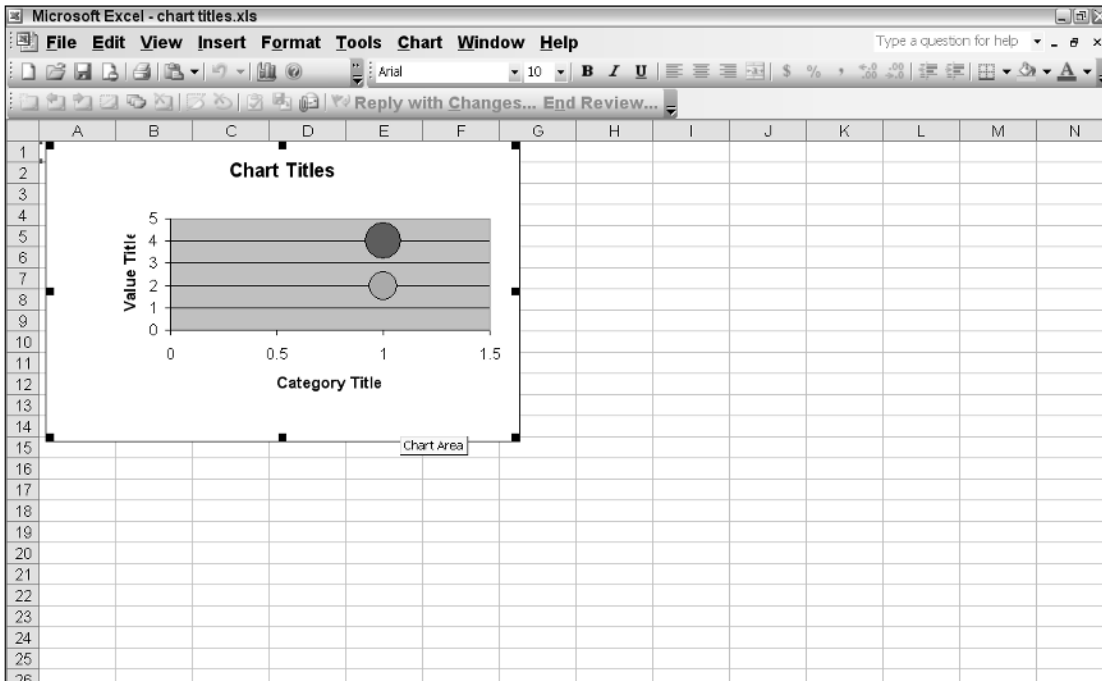


Figure 6-16

Listing 6-9 shows the code to create the chart titles.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
Handles Me.Startup
    Dim ChartObjects1 As Excel.ChartObjects = DirectCast (Me.ChartObjects(),
Excel.ChartObjects)
    Dim chartObject1 As Excel.ChartObject = ChartObjects1.Add(100, 20, 400,
300)
    chartObject1.Chart.ChartWizard(Me.Range("B1", "B5"),
Excel.XlChartType.xlArea)
    chartObject1.Activate()

    chartObject1.HasTitle = True
    chartObject1.Chart.ChartTitle.Caption = "Main Chart Title 1"

    Dim axis As Excel.Axis = DirectCast ( chartObject1
.Axes(Excel.XlAxisType.xlValue, Excel.XlAxisGroup.xlPrimary), Excel.Axis)
    axis.HasTitle = True
    axis.AxisTitle.Caption = "Value Axis Title 2"

    axis = DirectCast ( chartObject1.Axes(Excel.XlAxisType.xlCategory,
Excel.XlAxisGroup.xlPrimary), Excel.Axis)
    axis.HasTitle = True
    axis.AxisTitle.Caption = "Category Axis Title 3"
End Sub
```

C#

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace Ticks
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.ChartObjects ChartObjects1 = (Excel.ChartObjects)this.ChartObjects(missing);
            Excel.ChartObject chartObject1 = ChartObjects1.Add(100, 20, 400, 300);
            chartObject1.Chart.ChartWizard(this.Range["B1", "B5"],
                Excel.XlChartType.xlArea, missing, missing, missing,
                missing, missing, missing, missing, missing, missing);
            chartObject1.Activate();

            chartObject1.HasTitle = true;
            chartObject1.Chart.ChartTitle.Caption = "Main Chart Title 1";

            Excel.Axis axis = (Excel.Axis) chartObject1
            Axes(Excel.XlAxisType.xlValue, Excel.XlAxisGroup.xlPrimary);
            axis.HasTitle = true;
        }
    }
}
```



```
axis.AxisTitle.Caption = "Value Axis Title 2";

axis = (Excel.Axis) chartObject1 .Axes(Excel.XlAxisType.xlCategory,
Excel.XlAxisGroup.xlPrimary);
axis.HasTitle = true;
axis.AxisTitle.Caption = "Category Axis Title 3";
}

private void Sheet1_Shutdown(object sender, System.EventArgs e)
{
}
}
}
```

Listing 6-9 Multi-titled chart

The code is straightforward enough to not require any further explanation. The first three titles may be set through the Chart Wizard constructor as well. However, you may have noticed that we have only showed code to demonstrate three titles. The fourth title applies to charts with multiple series. This option to set chart titles in that scenario is presented in the multiple series section. The fifth title applies to charts that contain time scales and is not presented here.

Chart Groups

Every Microsoft VSTO-based chart application contains one or more chart groups. The `ChartGroup` object is implemented as a collection and represents the individual charts on the chart surface. The following groups are available for use: `AreaGroups`, `BarGroups`, `ColumnGroups`, `DoughnutGroups`, `LineGroups`, and `PieGroups`.

It's more than a programming convenience to use the groups to access the charts rather than retrieving the chart through the global object. There are also performance-based reasons to be had as well. The chart group collection is already available and does not have to incur the overhead of Interop calls to locate and load the respective chart. Listing 6-10 shows some code that manipulates the default chart group.

Visual Basic

```
Private Sub UseChartGroup()
    Dim xlChart As Excel.Chart = DirectCast
(Globals.ThisWorkbook.Charts.Add(), Excel.Chart)
    Dim cellRange As Excel.Range = DirectCast(Me.Range("a1", "e1"),
Excel.Range)
    xlChart.SetSourceData(cellRange, missing)
    xlChart.ChartType = Excel.XlChartType.xlDoughnut
    Dim chartGroup As Excel.ChartGroup = DirectCast
(xlChart.DoughnutGroups(1), Excel.ChartGroup)
    'perform some operation with chartgroup

    xlChart.HasLegend = False
End Sub
```

C#

```
private void UseChartGroup()
{
    Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
    Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;
    xlChart.SetSourceData(cellRange, missing);
    xlChart.ChartType = Excel.XlChartType.xlDoughnut;
    Excel.ChartGroup chartGroup =
(Excel.ChartGroup)xlChart.DoughnutGroups(1);
    //perform some operation with chartgroup

    xlChart.HasLegend = false;
}
```

Listing 6-10 Code to retrieve the default chart group

To be sure, there is nothing particularly amazing in the code implementation. The `chartGroup` method simply provides a programming convenience for code in Listing 6-9. However, there are instances where it may be quite useful. For instance, the `GapWidth` property can only be set through a chart group. The `ActiveChart` object does not expose the `GapWidth` property and neither does the `seriescollection`. The `GapDepth` properties that these objects expose are not functionally equivalent to the `GapWidth` property. Figure 6-17 shows the results of the code in Listing 6-9.

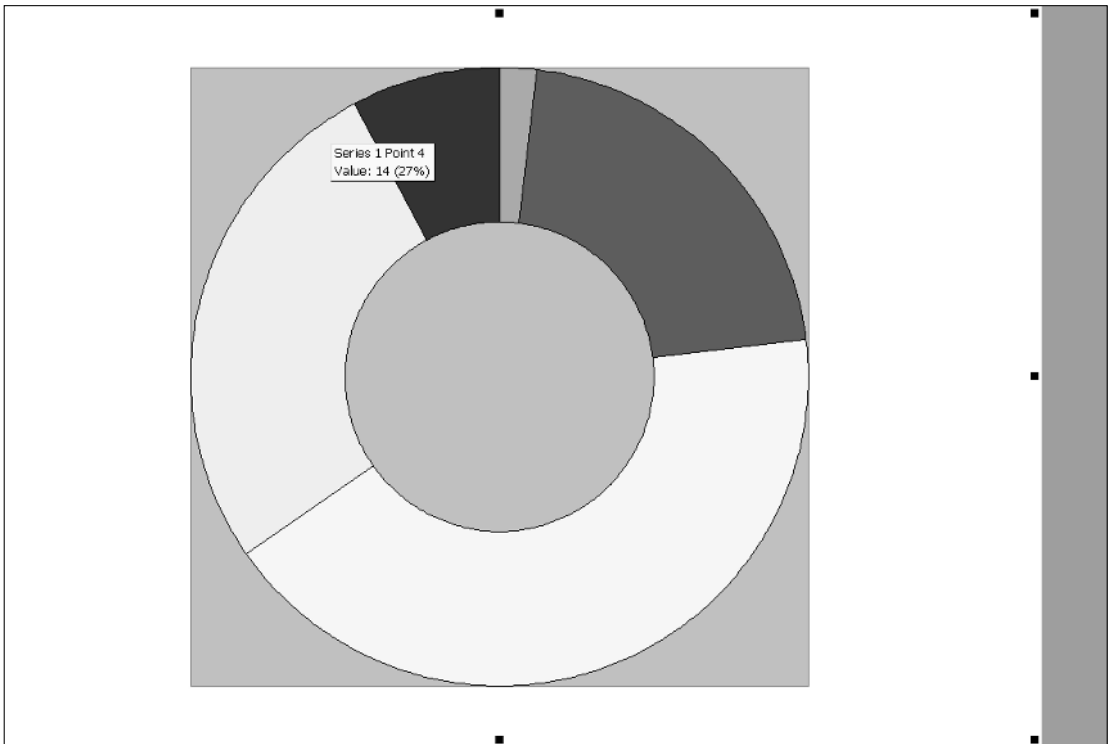


Figure 6-17

Formatting Chart Data

Now that you know how to load data into a chart and are familiar with a few key chart objects, let's focus on the presentation of the data. There is some merit to a chart that has a marginal amount of customization applied. Notice the term *marginal* in there. The implication is that the formatting must never be overdone because it will focus attention away from the data. Assuming that a good balance can be found, formatting can really act to emphasize the important parts of the data. For instance, anomalies can be colored to stand out. And this is what is important in charting.

The chart provides a number of ways to customize data. This section walks you through three main levers of control that may be used. The `font`, `NumberFormat` and `Chart` surface customization encompass most of the common customizations that may be applied to the chart. It bears repeating that overdoing the customization seriously impedes the charts ability to transform data into impact.

Consider Figure 6-18, which shows some formatting applied to the chart surface. Notice that the chart surface is uncluttered and contains only bare-bones formatting. While this is obviously not a route that the hotshot developer may want to adopt, it is a very prudent strategy because the formatting is conservative.

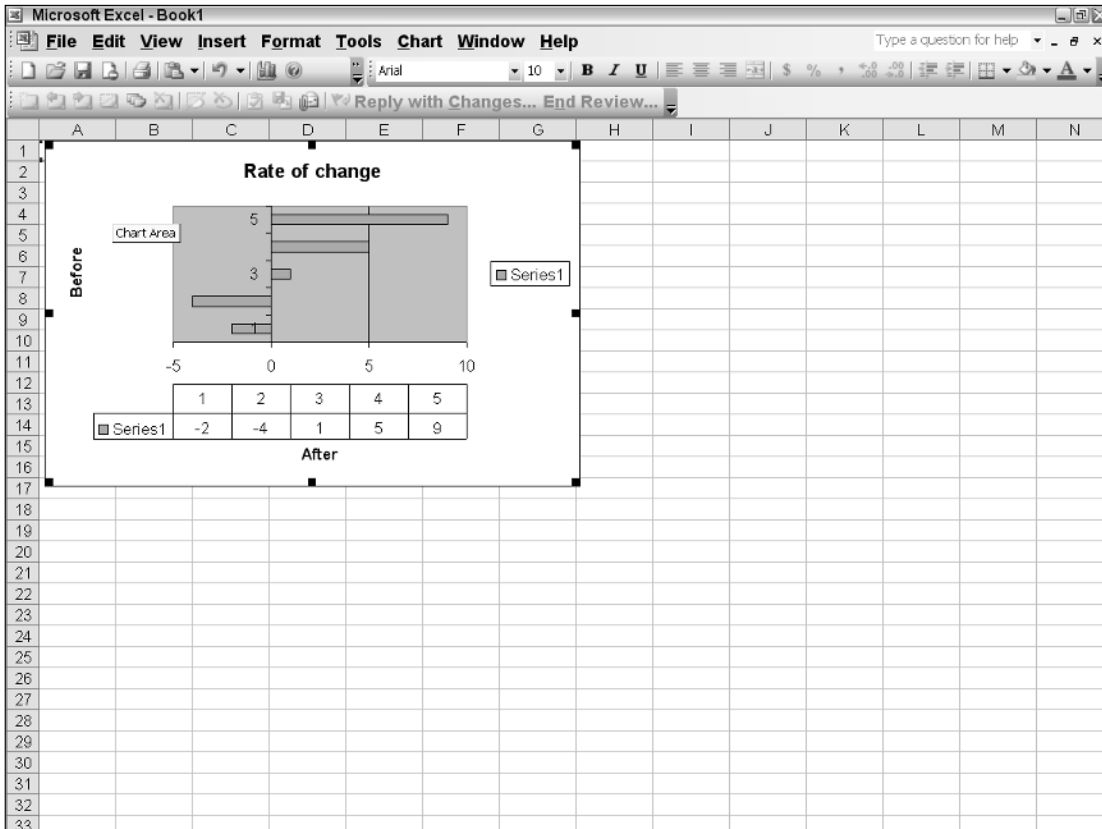


Figure 6-18

In case you were wondering, labels are added, and there is some customization of the legend area. That's about all the customization that is applied. However, you are welcome to add some more customization through events. For instance, you may choose to color the series differently or format the axes appropriately based on end-user events such as mouse clicks. The next few sections show you how to format a chart.

Font Object Customizations

The `font` object allows for font customizations to the target object. The `font` object is associated with many objects, including the `interior` object. Listing 6-11 shows an example.

Visual Basic

```

        Dim ChartObjects1 As Excel.ChartObjects = DirectCast (Me.ChartObjects(),
Excel.ChartObjects)
        Dim chartObject1 As Excel.ChartObject = ChartObjects1.Add(100, 20, 400,
300)

        chartObject1.Chart.ChartWizard(Me.Range("a1", "e1"),
Excel.XlChartType.xl3DColumn)
        Dim font As Excel.Font = chartObject1.Chart.ChartArea.Font
        font.Bold = True
        font.Underline = True
        font.Shadow = True
        font.Italic = True
        font.Color = ColorTranslator.ToOle(Color.Red)

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace ChartFont
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.ChartObjects ChartObjects1 =
(Excel.ChartObjects)this.ChartObjects(missing);
            Excel.ChartObject chartObject1 = ChartObjects1.Add(100, 20, 400, 300);

            chartObject1.Chart.ChartWizard(this.Range["a1", "e1"],
Excel.XlChartType.xl3DColumn, missing, missing, missing,
missing, missing, "Custom Chart", missing, missing, missing);
            Excel.Font font = chartObject1.Chart.ChartArea.Font;
            font.Bold = true;
            font.Underline = true;
            font.Shadow = true;

```

```
        font.Italic = true;
        font.Color = ColorTranslator.ToOle(Color.Red);
    }

    private void Sheet1_Shutdown(object sender, System.EventArgs e)
    {
    }
}
}
```

Listing 6-11 Font object customization

From Listing 6-11, the Chart Wizard is used to load a single series from range A1:E1. What is of interest to us is the reference to the `Font` object. Notice that the `font` object is obtained from the `ChartArea` object. It means that the customization applied in the next few lines of code will affect the entire chart area to include `Labels`, `Titles`, `Legends`, and `Captions`. The rest of the code is trivial to understand.

You should take from this discussion that it is possible to specifically target areas of the chart with font customizations. These customizations can serve as an override for the default settings. If you care to reset the customizations after they have been imposed but before the application has ended, simply call the `clearformat()` method. Once the application has ended, the default settings are restored. If you care to persist your customizations, you will need to implement code to save and load these settings from a persistent data store.

Number Format Customization

The `NumberFormat` object allows an object to apply appearance customizations to the target object. Usually, the customizations affect labels on the axis. You may recall that the `NumberFormat` property was discussed in Chapter 3. Although, it was discussed within the context of the `Excel` object, the principles may be applied to the chart object as well. Proceeding from that earlier discussion, let's examine an example (see Listing 6-12).

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
Handles Me.Startup
    Dim ChartObjects1 As Excel.ChartObjects = DirectCast (Me.ChartObjects(),
Excel.ChartObjects)
    Dim chartObject1 As Excel.ChartObject = ChartObjects1.Add(100, 20, 400,
300)
    chartObject1.Chart.ChartWizard(Me.Range("B1", "B5"),
Excel.XlChartType.xlArea, Title:="Formatter Chart")
    chartObject1.Activate()

    Dim axisScale As Excel.Axis = DirectCast ( chartObject1
.Axes(Excel.XlAxisType.xlCategory, Excel.XlAxisGroup.xlPrimary), Excel.Axis)
    If Not axisScale Is Nothing Then
        axisScale.HasTitle = True
        Dim ticklabels As Excel.TickLabels = DirectCast (axisScale.TickLabels,
Excel.TickLabels)
        ticklabels.Orientation =
Excel.XlTickLabelOrientation.xlTickLabelOrientationHorizontal
    End If
End Sub
```

```

        ticklabels.NumberFormat = "#.0##"
    End If
End Sub
C#
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace Ticks
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.ChartObjects chartObjects1 =
(Excel.ChartObjects)this.ChartObjects(missing);
            Excel.ChartObject chartObject1 = chartObjects1.Add(100, 20, 400, 300);
            chartObject1.Chart.ChartWizard(this.Range["B1", "B5"],
                Excel.XlChartType.xlArea, missing, missing, missing,
                missing, missing, "Formatter Chart", missing, missing, missing);
            chartObject1.Activate();

            Excel.Axis axisScale = (Excel.Axis) chartObject1
.Axes(Excel.XlAxisType.xlValue, Excel.XlAxisGroup.xlPrimary);
            if (axisScale != null)
            {
                axisScale.HasTitle = true;
                Excel.TickLabels tickLabels =
(Excel.TickLabels)axisScale.TickLabels;
                tickLabels.Orientation =
Excel.XlTickLabelOrientation.xlTickLabelOrientationHorizontal;
                tickLabels.NumberFormat = "#.0##";
            }
        }

        private void Sheet1_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}

```

Listing 6-12 Number format customization

From the code in Listing 6-12 and the results in Figure 6-19, observe that the vertical axis contains labels that are oriented horizontally. These values are also formatted with a decimal point. You should notice also that the customization is applied to the category axis. It is possible to customize the value axis and series axis as well by supplying the appropriate enumeration to the first parameter of the axes method.

The `NumberFormat` must be applied only to labels or pieces of text that are rendered on the chart surface. Not all target objects expose a `NumberFormat` property. However, axes and data labels usually expose this property. For a more detailed probe of the `NumberFormat` and the possible options that may be used with the formatter, see Chapter 3.

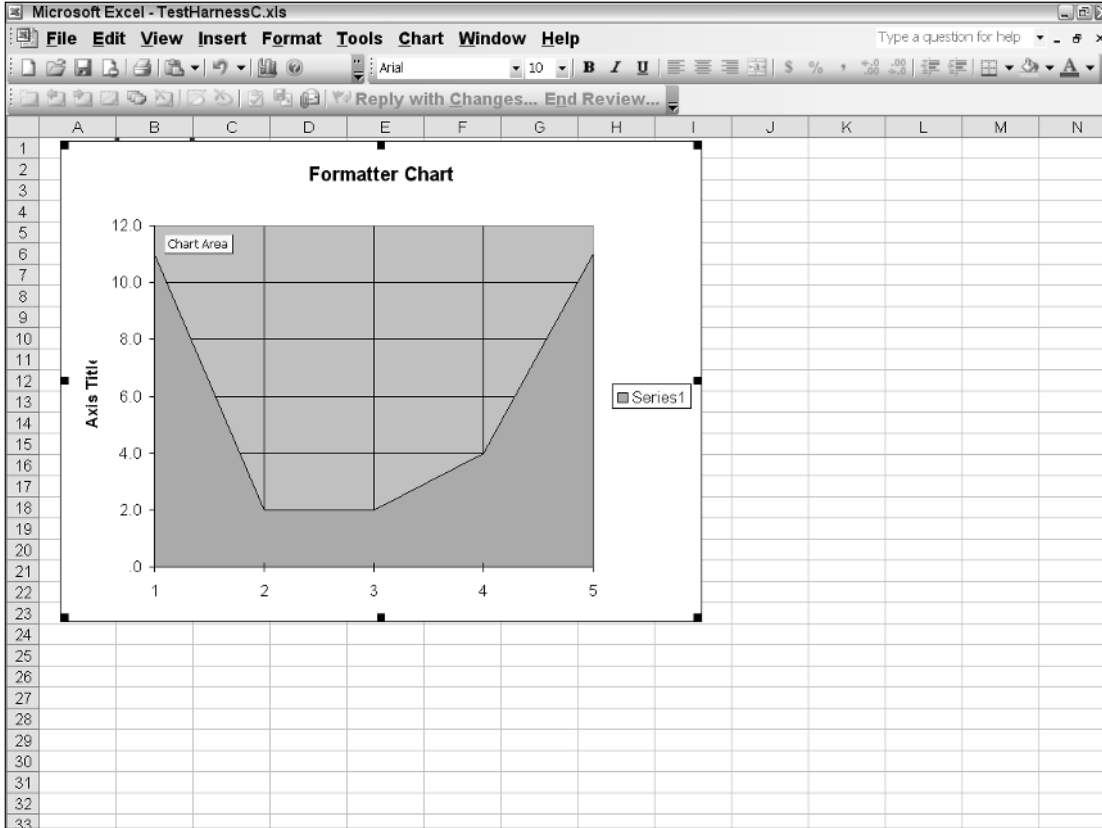


Figure 6-19

Chart Surface Customization

The walls of the chart are fully customizable. You can add backgrounds, fonts, and patterns to these surfaces. The `chart` object also supports a number of default patterns and backgrounds that may be used. Unfortunately, this area of customization is prone to abuse. It's quite common to view charts with gaudy backgrounds that are distracting and conspire to focus attention away from the data.

Consider Listing 6-13, which will customize the surface area of the walls of the chart.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Handles Me.Startup
        'xlChart.Location(Excel.XlChartLocation.xlLocationAsObject, Me.Name)
        Dim xlChart As Excel.Chart = DirectCast(Globals.ThisWorkbook.Charts.Add(),
        Excel.Chart)
        Dim cellRange As Excel.Range = Me.Range("a1", "a5")

        xlChart.SetSourceData(cellRange.CurrentRegion)
```

```

xlChart.ChartType = Excel.XlChartType.xl3DBarClustered

'set the pattern on the wall
Dim wall As Excel.Walls = xlChart.Walls
Dim interior As Excel.Interior = wall.Interior
interior.Pattern = Excel.XlPattern.xlPatternLightUp
interior.Color = ColorTranslator.ToOle(Color.White)
'add color to the floor
Dim floor As Excel.Floor = xlChart.Floor
Dim floorInterior As Excel.Interior = floor.Interior
floorInterior.Color = ColorTranslator.ToOle(Color.Yellow)

End Sub

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace Perspectives
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
            Excel.Range cellRange = this.Range["a1", "a5"] as Excel.Range;

            xlChart.SetSourceData(cellRange.CurrentRegion, missing);
            xlChart .ChartType = Excel.XlChartType.xl3DBarClustered;

            //set the pattern on the wall
            Excel.Walls wall = xlChart .Walls;
            Excel.Interior interior = wall.Interior;
            interior.Pattern = Excel.XlPattern.xlPatternLightUp;
            interior.Color = ColorTranslator.ToOle(Color.White);
            //add color to the floor
            Excel.Floor floor = xlChart .Floor;
            Excel.Interior floorInterior = floor.Interior;
            floorInterior.Pattern = Excel.XlPattern.xlPatternGray16;
            floorInterior.Color = ColorTranslator.ToOle(Color.Yellow);
        }

    }

    private void Sheet1_Shutdown(object sender, System.EventArgs e)
    {
    }

}
}

```

Listing 6-13 Chart surface customization

The code in Listing 6-13 produces Figure 6-20.

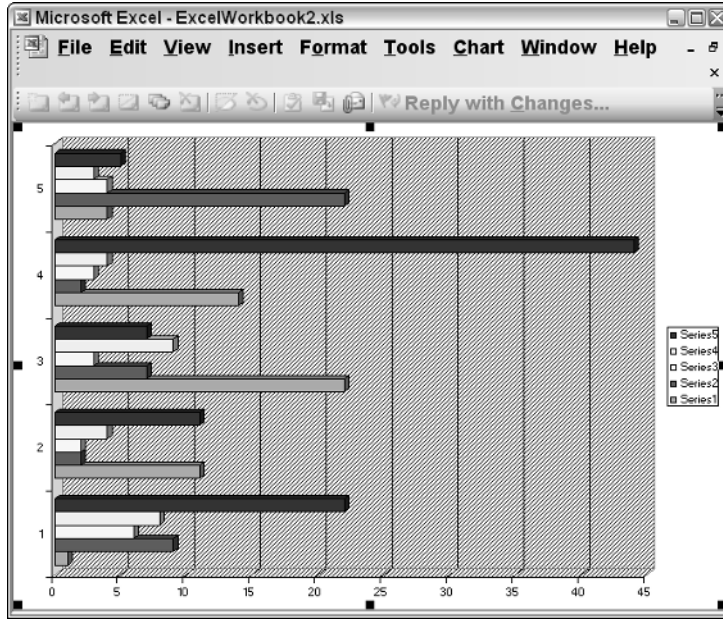


Figure 6-20

Listing 6-12 shows code to load data into the chart. Once the chart has been loaded with data, the chart type is set. With the necessary plumbing out of the way, a reference is obtained pointing to the walls of the chart. The wall is the vertical background of the chart not including the floor or ceiling of the chart. A predefined pattern is applied to the wall, and it is colored white.

Next, a reference for the floor object is obtained. The color is set to yellow. We could have added a pattern to the floor as well, but the contrast in patterns would probably be distracting. Because the chart is a clustered bar chart, it appears flipped on its side. So, the floor of the chart is not necessarily found at the bottom of the chart space. Further, the chart floor only applies to 3-D drawings since, there is perceived depth. Floor customizations for 2-D charts will fail. You should note again that the background is extremely important because the data essentially sits on that surface. If it is too loud, it will distract from the task at hand. The `Excel.XlPattern` enumeration contains 20 default patterns that may be applied as customizations. The chart does not allow you to add a custom pattern. However, you can work around this limitation by adding your custom pattern to an image and loading the image into the chart. An example of this is provided later.

Chart Legend Manipulation

A formal definition of a chart legend has already been presented at the start of the chapter. Expanding from this definition, the legend feature is exposed in a `Legend` object. Since the legend belongs to the chart as a whole and not to a particular series, you should expect to access this `Legend` object through the chart object.

The legend object contains legend entries. Legend entries are exposed through a `LegendEntries` object. It's helpful to note that `LegendEntries` are added to the legend by default as each chart series is being rendered. The entries follow the order of the series addition. Because each `LegendEntries` and `Legend` object contain formatting objects, it is possible to customize the appearance of each entry in the legend. The appearance and position of the legend can also be customized. Finally, the order of the legend entries may be overridden through code.

First, let's focus on a customer requirement. The customer would like the ability to apply customizations at runtime to a chart. The customization allows the end user to determine where to place an existing legend on the chart surface. The border and attributes must be customizable. Additionally, the end user must have the ability to change the background of the chart and impose font attributes on the chart title. The code in Listing 6-14 implements these end-user requirements.

Visual Basic

```
Dim xlChart As Excel.Chart = DirectCast (Globals.ThisWorkbook.Charts.Add(),
Excel.Chart)

    Dim cellRange As Excel.Range = Me.Range("a1","e1")
    xlChart.SetSourceData(cellRange.CurrentRegion)
    xlChart.ChartType = Excel.XlChartType.xl3DBarClustered

    xlChart.HasLegend = True
    Dim legend As Excel.Legend = xlChart.Legend
    legend.Position = Excel.XlLegendPosition.xlLegendPositionLeft
    legend.Shadow = True
    legend.Interior.Color = ColorTranslator.ToOle(Color.WhiteSmoke)

    'reposition the legend to fall inside the chart
    legend.Height += legend.Height
    legend.Width += legend.Width
    legend.Left += 140

    'customize the legend entry
    Dim enTry As Excel.LegendEntry = DirectCast(legend.LegendEntries(1),
Excel.LegendEntry)
    enTry.Font.Underline = True
    enTry.Font.Name = "Courier New"
    enTry.Font.Color = ColorTranslator.ToOle(Color.Blue)
```

C#

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;
using System.Globalization;

namespace Legend
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
```

```
Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;

xlChart.SetSourceData(cellRange.CurrentRegion, missing);
xlChart.ChartType = Excel.XlChartType.xl3DBarClustered;

Globals.ThisWorkbook.ActiveChart.HasLegend = true;
Excel.Legend legend = xlChart.Legend;
legend.Position = Excel.XlLegendPosition.xlLegendPositionLeft;
legend.Shadow = true;
legend.Interior.Color = ColorTranslator.ToOle(Color.WhiteSmoke);

//reposition the legend to fall inside the chart
legend.Height += legend.Height;
legend.Width += legend.Width;
legend.Left += 140;

//customize the legend entry
Excel.LegendEntry entry = legend.LegendEntries(1) as Excel.LegendEntry;
entry.Font.Underline = true;
entry.Font.Name = "Courier New";
entry.Font.Color = ColorTranslator.ToOle(Color.Blue);

}

private void Sheet1_Shutdown(object sender, System.EventArgs e)
{
}
}
```

Listing 6-14 Chart legend manipulation

The code in Listing 6-14 shows that manipulation of the legend involves obtaining a reference to the `Legend` object. However, before proceeding, you must always set the `HasLegend` property to `true`; otherwise, a runtime exceptions will be thrown. By default, the `Legend` enumeration position allows the legend to be set to specific quadrants on the chart surface. However, the code shows that the left property can be used to move the entire legend within the specified quadrant. In our case, we have moved the legend to an area inside the chart plot area (see Figure 6-21). This is only for illustrative purposes; the legend should remain outside of the chart plot area.

Finally, the code retrieves the first legend entry. By default, the legend entries are added in the order that the series are rendered on the chart surface. In our case, we simply impose some formatting on the first entry. The legend entry cannot be removed or hidden. For extra credit, we double the area of the legend and add a shadow. The customization gives you a foot holder if you decide to add cosmetic touchups to the legend object.

As the text has pointed out, the legend entry order is predetermined based on the order that each series holds in the series collection. However, you may want to impose your own order on the legend entries. This is not a particularly daunting task. It simply involves reordering your data set so that the legend entries are rendered in the correct order.

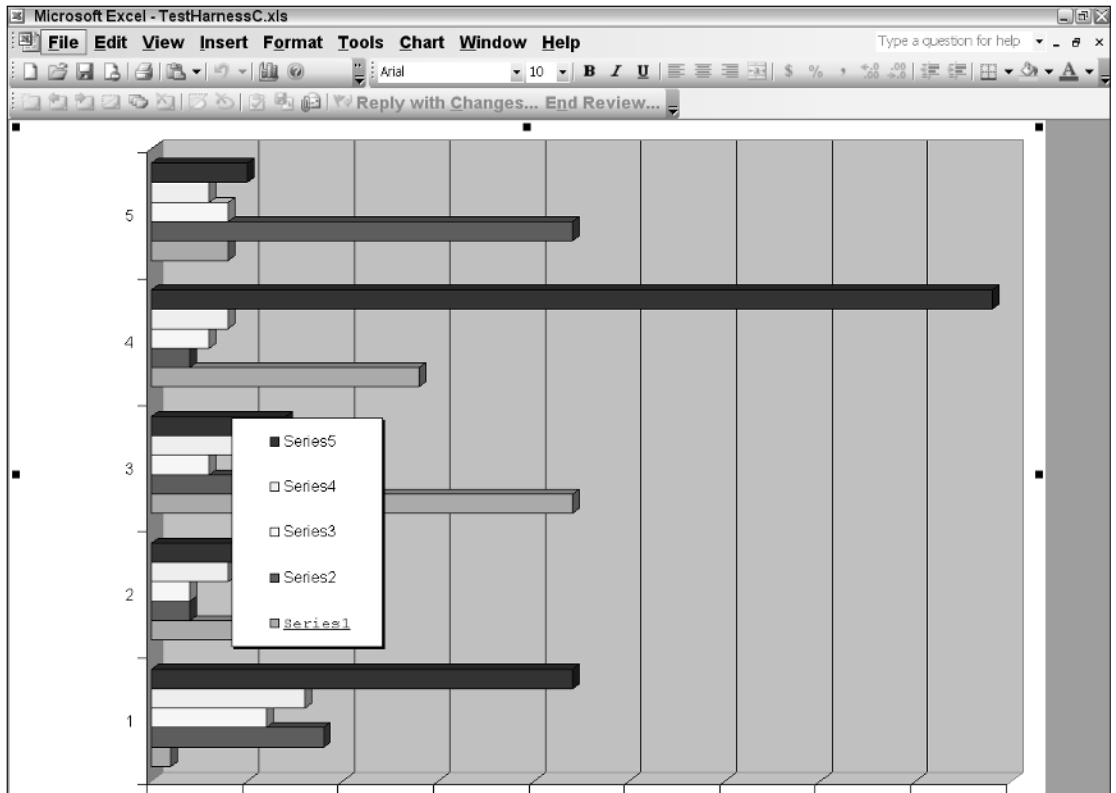


Figure 6-21

You should note also that marker styles (the small colored squares next to the legend entries in the legend) are automatically embedded in the legend keys. For instance, if you apply a diamond marker style to the series, the legend key will automatically be updated to reflect the new shape. There is no way to customize this appearance through code. We focus on marker objects in a later section.

Analyzing Data

A chart transforms data into impact. Behind this impact, there may be trends, downturns, eroding profit margins, or new business opportunities. Although that type of information may be elusive in raw data, a chart can immediately surface these concerns to decision makers. From this point, a decision maker can transform this concern into action. Therefore, it is fundamentally important that a chart facilitate the data evaluation process.

Charts that obstruct this process are distracting and counterproductive because they tend to focus attention on unimportant areas. For that reason, it is important to choose the right type of chart. More often than not, the developer cannot fully appreciate the significance and context of the data and may choose to display the raw data in an unsuitable chart type. However, such inconveniences are easily sterilized by providing the end user with the option to use different charts to display the data.

The ability to choose different chart types is simply one mechanism of data analysis. VSTO provides others, such as the trend line and error bar functionality. These features allow the end user to examine the data for consistency and even make predictions and assumptions based on the state of the data. The next few sections examine these concepts in more detail.

Trending through Trend Lines

A trend line is a line drawn on a chart that indicates a trend in the data. For VSTO-based charts, the trend line functionality is exposed through the `Trendline` object. The feature set is not necessarily limited to a straight line. The trend line may be a curve or derived from a logarithmic expression. Figure 6-22, shows a curved trend line on a chart.

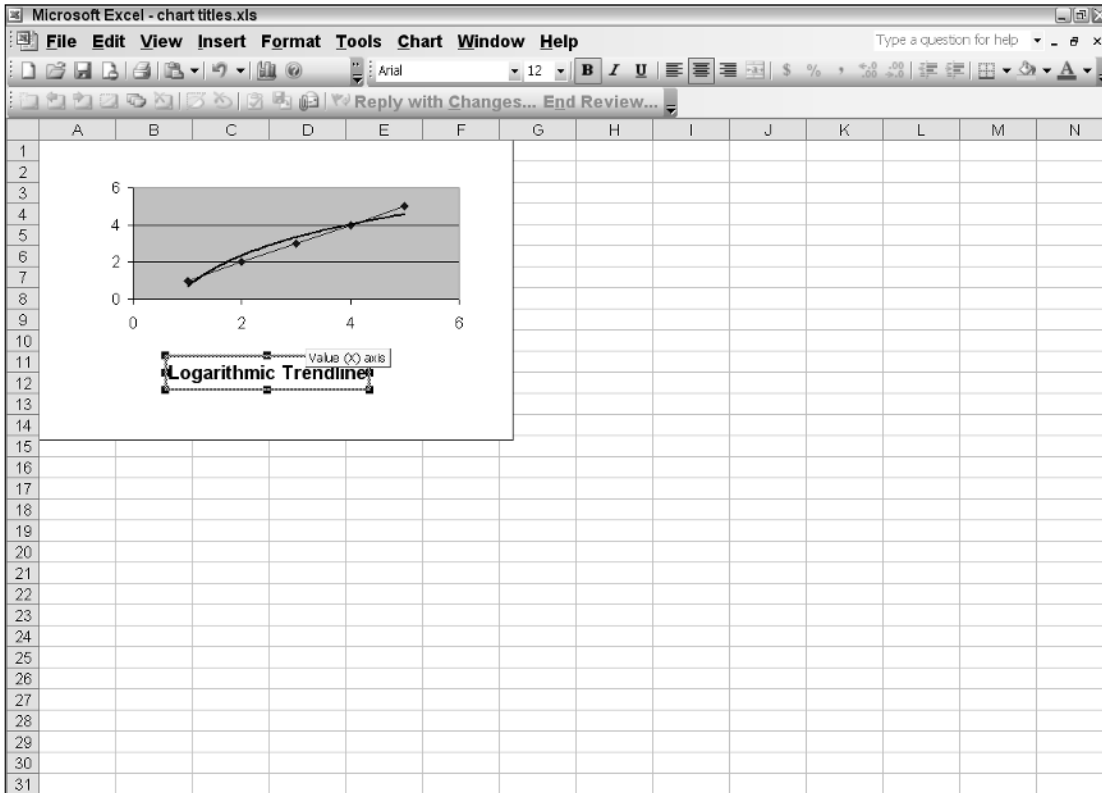


Figure 6-22

Internally Supported Trendlines

Let's have a look at the VSTO-supported `Trendlines`. The functionality is exposed through a `Trendlines` object collection. Each item in this collection contains a `Trendline` object. The `Trendline` object contains properties and methods that allow the developer to push and prod the `Trendline` to her or his own liking. Listing 6-15 shows some code to do this.

Visual Basic

```

Dim xlChart As Excel.Chart = DirectCast(Globals.ThisWorkbook.Charts.Add(),
Excel.Chart)
    Dim cellRange As Excel.Range = Me.Range("a1", "e1")
    xlChart.SetSourceData(cellRange)

    'get a reference to the series collection to plot a trend line
    Dim series As Excel.Series = xlChart.SeriesCollection(1)

    Dim line As Excel.Trendline = (DirectCast(series.Trendlines(),
Excel.Trendlines)).Add(Excel.XlTrendlineType.xlMovingAvg, , 2, 0.4, 0.4, , True, ,
"Moving ave.")

    line.Select()
    line.Border.LineStyle = Excel.XlLineStyle.xlDot
    line.Border.Color = ColorTranslator.ToOle(Color.Red)
    line.InterceptIsAuto = True

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace MultiChart
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            //first chart
            Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
            Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;
            xlChart.SetSourceData(cellRange, missing);

            //get a reference to the series collection to plot a trend line
            Excel.Series series = xlChart .SeriesCollection(1) as Excel.Series;

            Excel.Trendline line =
((Excel.Trendlines)series.Trendlines(missing)).Add(Excel.XlTrendlineType.xlMovingAv
g, missing, 2, 0.4, 0.4, missing, true, missing, "Moving ave.");

            line.Select();
            line.Border.LineStyle = Excel.XlLineStyle.xlDot;
            line.Border.Color = ColorTranslator.ToOle(Color.Red);
            line.InterceptIsAuto = true;
        }

        private void Sheet1_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}

```

The code presented in Listing 6-15 first loads data into the chart through range A1:E1. A chart `Trendline` cannot be rendered without data. In fact, an exception is thrown if you attempt to manipulate a `Trendline` of chart that does not contain data. Since the `Trendline` is part of the series, we use a reference to the series to gain access to the `Trendline`. For our purposes, we simply add a moving average `Trendline`. A moving average `Trendline` uses the average of a specific number of data points to create the `Trendline`. The net effect of using an average in this smoothes out the fluctuations that can occur in data.

You should notice that the line of code that creates a `Trendline` is very unfriendly and intimidating. However, the ugliness is a necessity since you cannot access a `Trendline` through any other means. The VSTO chart can support six types of `Trendlines`: linear, logarithmic, polynomial, power, exponential, and moving average. In each case, the approach is identical to Listing 6-14. Simply change the `Excel.XlTrendlineType`.

Custom Trendlines

For most charts and requirements, the internally implemented `Trendline` will suffice. However, there are some special cases or some lofty end-user requirements that necessitate taking control over the trending process. Let's examine a very rudimentary example so that you can get the basic idea. After that, you are free to let your imagination run wild.

Consider the piece of code in Listing 6-16.

Visual Basic

```
Private Sub AddCustomTrendline()  
    Dim xlChart As Excel.Chart = DirectCast (Globals.ThisWorkbook.Charts.Add(),  
Excel.Chart)  
    Dim cellRange As Excel.Range = Me.Range("a1", "e1")  
  
    xlChart.SetSourceData(cellRange.CurrentRegion)  
    Globals.ThisWorkbook.ActiveChart.ChartType = Excel.XlChartType.xl3DColumn  
  
    Dim series As Excel.Series =  
Globals.ThisWorkbook.ActiveChart.SeriesCollection(1)  
    Dim axisScale As Excel.Axis = DirectCast (  
xlChart.Axes(Excel.XlAxisType.xlCategory, Excel.XlAxisGroup.xlPrimary), Excel.Axis)  
  
    Dim line As Excel.Shape = xlChart.Shapes.AddLine(500, 250, 100, 100)  
    line.Name = "Custom trend line"  
End Sub
```

C#

```
private void AddCustomTrendline()  
{  
    Excel.Chart xlChart =  
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);  
    Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;  
  
    xlChart.SetSourceData(cellRange.CurrentRegion, missing);  
}
```

```

Globals.ThisWorkbook.ActiveChart.ChartType =
Excel.XlChartType.xl3DColumn;

Excel.Series series =
Globals.ThisWorkbook.ActiveChart.SeriesCollection(1) as Excel.Series;
Excel.Axis axisScale =
(Excel.Axis)Globals.ThisWorkbook.ActiveChart.Axes(Excel.XlAxisType.xlCategory,
Excel.XlAxisGroup.xlPrimary);

Excel.Shape line = Globals.ThisWorkbook.ActiveChart.Shapes.AddLine(500,
250, 100, 100);
line.Name = "Custom trend line";
}

```

Listing 6-16, Custom trendline generation

The code in Listing 6-16 produces the chart in Figure 6-23.

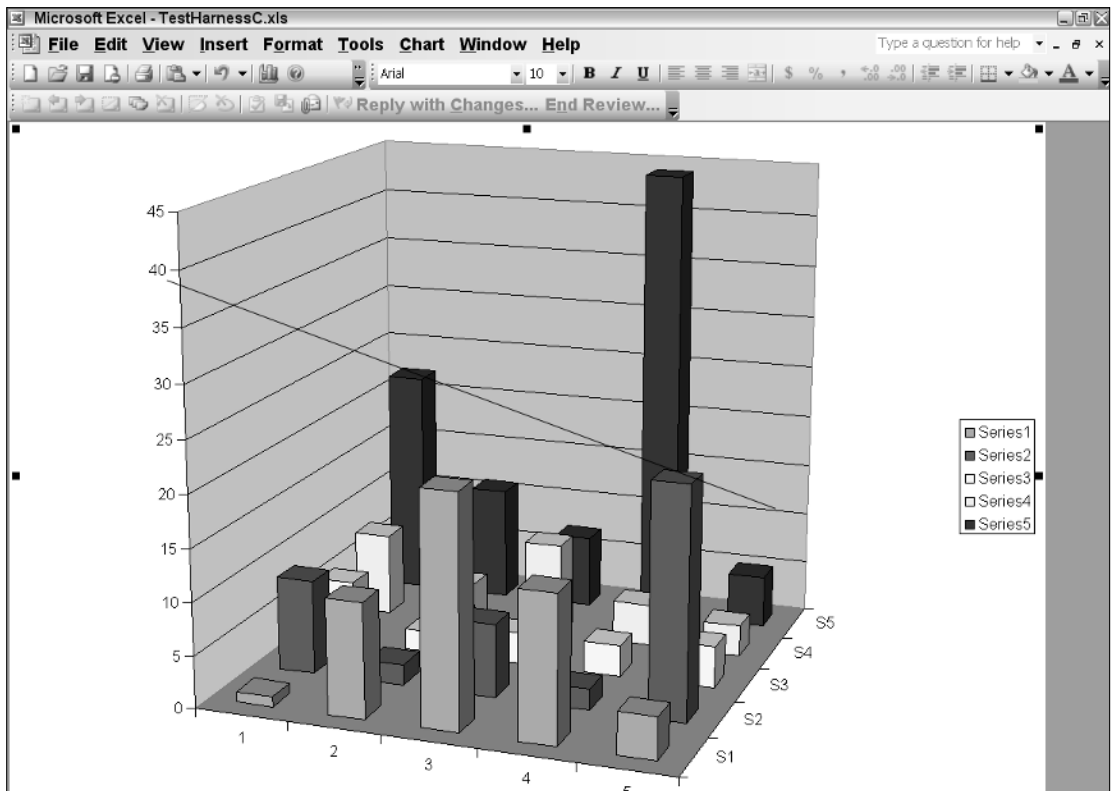


Figure 6-23

As you can see from Figure 6-23, a custom line is imposed on the chart surface. Before we go further, you should note that this is a simple example that creates a basic line on a chart. It does not necessarily relate

to the data points. However, you can render fancy Bezier curves, rectangles, and a multitude of VSTO-supported shapes that are based on the data in the series. In every case, the code will follow the approach outlined in Figure 6-23.

Another point of interest is that the line on the chart behaves much like an object added to a Microsoft Word document; that is, you can interact with the `Trendline` by dragging it and applying some customization to it, assuming that you have not set the `locked` property of the `Trendline` object to `true`. Finally, the line does not have to be hard-coded in. You can use the `maximum` and `minimum` scale property of the chart to determine where to draw the line relative to the series values. Or, the values may be calculated dynamically based on the values of each point. The data labels section shows code that can manipulate each point.

Error Bar Analysis

As the term implies, error bars denote a potential error with the data in the series. Usually, error bars and their associated notations are reserved for scientific charting or applications where the interpretation of data on a series must be mathematically correct. In this case, the error bar represents some fluctuation around the actual data point.

Aside from the fluffy definition, `ErrorBars` are not necessarily complicated. First, they apply to any chart that contains one or more series. However, `ErrorBars` are typically reserved for charts that must calculate some mathematical quantity. It's uncommon to find `ErrorBars` in a pie chart, for instance. Listing 6-17 has some code for error bars.

Visual Basic

```
Private Sub AddErrorBars()  
    Dim xlChart As Excel.Chart = Globals.ThisWorkbook.Charts.Add()  
    Dim cellRange As Excel.Range = Me.Range("a1", "e1")  
    xlChart.SetSourceData(cellRange)  
    xlChart.ChartType = Excel.XlChartType.xlLine  
    Dim series As Excel.Series = xlChart.SeriesCollection(1)  
    Dim errorBars As Excel.ErrorBars =  
    (DirectCast(series.ErrorBar(Excel.XlErrorBarDirection.xlY,  
Excel.XlErrorBarInclude.xlErrorBarIncludeBoth,  
Excel.XlErrorBarType.xlErrorBarTypeFixedValue), Excel.ErrorBars))  
  
End Sub
```

C#

```
private void AddErrorBars()  
{  
    Excel.Chart xlChart =  
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);  
    Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;  
    xlChart.SetSourceData(cellRange, missing);  
    xlChart.ChartType = Excel.XlChartType.xlLine;  
    Excel.Series series = xlChart.SeriesCollection(1) as Excel.Series;  
    Excel.ErrorBars errorBars =  
(Excel.ErrorBars)series.ErrorBar(Excel.XlErrorBarDirection.xlY,  
Excel.XlErrorBarInclude.xlErrorBarIncludeBoth,  
Excel.XlErrorBarType.xlErrorBarTypeFixedValue, missing, missing);  
}
```

Listing 6-17 Error bar customization

Figure 6-24 shows the code in action.

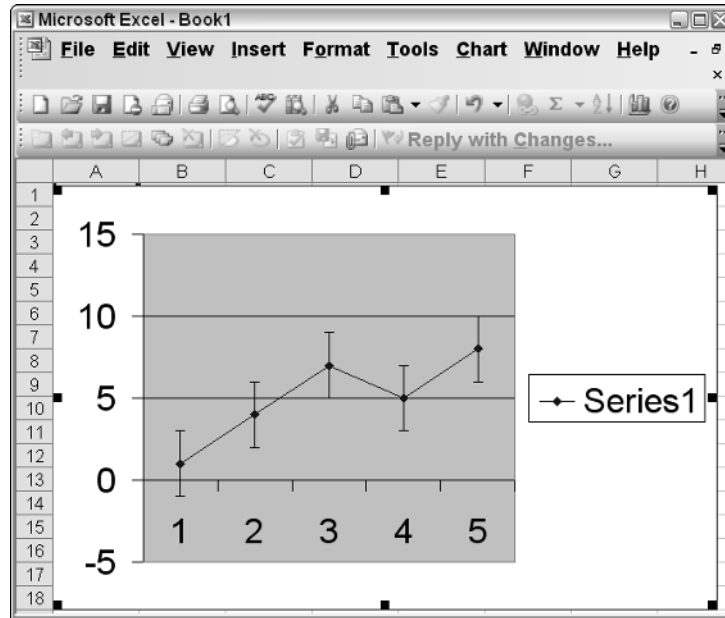


Figure 6-24

According to the code presented in Listing 6-16, the VSTO Excel object presents no direct access to the `ErrorBars` collection. This is exceedingly unfriendly to code because we must rely on an ugly cast. Once we perform the necessary surgery, we can then add an error bar to the chart by calling the `ErrorBars` method. The first parameter tells the chart the particular axis that will receive the `ErrorBar`. In our case, it is the Y axis.

The second parameter specifies which part of the error bar to include. An `ErrorBar` is made up of two parts; the top part and bottom part. The code instructs the chart to render both parts. The third parameter is the `ErrorBar` type. The fourth and fifth parameters are optional, having to do with the `xlErrorBarTypeCustom` setting. Normally, these features should not form part of the immediate data on the series unless specifically required. However, it is prudent to provide such features to the end user through some sort of click event.

Chart Customization through Events

If you have been following the material presented in this book so far, you should realize that events have a consistent model in Visual Studio Tools for the Office System. Additionally, the approach to programming events remains the same even though the object target changes. For instance, the event hookup for Excel is largely the same for Outlook. There is no reason to expect this model to change for the charts as well.

Chapter 6

The basic premise still remains; find a suitable event and wire it up. When the event fires, the VSTO engine notifies all subscribers that the event has fired. The handling code is then called into action. If you have indicated your intent to handle a particular event, your code is executed.

Let's proceed with a simple example. Our example adds a different line marker style to a chart once the chart is clicked. Listing 6-18 has the code.

Visual Basic

```
Public Class Sheet1
    Private Sub MouseUp(ByVal Button As Integer, ByVal Shift As Integer, ByVal x As Integer, ByVal y As Integer)
        'get a reference to the series collection to customize a chart point
        Dim series As Excel.Series =
        DirectCast(Globals.ThisWorkbook.ActiveChart.SeriesCollection(1), Excel.Series)
        If series.MarkerStyle <>
        Microsoft.Office.Interop.Excel.XlMarkerStyle.xlMarkerStylePlus Then
            series.MarkerStyle =
        Microsoft.Office.Interop.Excel.XlMarkerStyle.xlMarkerStylePlus
            series.MarkerSize = 4
            series.MarkerBackgroundColor = ColorTranslator.ToOle(Color.Red)
            series.MarkerForegroundColor = ColorTranslator.ToOle(Color.White)
            Dim oPoint As Excel.Point = DirectCast(series.Points(2), Excel.Point)
            oPoint.HasDataLabel = True
            Globals.ThisWorkbook.ActiveChart.Refresh()
        End If

    End Sub

    Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
        Handles Me.Startup
        'first chart
        Dim xlChart As Excel.Chart = DirectCast (Globals.ThisWorkbook.Charts.Add(),
        Excel.Chart)
        Dim cellRange As Excel.Range = Me.Range("a1", "e1")
        xlChart.SetSourceData(cellRange)
        xlChart.ChartType = Excel.XlChartType.xlRadar

        AddHandler xlChart.MouseUp, AddressOf MouseUp

    End Sub

    Private Sub Sheet1_Shutdown(ByVal sender As Object, ByVal e As System.EventArgs)
        Handles Me.Shutdown

    End Sub

End Class
```

C#

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
```

```

using Office = Microsoft.Office.Core;

namespace ChartEvent
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
            Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;
            xlChart.SetSourceData(cellRange, missing);
            xlChart.ChartType = Excel.XlChartType.xlRadar;

            //wire event handler
            xlChart.MouseUp += new
Microsoft.Office.Interop.Excel.ChartEvents_MouseUpEventHandler(xlChart_MouseUp);
        }

        void xlChart_MouseUp(int Button, int Shift, int x, int y)
        {
            //get a reference to the series collection to customize a chart point
            Excel.Series series =
Globals.ThisWorkbook.ActiveChart.SeriesCollection(1) as Excel.Series;
            if (series.MarkerStyle !=
Microsoft.Office.Interop.Excel.XlMarkerStyle.xlMarkerStylePlus)
            {
                series.MarkerStyle =
Microsoft.Office.Interop.Excel.XlMarkerStyle.xlMarkerStylePlus;
                series.MarkerSize = 4;
                series.MarkerBackgroundColor = ColorTranslator.ToOle(Color.Red);
                series.MarkerForegroundColor = ColorTranslator.ToOle(Color.White);
                Excel.Point oPoint = series.Points(2) as Excel.Point;
                Globals.ThisWorkbook.ActiveChart.Refresh();
            }
        }

        private void Sheet1_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}

```

Listing 6-18 Customization using the event model

From the code presented in Listing 6-18, a reference to the series must be obtained. Next, the marker styles and settings are applied to the series. The result of this customization is shown in Figure 6-25. You should note that the markers are emphasized with color so that they appear visible. Normally, the markers are small enough so that they may not be readily visible. Also, you should note that after the customization has been applied to the marker, the chart must be refreshed so that the customizations can appear. The markers and customizations will not be available otherwise.

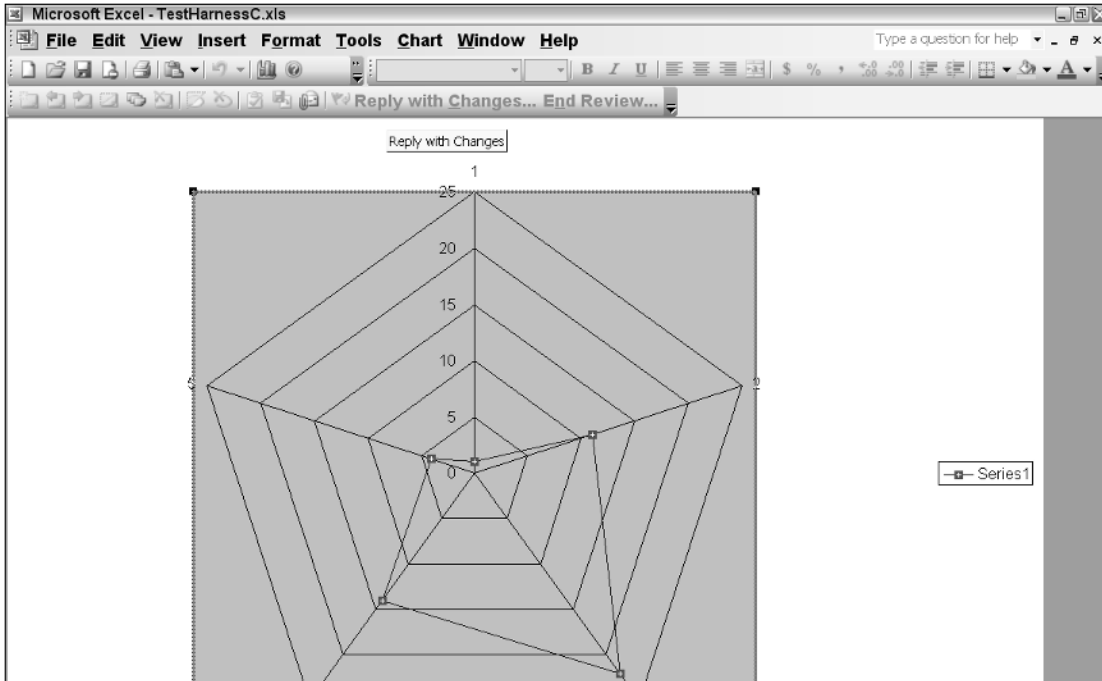


Figure 6-25

A well-designed chart application can improve the end-user experience through interaction, while preventing information overload. Information is passed along to the end user in digestible quantities. For instance, consider a chart that appears with only a series on the chart surface. However, captions and titles are added when the end user clicks a particular area on the chart. Information presented in this way has two distinct advantages. The chart surface remains uncluttered and easy to read. The information presented in an uncluttered matter is not intimidating to the end user.

Advanced Chart Manipulation

Now that you have the basics of charting, it's time to take it up a notch or two. Before you plunge in, observe this word of caution. Be very careful to avoid dazzling end users with endless functionality. Remember that a chart transforms data so that it makes an impact in a way that does not overwhelm the end user. A visually interesting chart that has a lot going on tends to intimidate the average user. Intimidation is counterproductive. It's better to keep a chart clean and simple, using events here and there to bring in extra information.

Charts with Multiple Series

Usually, charts are rendered with a single series for simple data sets. However, for more involved data analysis, more than one series may be required to represent the data. In this circumstance, two or more series are rendered on the chart surface. Consider the bar chart in Figure 6-26, showing two series.

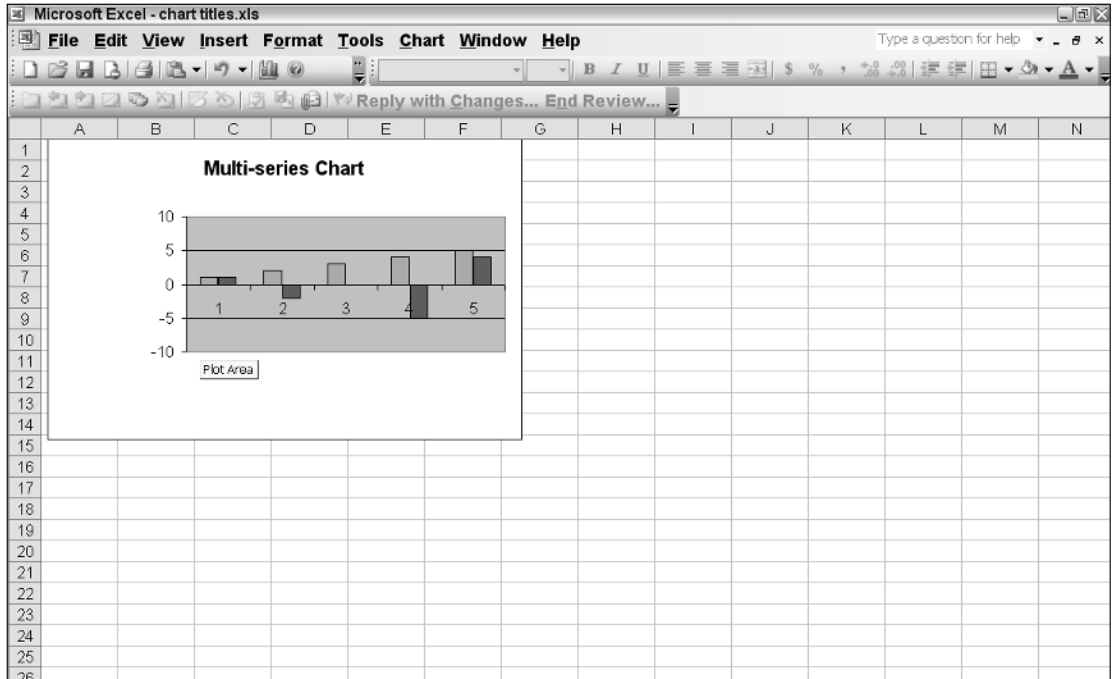


Figure 6-26

All of the functionality available to one series is available to the other series. Therefore, the multiple series approach is no different from the single series approach. Listing 6-19 shows the code.

Visual Basic

```
Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
    Handles Me.Startup
    Dim xlChart As Excel.Chart = DirectCast (Globals.ThisWorkbook.Charts.Add(),
    Excel.Chart)
    Dim cellRange As Excel.Range = Me.Range("a1", "b5")

    xlChart.SetSourceData(cellRange.CurrentRegion)
    Globals.ThisWorkbook.ActiveChart.ChartType =
    Excel.XlChartType.xl3DBarClustered
End Sub
```

C#

```
private void Sheet1_Startup(object sender, System.EventArgs e)
{
    Excel.Chart xlChart =
    (Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
    Excel.Range cellRange = this.Range["a1", "b5"] as Excel.Range;

    xlChart.SetSourceData(cellRange.CurrentRegion, missing);
    xlChart .ChartType = Excel.XlChartType.xl3DBarClustered;
}
```

Listing 6-19 Multi-series charting code

Listing 6-19 shows the fourth chart title associated with a chart. Notice that, as discussed earlier, a chart can have as many as five titles imposed on its surface.

A single series is given by a single row of data in the spreadsheet. Consider an example that sources data from the range A1:A5. This produces a single series on the chart. A contiguous range A1: E5 actually renders five different series, since there are five rows in this range. Listing 6-19 shows an easy way to create a multiple series chart. The multiple series is sourced from two Excel columns. Examine Figure 6-27 to see the data and the results of the code when it is executed.

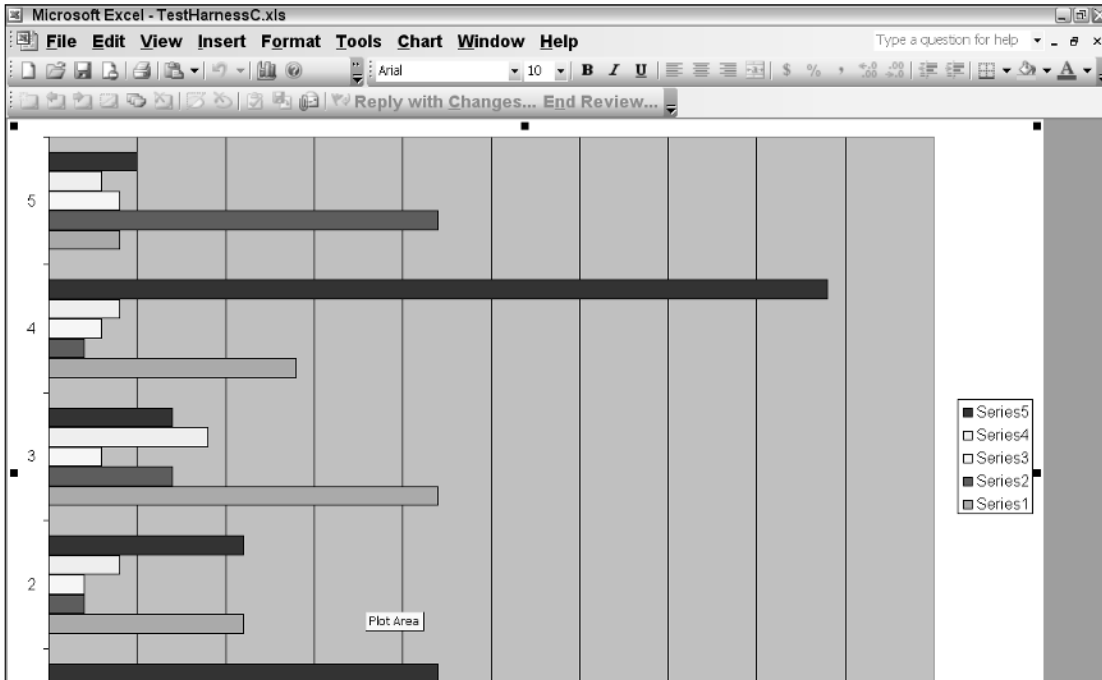


Figure 6-27

You can exercise the same precise control even though there may be more than one series on the chart surface. The trick is to always obtain a reference to the correct series. With a suitable reference, the code to modify the series remains the same as presented in the series customization section. Remember, a multi-series chart customization is handled the same way; one series at a time.

Combination Charts

For comparison purposes, it can be handy to render a combination of two charts instead of a single chart with multiple series. Each chart in a combination chart surface is independent of the other and may be targeted with code similar to the approach for building single charts. Listing 6-20 shows the code.

Visual Basic

```

Dim ChartObjects As Excel.ChartObjects = DirectCast (Me.ChartObjects(),
Excel.ChartObjects)
    Dim chartObject As Excel.ChartObject = ChartObjects.Add(1, 20, 250, 250)

    chartObject.Chart.ChartWizard(Me.Range("A1", "E1"),
Excel.XlChartType.xl3DColumn, Title:="First Chart")
    chartObject.Name = "InitialiChartObject"

ChartObjects = DirectCast (Me.ChartObjects(), Excel.ChartObjects)
Dim chartObject2 As Excel.ChartObject = ChartObjects.Add(251, 20, 250, 250)

chartObject2.Chart.ChartWizard(Me.Range("F1", "K1"),
Excel.XlChartType.xl3DColumn, Title:= "Second Chart")
chartObject2.Name = "SecondaryChartObject"
chartObject2.Activate()

```

C#

```

private void Sheet1_Startup(object sender, System.EventArgs e)
{
Excel.ChartObjects ChartObjects =
    (Excel.ChartObjects)this.ChartObjects(missing);
Excel.ChartObject chartObject = ChartObjects.Add(1, 20, 250, 250);

chartObject.Chart.ChartWizard(this.Range["A1", "E1"],
Excel.XlChartType.xl3DColumn, missing, missing, missing,
missing, missing, "First Chart", missing, missing, missing);
chartObject.Name = "InitialiChartObject";

ChartObjects = (Excel.ChartObjects)this.ChartObjects(missing);
Excel.ChartObject chartObject2 = ChartObjects.Add(251, 20, 250, 250);

chartObject2.Chart.ChartWizard(this.Range["F1", "K1"],
Excel.XlChartType.xl3DColumn, missing, missing, missing,
missing, missing, "Second Chart", missing, missing, missing);
chartObject2.Name = "SecondaryChartObject";
chartObject2.Activate();
}

```

Listing 6-20 Rendering multiple charts on a single spreadsheet

As the name implies, combination charts combine two charts on a single Excel spreadsheet. Notice that this is distinctly different from multiple charts imposed on a single chart surface. VSTO does not currently support multiple charts on a single chart surface. It's important to note here that this is a VSTO imposed limitation since combination charts imposed on a single chart surface are supported in the Office Web Components and through the regular Excel Interop libraries. The idea behind combination charts is to expose some sort of contrast in the data.

Notice the adjustment with the numerical parameters so that the two charts line up correctly on the spreadsheet. Using these position parameters, it is possible to site a chart in specific regions of the spreadsheet. Although the numerical parameters are hard-coded based on the size of the chart space, it is just as easy to perform at runtime. Just obtain a reference to the `chartspace` object and use the `width` and `height` properties.

Chapter 6

As with multiple series, you can precisely control the two charts on the chart surface. Listing 6-20 shows some code to manipulate the charts independently (see Figure 6-28).

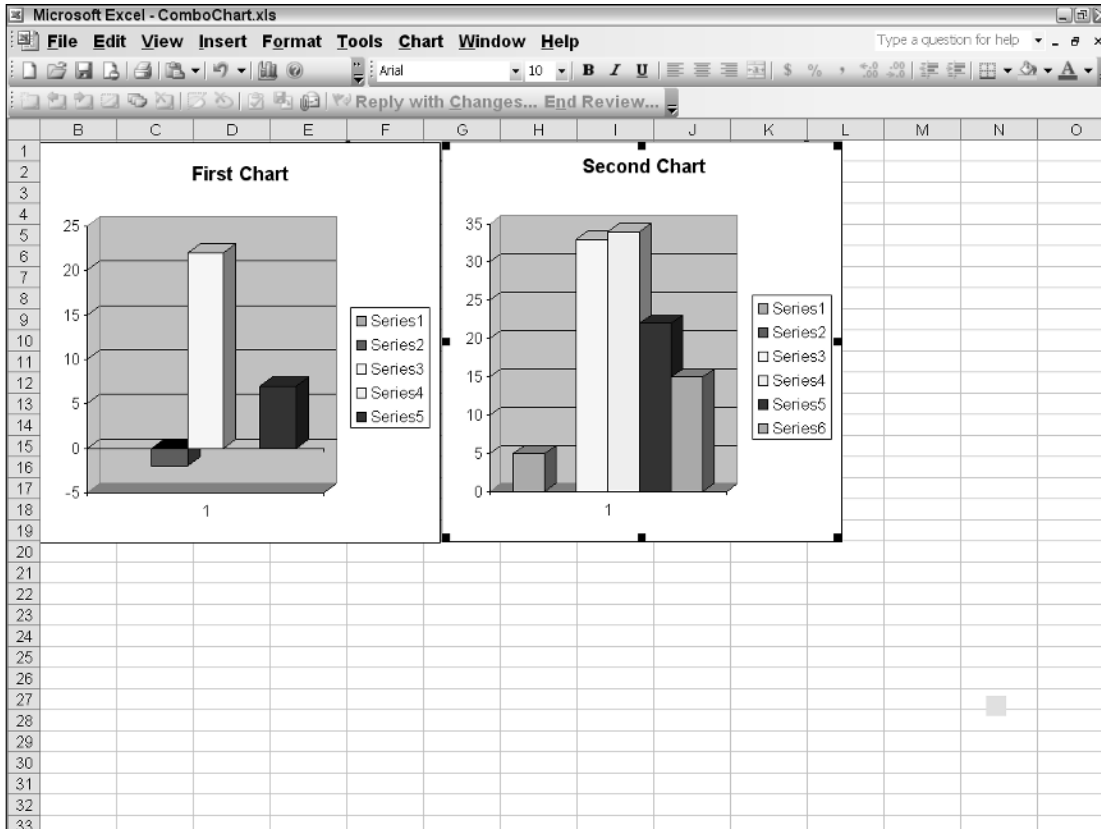


Figure 6-28

The VSTO chart offers no way to split axes to specific intervals. Again, this is only a VSTO-imposed limitation. This functionality is alive and well through other automation techniques.

Perspectives and 3-D Drawing

VSTO is quite capable of taking it up a notch or two. And you may need that type of flexibility if your requirements dictate that the chart must have bang for the buck. One good way to dazzle is to use perspectives and 3-D functionality. Another simpler approach is to customize the surfaces of the chart. Listing 6-21 has some code to illustrate some basic concepts for customizations.

Visual Basic

```

Private Sub Sheet1_Startup(ByVal sender As Object, ByVal e As System.EventArgs)
Handles Me.Startup
    Dim xlChart As Excel.Chart = DirectCast (Globals.ThisWorkbook.Charts.Add(),
Excel.Chart)
    Dim cellRange As Excel.Range = Me.Range("a1", "b5")

    xlChart.SetSourceData(cellRange.CurrentRegion)
    xlChart .ChartType = Excel.XlChartType.xl3DBarClustered

    xlChart .RightAngleAxes = false
    xlChart .Perspective = 15
    xlChart .Elevation = 44
End Sub

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace Perspectives
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
            Excel.Range cellRange = this.Range["a1", "b5"] as Excel.Range;

            xlChart.SetSourceData(cellRange.CurrentRegion, missing);
            xlChart .ChartType = Excel.XlChartType.xl3DBarClustered;

            xlChart .RightAngleAxes = false;
            xlChart .Perspective = 15;
            xlChart .Elevation = 44;
        }

        private void Sheet1_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}

```

Listing 6-21 Chart perspectives

The code produces the chart in Figure 6-29.

Notice how the perspective of the chart is tilted by 15 degrees. While this is a theoretical exercise, such functionality can have real impact in an enterprise software world, especially when chart objects are mixed with pieces of text on-screen. Also, you should notice that the chart type can simply be changed to a 3-D-supported type, and the chart engine will render a 3-D image.

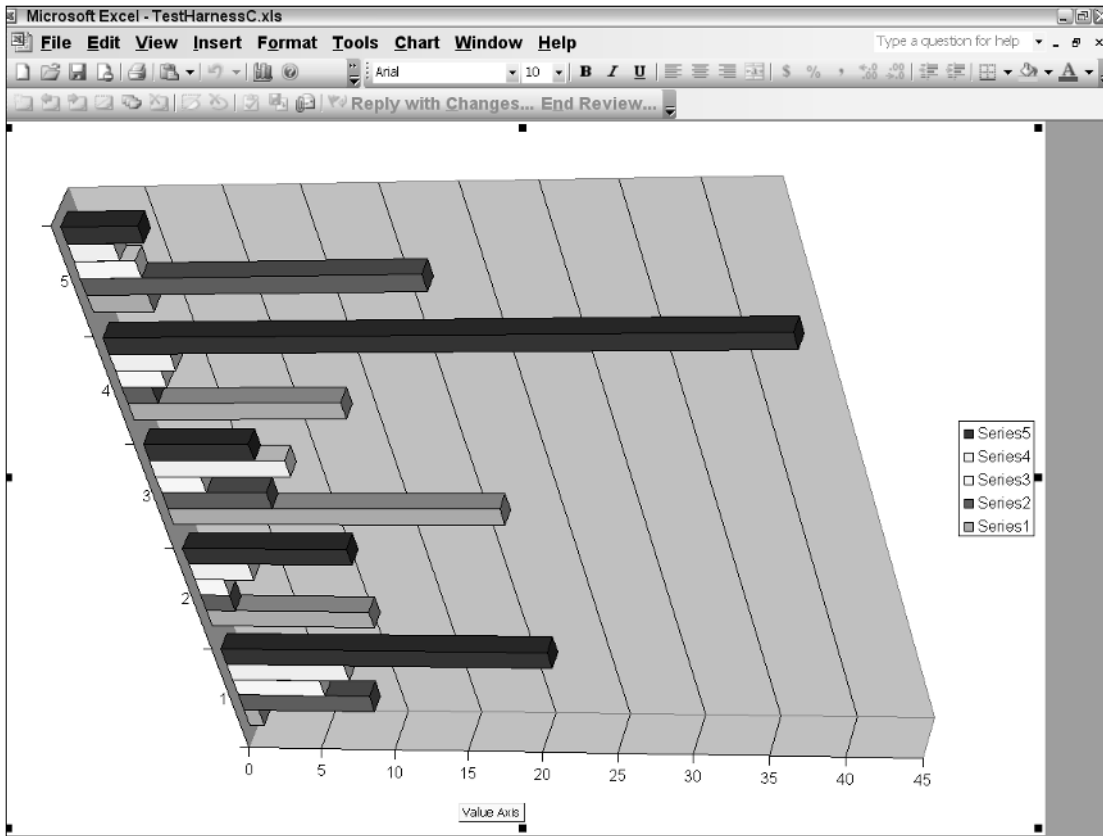


Figure 6-29

The final line of code sets the elevation of the chart. The elevation is measured in degrees, since it acts as a sort of rotation to the chart view plane. If you examine Figure 6-29, the elevation angle is responsible for the awkward tilt. Valid values range from 0 through 44 degrees. Where would this functionality be helpful? Consider an application that displays a chart that is able to respond to mouse movements in real time. As the mouse moves, code grabs the mouse coordinates and updates the perspective. High-end analytical packages are built on this simple premise.

Chart Point Customization

A series is a collection of points plotted on a chart surface. The code you've seen so far has manipulated the series as a whole; that is, the collection of points. Let's turn our attention to the individual points that form part of the series. It just so happens that the `SeriesCollection` exposes a `Points` object collection that provides access to the individual points that make up the series. If you think about it for a moment, you should begin to form a picture of endless possibilities in your mind.

Consider a requirement that performs a certain customization on a chart based on the value of each point. One approach to fulfilling this requirement is to gain access to the `Points` object collection. Consider the code in Listing 6-22.

Visual Basic

```

Dim xlChart As Excel.Chart = DirectCast
(Globals.ThisWorkbook.Charts.Add(), Excel.Chart)
Dim cellRange As Excel.Range = Me.Range("a1", "e1")
xlChart.SetSourceData(cellRange)
xlChart.ChartType = Excel.XlChartType.xlPie
'get a reference to the series collection to customize a chart point
Dim series As Excel.Series =
DirectCast(Globals.ThisWorkbook.ActiveChart.SeriesCollection(1), Excel.Series)
Dim oPoint As Excel.Point = series.Points(1)
oPoint.HasDataLabel = True
oPoint.Interior.Color = ColorTranslator.ToOle(Color.Red)

```

C#

```

using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace Points
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, System.EventArgs e)
        {
            Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
            Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;
            xlChart.SetSourceData(cellRange, missing);
            xlChart.ChartType = Excel.XlChartType.xlPie;
            //get a reference to the series collection to customize a chart point
            Excel.Series series = xlChart .SeriesCollection(1) as Excel.Series;
            Excel.Point oPoint = series.Points(1) as Excel.Point;
            oPoint.HasDataLabel = true;
            oPoint.Interior.Color = ColorTranslator.ToOle(Color.Red);
        }

        private void Sheet1_Shutdown(object sender, System.EventArgs e)
        {
        }
    }
}

```

Listing 6-22 Customization of points in a data series

The larger idea here is that the series collection provides access to the points object collection. Each point corresponds to the value in the first row of the spreadsheet, as shown in Figure 6-30. In Figure 6-30, one slice of the pie is colored red, since it represents the first point in the collection.

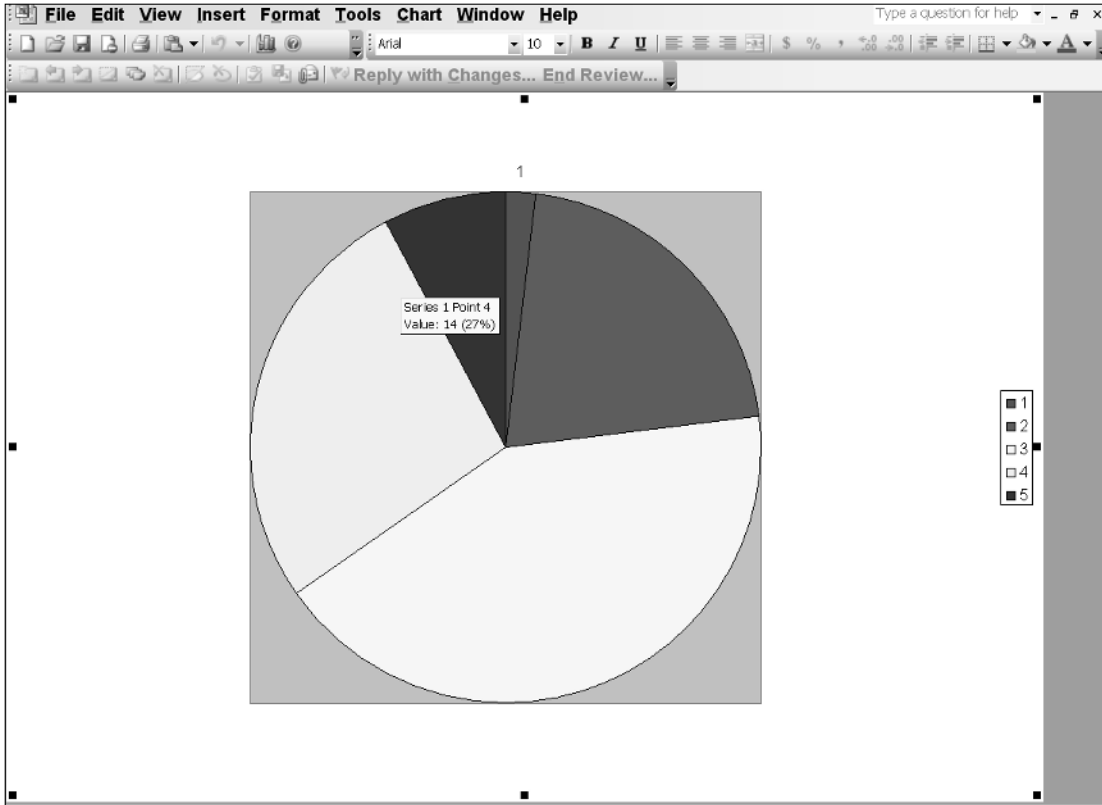


Figure 6-30

For simplicity, the code simply picks the first point. However, you can let your imagination run wild here because you have access to all the points that can be plotted on the chart surface. For instance, you may consider adding a `MarkerStyle` to the point if it meets a specific criterion. However, you should be careful because `MarkerStyles` apply to the series as a whole and not to this single point. Another point worth noting is that certain chart types, line charts for instance, don't support point color formatting.

Now that the basics are safely out of the way, let's consider an advanced scenario. How do we test a point to determine its value before applying customization? As pointed out previously, we may need to apply some specific format if a value of a point is above 5 for instance. Listing 6-23 has some code that demonstrates an approach.

Visual Basic

```
Dim xlChart As Excel.Chart = DirectCast (Globals.ThisWorkbook.Charts.Add(),
Excel.Chart)
    Dim cellRange As Excel.Range = Me.Range("a1", "b5")

    xlChart.SetSourceData(cellRange.CurrentRegion)
    Globals.ThisWorkbook.ActiveChart.ChartType =
Excel.XlChartType.xl3DBarClustered
    'get a reference to the series collection to customize a chart point
    Dim series As Excel.Series =
DirectCast(Globals.ThisWorkbook.ActiveChart.SeriesCollection(1),Excel.Series)
```

```

Dim stopFlag As Boolean = False
Dim index As Integer = 0
Do
    Try
        index = index + 1
        Dim oPoint As Excel.Point =
DirectCast(series.Points(index),Excel.Point)
        oPoint.HasDataLabel = True

        If oPoint.DataLabel IsNot Nothing Then
            Dim val As Double = 0
            If Double.TryParse(oPoint.DataLabel.Text, NumberStyles.Integer
Or NumberStyles.AllowTrailingWhite Or NumberStyles.AllowLeadingWhite Or
NumberStyles.AllowLeadingSign Or NumberStyles.AllowDecimalPoint,
NumberFormatInfo.InvariantInfo, val) Then
                If val > 5 Then
                    'do something with point
                End If
            End If
        End If

        Catch ex As Exception
            stopFlag = True
        End Try
    Loop While Not stopFlag

```

C#

```

//get a reference to the series collection to customize a chart point
Excel.Series series =
Globals.ThisWorkbook.ActiveChart.SeriesCollection(1) as Excel.Series;

bool stop = false;
int index = 0;
do
{
    try
    {
        index++;
        Excel.Point oPoint = series.Points(index) as Excel.Point;
        oPoint.HasDataLabel = true;

        if (oPoint.DataLabel != null)
        {
            double val = 0;
            if (Double.TryParse(oPoint.DataLabel.Text,
NumberStyles.Integer | NumberStyles.AllowTrailingWhite |
NumberStyles.AllowLeadingWhite | NumberStyles.AllowLeadingSign |
NumberStyles.AllowDecimalPoint, NumberFormatInfo.InvariantInfo, out val))
            {
                if (val > 5)
                {
                    //do something with point
                }
            }
        }
    }
}

```

```
        }
        catch (Exception ex)
        {
            stop = true;
        }
    } while (!stop);

    Globals.ThisWorkbook.ActiveChart.Refresh();
```

Listing 6-23 Code to test point values

To compile this code you will need to include the `System.Globalization` namespace.

First, an appropriate reference is obtained for series. With that reference, the code checks to see if the appropriate `MarkerStyle` has been applied. The size of the marker is adjusted for visibility purposes. This code is essentially a repeat from the code in Listing 6-21. From that point, things get progressively ugly. Here is why.

The idea behind the code is to test each point to see if it matches some criterion. In that case, if the value is greater than 5, then some action is taken. However, the `Points` object collection does not contain a `count` property. What's worse is that the `Points` object is implemented as a method. If you care to examine the documentation at this point, it indicates that the `Points(Points.Count)` is an index into the last point in the series. However, since `Points` is implemented as a method, `Points.Count` access is simply an invalid syntax.

The end user is usually not interested in this sort of squabbling, so in the interest of getting this to work, we improvise with a counter and a `do-while` loop. Since the collection is not zero based, the index is incremented to 1 on the first pass. The value of the index is passed as an index into the `Points` object collection. The `while` loop will continue to work as long as the indices are valid. An exception is thrown to notify us that the last index was out of bounds. At that point, we set a Boolean flag to stop the `while` loop. You should note that we have broken with best practices convention by using exceptions to determine logic flow. This is awful!

Inside the loop, we need to find out if the point contains a `DataLabel`. The `DataLabel` is equal to the data point being plotted. However, the data point is of type string that we retrieve from the `DataLabel.Text` property. Using the .NET `TryParse` method, we obtain the value of the text and convert it to double. We are not interested in using a simple parse on the string, since an invalid input can throw an exception resulting in the `while` loop stopping prematurely. If you care to examine the `TryParse` method, you should note that the various parameters that are OR'd together simply provide more flexibility in handling the input. The code certainly works but is not elegant at all.

As it turns out, the `Excel.Points` object collection contains a `count` method. That important morsel of information is buried deep inside the help documentation. But, we can use this to our benefit by simply replacing the line that casts the point to a `series.Points` with one that casts to an `Excel.Points` object instead in Listing 6-23. Now that we have a count, we can replace the entire loop code with cleaner, more efficient code. The lesson learned here is that a suitable programming resource can often prevent unnecessary work and result in more efficient code. Listing 6-24 is another example of the `Excel.Points` object in action.

Visual Basic

```

Dim series As Excel.Series =
DirectCast(Globals.ThisWorkbook.ActiveChart.SeriesCollection(1), Excel.Series)
    Dim pnts As Excel.Points = DirectCast(series.Points(), Excel.Points)

    Dim i As Integer
    For i = 1 To pnts.Count - 1 Step i + 1
        Dim val As Excel.Point = pnts.Item(i)
        If i / 2 = 0 Then
            val.Shadow = True
        Else
            val.Shadow = False
        End If
    Next

```

C#

```

Object o;
Excel.Points pnts;
Excel.Series series;
o = ((Excel.Chart)Globals.ThisWorkbook.Charts[1]).SeriesCollection(1);
series = (Excel.Series)o;
pnts = (Excel.Points)series.Points(missing);

for (int i = 1; i < pnts.Count; i++)
{
    Excel.Point val = pnts.Item(i);
    if (i % 2 == 0)
        val.Shadow = true;
    else
        val.Shadow = false;
}

```

Listing 6-24 Series point iteration

The code in Listing 6-24 is essentially unchanged from other code presented earlier, so there is no need to beat a dead horse. On the other hand, the new piece of code requires some analysis. As pointed out earlier, if we cast the object returned from the `series.Points()` method call into an `Excel.Points` object, we gain the use of a count property. And the code approach is neither a hack nor is it inefficient. It's simply poorly documented. Inside the `for` loop, we can perform some action based on the value of the point. In our case, we add a shadow effect to every odd point value.

Adding Objects to the Chart Surface

From time to time, the end user gets bored with the default shapes and may require the ability to jazz things up a bit. How does VSTO charting rise to this challenge? VSTO-based charts can convert default shapes into aesthetic flattery. Consider Listing 6-25, which converts a 3-D clustered bar into a cylinder.

Visual Basic

```

Private Sub CreateCylinderShape()
    Dim xlChart As Excel.Chart = DirectCast
(Globals.ThisWorkbook.Charts.Add(), Excel.Chart)
    Dim cellRange As Excel.Range = Me.Range("a1", "e1")
xlChart.SetSourceData(cellRange)
xlChart.ChartType = Excel.XlChartType.xl3DBarClustered
'get a reference to the series collection to customize a chart point

```



```

Dim series As Excel.Series = xlChart .SeriesCollection(1)
series.BarShape = Microsoft.Office.Interop.Excel.XlBarShape.xlCylinder
End Sub

```

C#

```

private void CreateCylinderShape()
{
    Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;
xlChart.SetSourceData(cellRange, missing);
xlChart.ChartType = Excel.XlChartType.xl3DBarClustered;
//get a reference to the series collection to customize a chart point
Excel.Series series = xlChart .SeriesCollection(1) as Excel.Series;

    series.BarShape = Microsoft.Office.Interop.Excel.XlBarShape.xlCylinder;
}

```

Listing 6-25 Conversion of bars to cylinders

Well, that was surprisingly easy to achieve. As you can see, the series contains built-in support for this conversion. In fact, the enumeration contains six members that allow for interesting charts to be rendered on the chart surface. Figure 6-31 shows the result of the code.

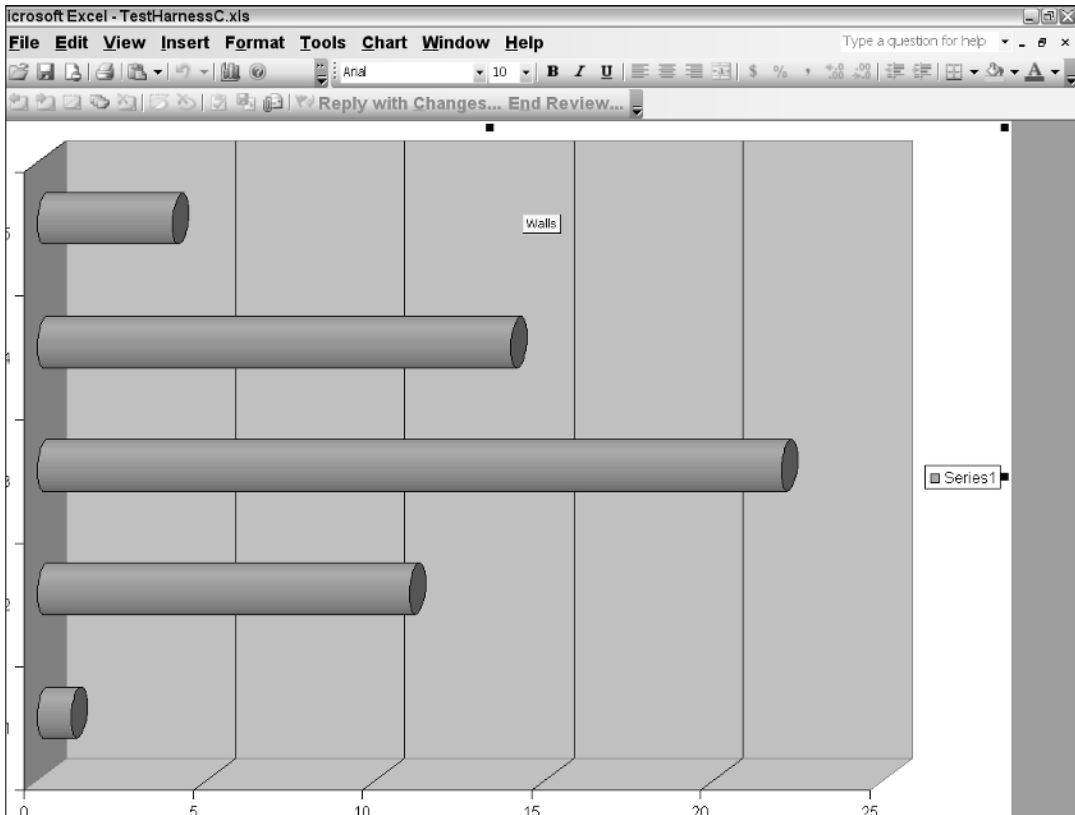


Figure 6-31

From time to time, the end user may also require that certain portions of the data may be highlighted or emphasized. We have already presented code to show how to iterate the `Points` collection. However, this approach is somewhat limited since the `Points` object is marked as read-only. As an alternative, you can perform the cast described in Listing 6-23 to gain access to a `Point` object collection that is not read-only. Another good way to emphasize data is simply to draw or impose a picture on the chart surface next to the area of interest. This is the basic concept but you are free to employ your imagination.

Let's consider an example that renders data to a chart and then emphasizes a particular area of the chart by placing an image next to an area of the chart in response to the end-user click event. The example will proceed with the difficult part, you should be able to wire the click event to the part of the code shown in Listing 6-26.

Visual Basic

```
Private Sub CreateFlag()
    Dim xlChart As Excel.Chart = DirectCast
    (Globals.ThisWorkbook.Charts.Add(), Excel.Chart)
    Dim cellRange As Excel.Range = Me.Range("a1", "e1")
    xlChart.SetSourceData(cellRange)
    xlChart.ChartType = Excel.XlChartType.xl3DBarClustered
    'get a reference to the series collection to customize a chart point
    Dim series As Excel.Series =
    DirectCast(Globals.ThisWorkbook.ActiveChart.SeriesCollection(1), Excel.Series)
    xlChart.Shapes.AddPicture("c:\download\images.jpg",
    Microsoft.Office.Core.MsoTriState.msoCTrue,
    Microsoft.Office.Core.MsoTriState.msoFalse, 50, 50, 100, 100)
End Sub
```

C#

```
private void CreateFlag()
{
    Excel.Chart xlChart =
    (Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
    Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;
    xlChart.SetSourceData(cellRange, missing);
    xlChart.ChartType = Excel.XlChartType.xl3DBarClustered;
    //get a reference to the series collection to customize a chart point
    Excel.Series series = xlChart.SeriesCollection(1) as Excel.Series;
    xlChart.Shapes.AddPicture(@"c:\download\images.jpg",
    Microsoft.Office.Core.MsoTriState.msoCTrue,
    Microsoft.Office.Core.MsoTriState.msoFalse, 50, 50, 100, 100);
}
```

Listing 6-26 Addition of picture objects to chart surface

The code shows that a gif image is imposed on the chart surface. The `CreateFlag` method contains code that simply uses the `Globals` object hook to gain access to the `shapes` property. The `Shapes` property is an object collection that holds all the shapes that are part of the `Chart` collection. We simply use the `AddPicture` method to add our own picture to the collection. The picture is given by the file path `C:\download\flag.bmp`. If the path is not valid, an exception is thrown. Figure 6-32 shows the code in action.

The code shows that an image may be imposed on the chart surface. This is truly magical. Functionality such as this is simply not possible through some of the Excel automation surfaces such as the Office Web Components. However, you should take care to not overdo that type of functionality.

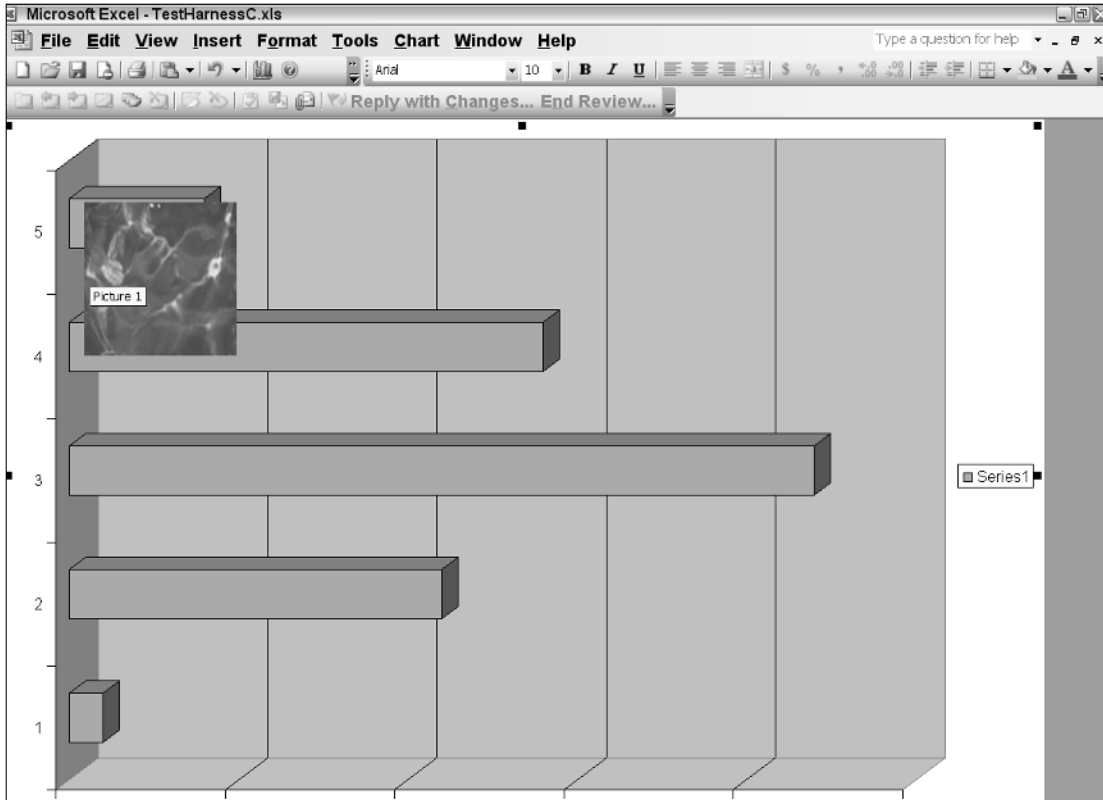


Figure 6-32

It goes without saying that the code only uses the `AddPicture` method, but it is possible to add more shapes to the chart's surface. However, it bears repeating that you should always attempt to practice moderation with your feature list.

Charts with Special Needs

Some charts can render directly to the chart surface and not necessarily to the chart plot area. While it is not important to understand the reason why this is so, these changes do have significant implications for constructing these charts. To avoid complications, you need to be familiar with these charts.

The pie, doughnut, and polar charts are rendered directly to the `chartspace` object and not to the chart plot area. Also, the concept of X and Y axes is not applicable. Instead, you must use the value and category terminology. Finally, these charts contain only one data series. It is not possible to render a multi-series chart of the pie, doughnut, or polar type.

The code to produce these charts remains basically the same as that of standard charts. This is due in large part to a good design on the part of the VSTO architects. However, if you must manipulate these charts through code, the differences between a standard chart and a chart with special needs becomes apparent rather quickly. Usually, these differences are manifested through runtime exceptions in the code.

Consider Listing 6-27, an example of how to render a chart. We perform some common customization as well to complete the example.

Visual Basic

```
Private Sub CreateAndCustomizePie()
Dim xlChart As Excel.Chart = DirectCast (Globals.ThisWorkbook.Charts.Add(),
Excel.Chart)
    Dim cellRange As Excel.Range = Me.Range("a1", "e1")
    xlChart.SetSourceData(cellRange)
    xlChart.ChartType = Excel.XlChartType.xl3DPieExploded

    Dim series As Excel.Series =
DirectCast(Globals.ThisWorkbook.ActiveChart.SeriesCollection(1), Excel.Series)
    series.Shadow = True
    Dim oPoint As Excel.Point = DirectCast(series.Points(4),Excel.Point)
    oPoint.Explosion = 600
    oPoint.HasDataLabel = True
    oPoint.Interior.Color = ColorTranslator.ToOle(Color.Wheat)
    xlChart .SetBackgroundPicture("c:\download\images.jpg")
    Dim pieGroup As Excel.ChartGroup = CType(xlChart.ChartGroups(1),
Excel.ChartGroup)
    pieGroup.GapWidth = 9
    pieGroup.FirstSliceAngle = 45

    xlChart .HasLegend = False
End Sub
```

C#

```
private void CreateAndCustomizePie()
{
    Excel.Chart xlChart =
(Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
    Excel.Range cellRange = this.Range["a1", "e1"] as Excel.Range;
    xlChart.SetSourceData(cellRange, missing);
    xlChart.ChartType = Excel.XlChartType.xl3DPieExploded;

    Excel.Series series = xlChart .SeriesCollection(1) as Excel.Series;
    series.Shadow = true;
    Excel.Point oPoint = series.Points(4) as Excel.Point;
    oPoint.Explosion = 600;
    oPoint.HasDataLabel = true;
    oPoint.Interior.Color = ColorTranslator.ToOle(Color.Wheat);

    xlChart .SetBackgroundPicture(@"c:\download\flag.bmp");
    Excel.ChartGroup pieGroup = (Excel.ChartGroup)xlChart.ChartGroups(1);
    pieGroup.GapWidth = 9;
    pieGroup.FirstSliceAngle = 45;

    xlChart .HasLegend = false;
}
```

Listing 6-27 A pie chart with an explosion

The code in Listing 6-27 produces the image in Figure 6-33. The image in Figure 6-32 has been used as the background in 6-33.

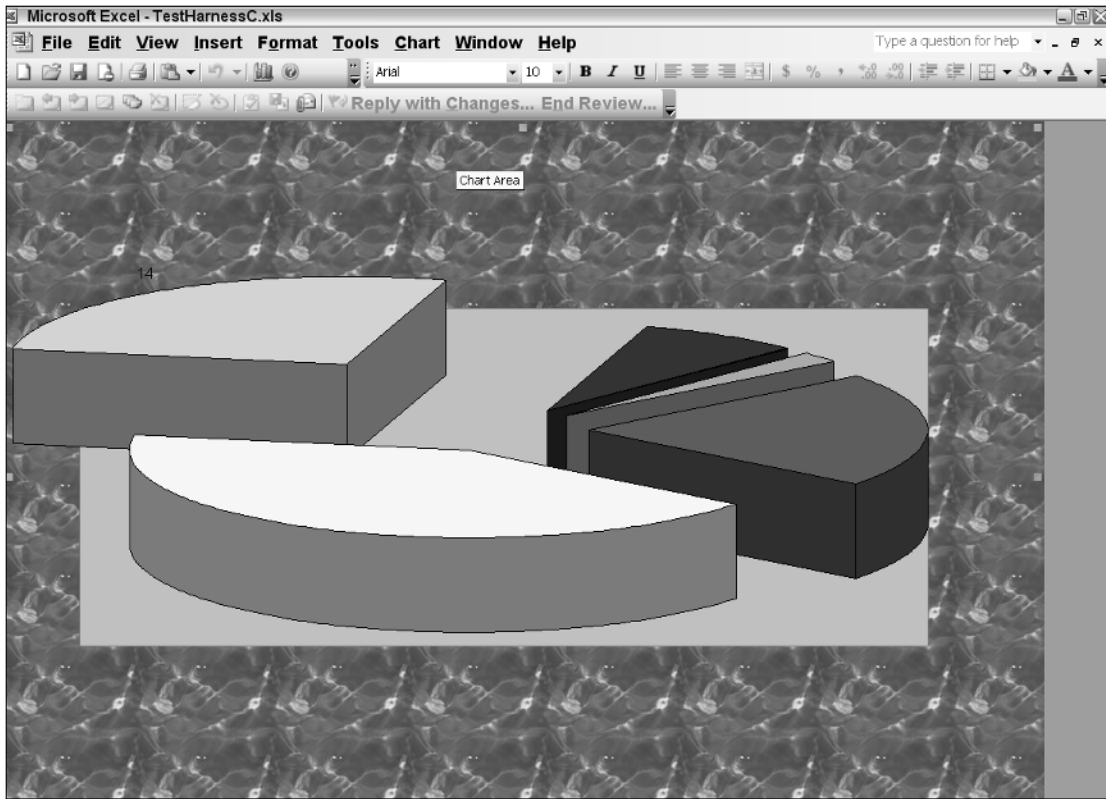


Figure 6-33

Listing 6-27 shows that there is quite a lot going on in the code. First, we create an exploded pie chart. This is a regular 3-D pie chart with one slice of the pie removed. Exploded pie charts are typically used to emphasize some part of the data. Using the usual means, we obtain a series and add a shadow for a professional effect. Then, we select the fourth slice from the point object and cause it to explode by 600 units. See Figure 6-32 for the details. We add a particular color to the exploded slice. Actually, `Color.Wheat` is not particularly impressive and doesn't really stand out, but the point of the exercise is to show that we can target the slice with code.

The code also shows that the background or chartspace of the chart can be customized as well. In this case, we have used a simple image from a `.bmp` file to serve as the background. Expect this to be replaced with a corporate logo in a production environment.

We talked about chart groups in some detail previously and even presented code to isolate chart groups. Following from this discussion, the code gets the default group and applies some customizations. The `GapWidth` property instructs the chart to vary the space between each individual slice as a percentage of the slice width. In a nutshell, the gap between each slice is shrunk or expanded appropriately. Had we not set this property, the pie chart would look less pleasing, since the default `GapWidth` property is rather awkward.

The `FirstSliceAngle` also targets the exploded slice. This setting is specific to pies and doughnut charts that use 3-D rendering. The property sets the angle of the slice to 45 degrees. Angling the slice can be a very effective visual cue for the end user. Finally, the legend is turned off.

Chart Limitations

Most of the documentation available for charting is incomplete on MSDN. There are very few workable examples as of this writing. One way to work around this nuisance is to use an enhancement to the IntelliSense editor found here: www.microsoft.com/downloads/details.aspx?FamilyID=e455f3d6-4177-4876-bcb3-2791625ea7ab&displaylang=en. This enhancement adds valuable information to every property or method that displays in the IntelliSense window. You may use this information to better construct arguments that are passed to poorly documented methods.

In the interest of full disclosure, the VSTO chart object falls short of the mark as compared to other charting packages such as the Office Web Components or the Excel Interop libraries. More is possible with these packages, and there is no cost associated with them; that is, they are available as free downloads. However, you must note that the VSTO chart is relatively new. Expect an improved feature set as this package matures.

One of the stumbling blocks of the VSTO charting package is that it can only be bound to a range in a spreadsheet. After you get over this speed bump, you should realize that this is an inconvenience for which there are quite a few workarounds. In fact, Chapter 3 showed several approaches to loading data from a variety of data sources. Simply use the spreadsheet to load data from these sources, and then bind the Excel range to the chart. It's the long way around the block, but it is sufficient.

Another limitation of the chart control is that there may only be two axes. That is, a third axis representing depth cannot be imposed on a chart. However, this does not mean that the chart cannot render in three dimensions. It simply means that the depth cannot be easily adjusted, since there is no axis available to serve as a point of reference. This chapter has presented several examples of 3-D drawing.

The following hard limits apply to charts that are based on Excel spreadsheets.

- The chart object can only refer to 255 worksheets.
- The data series in a chart cannot exceed 255.
- Each data series cannot exceed 32,000 points. However, the data points in a 3-D chart are limited to 4000.
- The total number of data points for all data series in one chart cannot exceed 256,000.
- The line styles cannot exceed 8.
- The line weights cannot exceed 4.
- There can only be 18 area patterns displayed on screen.
- The total area pattern and color combinations displayed cannot exceed 56,448.

Summary

This chapter focused on the chart terminology from a practical perspective. Rather than throwing out terms and definitions that may or may not sink in, this chapter used the chart wizard and the design-time environment to align key charting terminology with the chart feature set so that a strong bond between the definition and the actual implementation is formed in the mind of the reader that cannot be easily broken or forgotten.

In the majority of cases, you can simply change the chart type to have the chart render a different family of charts. We have noted that the chart can render at least 75 different types of charts, and this is available through a single line of code. There is one exception. Charts with special requirements need a few housekeeping items adjusted before you can simply change the type.

This chapter also showed that chart series are made up of contiguous points. Every point in this series may be targeted through code. You should note that, we have only examined the trivial case where the data set is manageable and the number of points is relatively small. However, real-world charting applications can typically render a great number of points. Code to examine and manipulate each point on the chart surface may incur a performance penalty, so you should exercise due restraint when implementing this approach.

The ability to add shapes are also an interesting addition to VSTO. For the most part, VSTO hides most of the manual labor for this process. Simply call the appropriate method with the required parameter list, and the chart will render the shape. This is a welcomed feature, and one that was long overdue. It also allows the developer to get around some obstacles imposed by VSTO. For instance, a chart can only contain one legend. However, you can now impose your own legend as a rectangle sited appropriately on the chart. You can use the border object to customize the border of the custom legend and you can use the `SeriesCollection` to determine the appropriate `LegendEntries` for the custom legend.

If you must develop charting applications for Windows, this chapter showed you how to develop robust applications that is responsive and chock full of functionality. In short, enterprise-level charting applications are available today through VSTO.

7

Pivot Table Automation

Pivot tables are objects that are able to display data from relational databases and multidimensional data sources, such as Online Analytical Processing (OLAP) cubes. The pivot table provides user interface levers that the end user can use to push and prod the data, exposing relationships and patterns in a way that is simply not possible with any other Windows control. That type of analytic interactivity is also fundamentally different from any other Windows control in that the knowledge worker, not the developer, determines the degree of analysis that is performed on the data.

Pivot tables are new to most .NET developers with no Microsoft Office development experience. However, pivot tables are extremely important for end users who must analyze data. Often, the route of less resistance adopted by .NET programmers is to provide data analysis in the form of datagrids with master/detail relationships. End users accustomed to sophisticated data analysis through the pivot table will find such datagrids immature and clumsy. For instance, the user loses the ability to drill through and to slice and dice data. Although the master/detail `DataGrid` or `GridView` object is not exceedingly complex in implementation, it significantly restricts the user to programmer-imposed functionality. Consequently, the master/detail datagrid offers only limited interactivity compared to the pivot table report application.

The basic idea behind a pivot table is to allow the end user to view a cross-section of the data. For instance, in a financial application, the user may want to view profit versus loss. When anomalies are detected, the knowledge worker has the option to manipulate the pivot table in such a way as to clarify the reasons for these anomalies. The primary lever for this activity is the ability to pivot one axis against the other.

The cycle of pivoting and drilling is practically limitless. But you should note that no work is done by the developer to accomplish this. The functionality is already built into the pivot table component. It supports drill-through functionality, pivoting, drag and drop, filtering, and a host of other functions right out of the box. Most Windows controls can meet this demand.

While a large majority of this functionality is readily available to the knowledge worker, you also have the option to build or customize this functionality through code. This chapter will focus on manipulating pivot tables at design time and runtime. The code will show how easy it is to load data into Pivot tables using a variety of options. Once you understand how to load data, the chap-

ter will show you how to manipulate data and display it to the user. The remainder of the chapter is geared toward showing you how to accomplish common functionality in the pivot table through code. Finally, we examine some advance techniques and strategies for pivot table customizations. The material is presented in such a way as to allow you to grow in comfort with the `PivotTable` object. With the basics in place, you will be able to build your own customizations into any pivot table application.

Design-Time Pivoting

This section will show you how to hook a pivot table object to a data source. Once the data source appears in the control, you will learn how to slice and dice data. It's important to note that pivot table applications are extremely easy to build and involve very little programming effort, as the code will demonstrate. The reason for this is that the user assumes all the responsibilities of data analysis. As a developer, your duty is simply to make the data available to the pivot table object. The user takes over from that point.

Consider an Excel spreadsheet that looks similar to Figure 7-1. An analyst working for the accounts payable department may want to find the customers who placed orders using specific shipping agents on a particular day. Or, consider a fraud department investigating the number of suspicious orders shipped to a particular address. In that case, the information is contained in the body of the pivot table report.

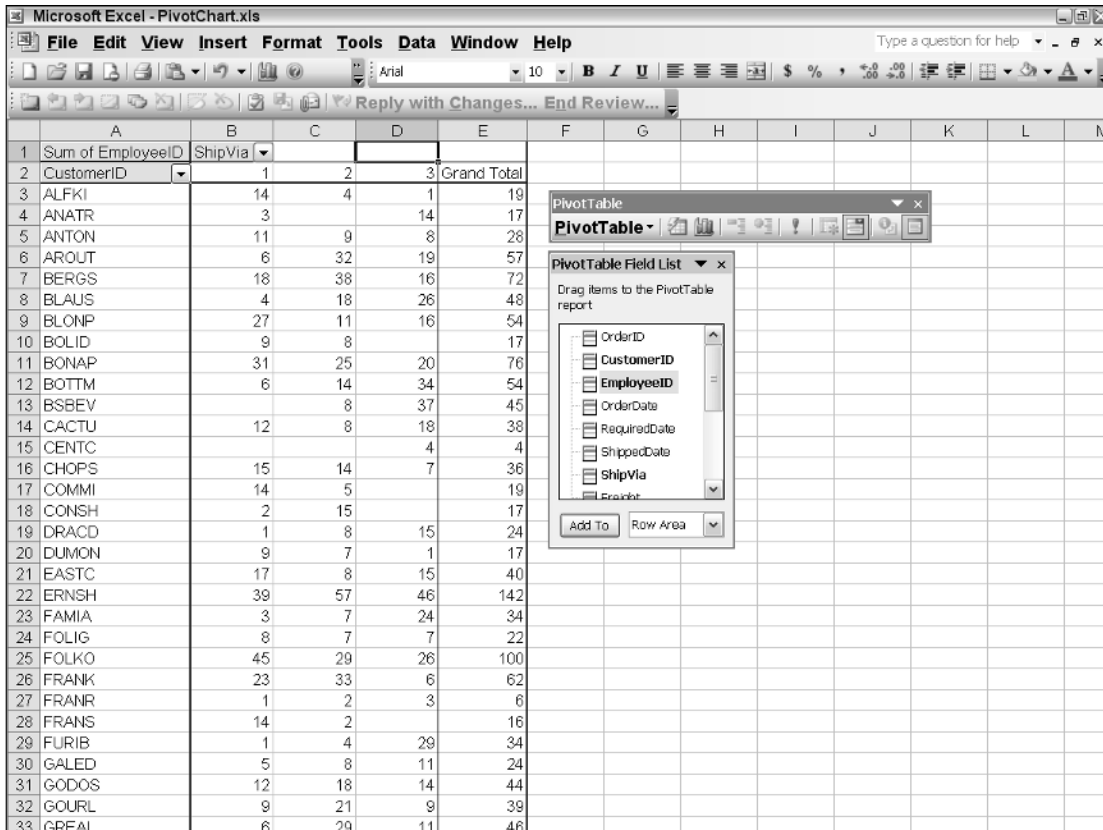


Figure 7-1

In the absence of a pivot table report application, these scenarios and others often require a database person in the background who must understand the request from the IT worker and then translate these requests into SQL query statements that retrieve the data from a data store. You should be able to appreciate the overhead of the database person who is a middleman in the chain. Needless to say, if that database person does not sufficiently understand the requests, incorrect data is harvested often with disastrous results. For instance, imagine an auditor who receives incorrect data from the database middleman while performing a corporate tax audit!

Now, consider the case where the data is available and the pivot table is already set up to connect to that data source. The IT worker can bypass the middleman and immediately begin to push and prod the data into a series of reports. Even in the situation where the data is not immediately available from the data store, this is simply expensed as a one-time setup cost by a developer. It does not need to be redone if requirements for the data change.

VSTO PivotTable Wizard

The VSTO pivot table is intrinsically tied to the Excel object. VSTO does not provide the ability to generate a stand-alone pivot table application. To begin building a pivot table application, first create an Excel project as described in Chapter 1. Name the project Pivot. Navigate to the design view. In the Excel spreadsheet, enter the following data as listed in Figure 7-2. This data will be loaded into the `PivotTable` object. You do not have to enter all the rows, 3 to 5 rows is more than sufficient to create a working example. You can also limit your columns to `EmployeeID`, `ShipVia`, and `CustomerID` fields. If you have access to the `Northwinds` sample database from Microsoft, simply extract these fields into a table. Then, export the entire table to an `.xls` file. This `.xls` file will serve as your data source.

The data in Figure 7-2 was generated from the popular Microsoft Northwind database using appropriate SQL queries. The data was then exported to an `.xls` file. If you do not have the Northwind database system, simply create an empty worksheet and add only the relevant columns with data. Unless specifically stated otherwise, the default dataset for the pivot table list objects for the next few sections will be sourced from the data presented in Figure 7-2.

Next, select Data ⇄ Microsoft Office Excel data. Select Pivot table and Pivot chart report. The action will surface the PivotTable Wizard. Figure 7-3 shows page 1 of the PivotTable Wizard.

Follow the wizard through to load data into the pivot table object. Once the wizard has walked you through the process, the pivot table will be displayed on screen similarly to Figure 7-1. You should note that if your underlying data is not the same as that presented in Figure 7-2, the appearance of the pivot table report will vary. The impact of these differences is negligible.

	A	B	C	D	E	F	G	H	I	J
1	OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress
2	10248	WILMK	5	04/07/1996	01/08/1996	16/07/1996	3	\$32.38	Vins et alc 59 rue de l'Abbaye	
3	10249	TRADH	6	05/07/1996	16/08/1996	10/07/1996	1	\$11.61	Toms Spe Luisenstr. 48	
4	10250	HANAR	4	08/07/1996	05/08/1996	12/07/1996	2	\$65.83	Hanari Ca Rua do Paço, 67	
5	10251	VICTE	3	08/07/1996	05/08/1996	15/07/1996	1	\$41.34	Victuailles 2, rue du Commerce	
6	10252	SUPRD	4	09/07/1996	06/08/1996	11/07/1996	2	\$51.30	Suprêmes Boulevard Tirou, 255	
7	10253	HANAR	3	10/07/1996	24/07/1996	16/07/1996	2	\$58.17	Hanari Ca Rua do Paço, 67	
8	10254	CHOPS	5	11/07/1996	08/08/1996	23/07/1996	2	\$22.98	Chop-suey Hauptstr. 31	
9	10255	RICSU	9	12/07/1996	09/08/1996	15/07/1996	3	\$148.33	Richter Su Starenweg 5	
10	10256	WELLI	3	15/07/1996	12/08/1996	17/07/1996	2	\$13.97	Wellington Rua do Mercado, 12	
11	10257	HILAA	4	16/07/1996	13/08/1996	22/07/1996	3	\$81.91	HILARIÓN Carrera 22 con Ave. Carlos Soubllette	
12	10258	ERNSH	1	17/07/1996	14/08/1996	23/07/1996	1	\$140.51	Ernst Han Kirchgasse 6	
13	10259	CENTC	4	18/07/1996	15/08/1996	25/07/1996	3	\$3.25	Centro cor Sierras de Granada 9993	
14	10260	OLDWO	4	19/07/1996	16/08/1996	29/07/1996	1	\$55.09	Ottilies Kä Mehrheimerstr. 369	
15	10261	QUEDE	4	19/07/1996	16/08/1996	30/07/1996	2	\$3.05	Que Delfoi Rua da Panificadora, 12	
16	10262	RATTC	8	22/07/1996	19/08/1996	25/07/1996	3	\$48.29	Rattlesnak 2817 Milton Dr.	
17	10263	ERNSH	9	23/07/1996	20/08/1996	31/07/1996	3	\$146.06	Ernst Han Kirchgasse 6	
18	10264	FOLKO	6	24/07/1996	21/08/1996	23/08/1996	3	\$3.67	Folk och få Åkergatan 24	
19	10265	BLONP	2	25/07/1996	22/08/1996	12/08/1996	1	\$55.28	Blondel pø 24, place Kléber	
20	10266	WARTH	3	26/07/1996	06/09/1996	31/07/1996	3	\$25.73	Wartian H Torikatu 38	
21	10267	FRANK	4	29/07/1996	26/08/1996	06/08/1996	1	\$208.58	Frankenve Berliner Platz 43	
22	10268	GROSR	8	30/07/1996	27/08/1996	02/08/1996	3	\$66.29	GROSELL 5ª Ave. Los Palos Grandes	
23	10269	WHITC	5	31/07/1996	14/08/1996	09/08/1996	1	\$4.56	White Clox 1029 - 12th Ave. S.	
24	10270	WARTH	1	01/08/1996	29/08/1996	02/08/1996	1	\$136.54	Wartian H Torikatu 38	
25	10271	SPLIR	6	01/08/1996	29/08/1996	30/08/1996	2	\$4.54	Split Rail E.P.O. Box 555	
26	10272	RATTC	6	02/08/1996	30/08/1996	06/08/1996	2	\$98.03	Rattlesnak 2817 Milton Dr.	
27	10273	QUICK	3	05/08/1996	02/09/1996	12/08/1996	3	\$76.07	QUICK-Sti Taucherstraße 10	
28	10274	VINET	6	06/08/1996	03/09/1996	16/08/1996	1	\$6.01	Vins et alc 59 rue de l'Abbaye	
29	10275	MAGAA	1	07/08/1996	04/09/1996	09/08/1996	1	\$26.93	Magazzini Via Ludovico il Moro 22	
30	10276	TORTU	8	08/08/1996	22/08/1996	14/08/1996	3	\$13.84	Tortuga Ri Avda. Azteca 123	
31	10277	MORGK	2	09/08/1996	06/09/1996	13/08/1996	3	\$125.77	Morgenste Heerstr. 22	
32	10278	BERGS	8	12/08/1996	09/09/1996	16/08/1996	2	\$92.69	Berglunds Berguvsvägen 8	
33	10279	LEHMS	8	13/08/1996	10/09/1996	16/08/1996	2	\$25.83	Lehmans Magazinweg 7	

Figure 7-2



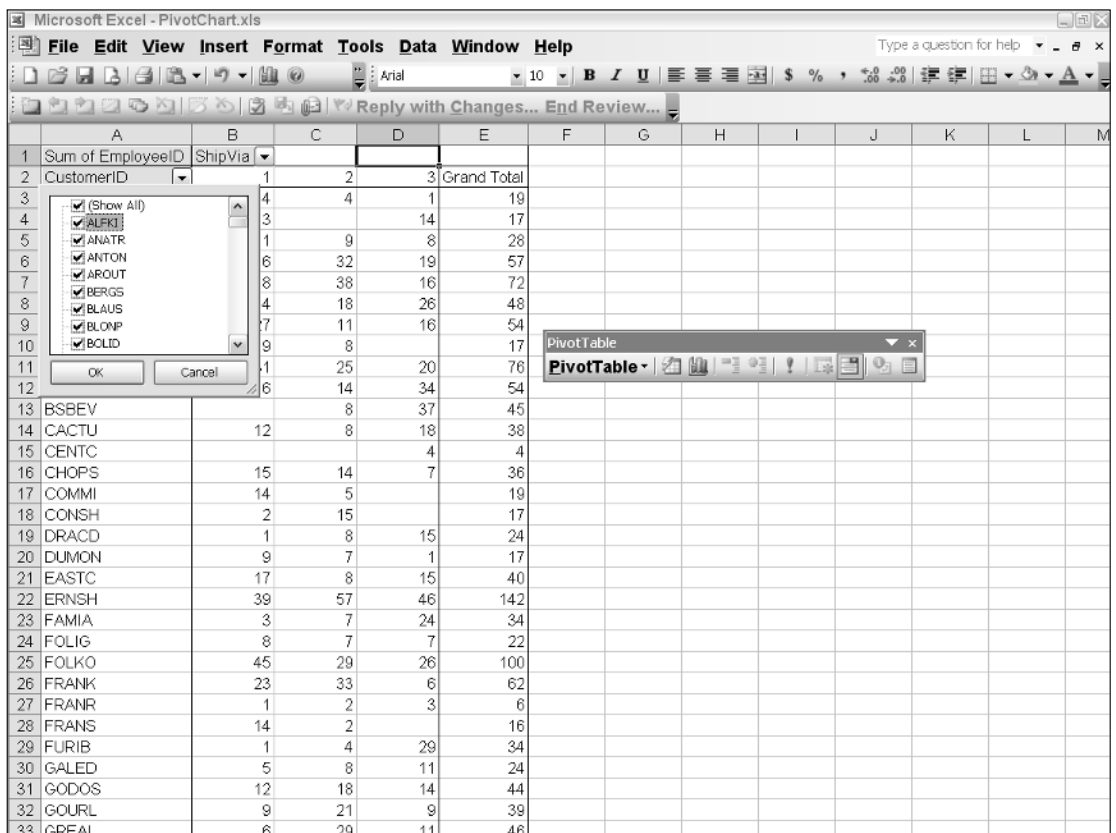
Figure 7-3

Pivoting and Drilling

Let's walk through pivoting and drilling in a pivot table report object. The example proceeds from the previous section and assumes that you have created a pivot table report with data in it. Drag the `CustomerID` field from the pivot field list on to the row axis and drop it on the row axis drop zone so that the row axis contains a number of customer field names. The row axis drop zone is the leftmost area of the pivot table, where data can be released after a drag operation. The row axis drop zone is clearly marked on the pivot table during the design phase.

Next, drag the `orderID` from the pivot field list to the column axis drop zone, and drop it so that the column axis contains a number of orders. Finally, drag the `EmployeeID` field from the pivot field list object onto the data axis drop zone so that values show up in the body of the pivot table. The data axis drop zone is the center portion of the pivot table where data can be released after a drag operation. The data axis drop zone is clearly marked on the pivot table during the design phase. Done!

From this point, if you are satisfied with the report, you can simply select `File` ⇨ `Print` so that the data prints out to the default printer attached to the machine. However, you may also select the down arrow button on any of the axes to display the filter dialog box, as shown in Figure 7-4. Check or uncheck the items in the filter dialog to trim the report to your specifications, and then click `OK`. At this point, your pivot table report will be adjusted to reflect the results of the filter. When you are satisfied, you can choose to export or print the report. The print functionality prints the data in the pivot table; it does not print the pivot table dialogs and objects.



The screenshot shows a Microsoft Excel window titled "Microsoft Excel - PivotChart.xls". The PivotTable is located in the range A1:E33. The row axis is labeled "CustomerID" and the column axis is labeled "EmployeeID". The data is summarized by "Sum of EmployeeID". A filter dialog box is open for the "ShipVia" field, showing a list of shipping methods with checkboxes. The PivotTable data is as follows:

CustomerID	EmployeeID	Sum of EmployeeID
1	2	3
2	4	1
3	3	14
4	1	9
5	6	32
6	8	38
7	4	18
8	7	11
9	9	8
10	1	25
11	6	14
12	8	37
13	12	8
14	8	18
15	4	4
16	15	14
17	14	5
18	2	15
19	1	8
20	9	7
21	17	8
22	39	57
23	3	7
24	8	7
25	45	29
26	23	33
27	1	2
28	14	2
29	1	4
30	5	8
31	12	18
32	9	21
33	6	29

Figure 7-4

Notice that if you double-click on one of the cells in the main data area, the pivot table report will display the items that formed this value. This is drill-through. Drill-through functionality exists because the data displayed in the data area is an aggregation of data from the various fields that form the pivot axis. Drill-through functionality will be discussed in more detail later in the chapter.

Pivot Table Terminology

The pivot table defines its own world complete with special query syntax, object hierarchy, and supporting terminology. While it's not critical to understand the query syntax, Multidimensional Expressions (MDX), it is important to have more than a passing understanding of pivot table terminology. That understanding can go a long way toward carving out functionality from a pivot table application based on customer requirements.

Let's start with a diagram of another pivot table similar to Figure 7-1. However, this pivot table will indicate the key user interface widgets that form part of the pivot table object.

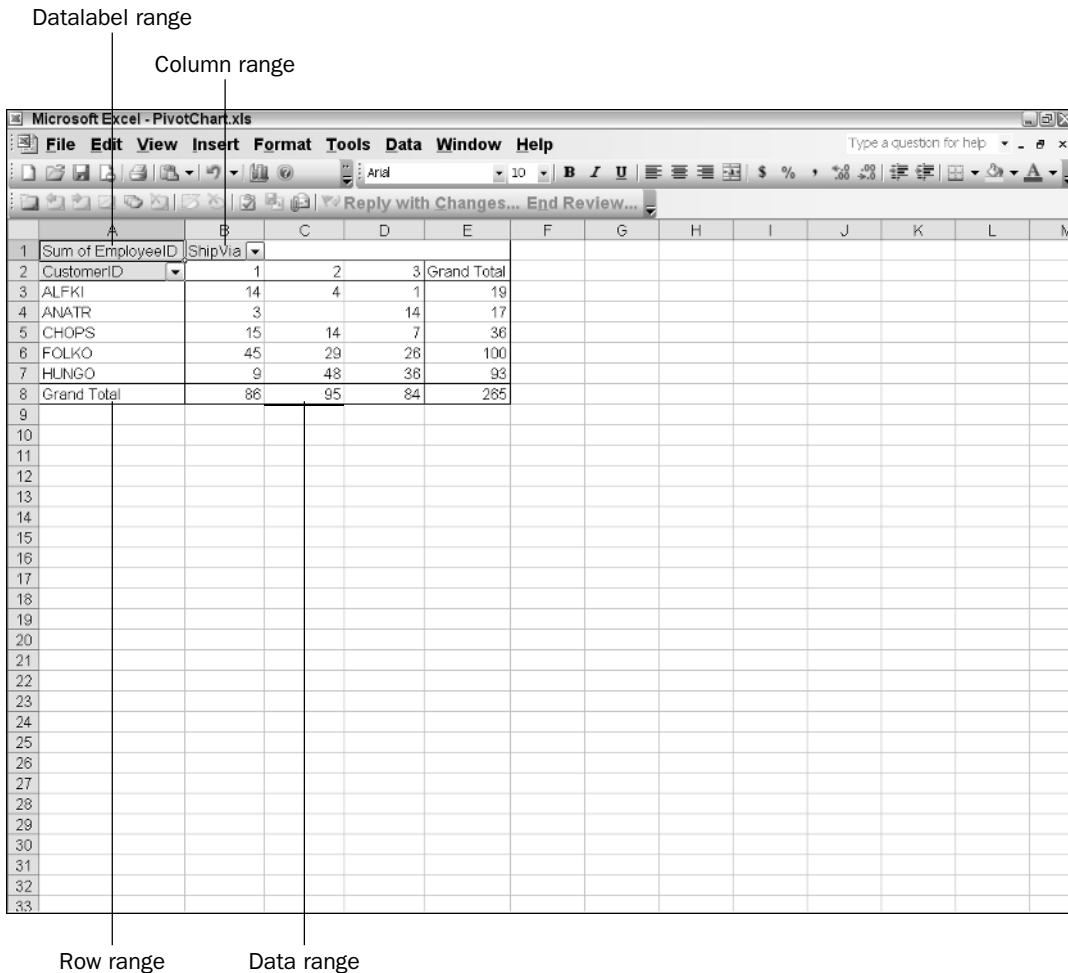


Figure 7-5

For the next few sections, refer back frequently to the user interface widget presented in Figure 7-5 to associate the new terms with their on-screen widgets. To acquaint you with pivot terminology, we will mention appropriate terms as we explore examples.

One point of interest is that the pivot table sits inside the Excel spreadsheet, so you can expect to port most of your knowledge of Excel spreadsheets to the pivot table-reporting application. You may have to learn a few new terms, but hopefully, the learning curve will be significantly reduced. For instance, consider that pivot charts are based on Excel charts and the data displayed in the pivot table is based on Excel Range objects. Charts were covered in Chapter 6. Excel Ranges were covered in Chapters 2 and 3.

PivotCache Object

A typical pivot application can process a lot of data, on the order of several hundred thousand rows of data. The VSTO-based pivot table application handles this data manipulation a bit differently from traditional pivot table applications such as the Office Web Components pivot table. Data that is retrieved from an OLAP cube or other data source is stored in a cache called the *PivotCache*. The cache is queried for data whenever the user begins to manipulate the `PivotTable` object. The VSTO-based pivot table does not requery the original data source by default. However, this behavior may be changed by setting the appropriate connection property of the connection object. In fact, for a very large dataset you should make every effort to reduce the size of the `PivotCache` in order to squeeze performance out of the application since a very large cache can overrun the system resources on the local machine.

On-Line Analytical Processing (OLAP) was first described in the early 1990s and rose to prominence soon after. OLAP defines access to and storage of data into a multidimensional data source. The pivot table is an effective way to model the data in an OLAP cube.

Each pivot table report application contains its associated cache. It's fairly common to have a collection of reports in a single pivot reporting application. Consequently, there may be several `PivotCache` objects. The native store for each `PivotCache` object is the `PivotCaches` object collection. The `PivotCaches` object collection contains the required objects that enable collection iteration and enumeration.

PivotData Objects

The data displayed in a pivot table report is contained in a `pivotdata` object. Through this object, you can massage and manipulate the data inside the pivot table. For instance, you can apply number formats or font customizations to the data if it meets a certain criterion. The data contained in each axis may be addressed through the `pivotdata` object once an appropriate reference is obtained for the axis. Chapters 2 and 3 explained number formats and font customization in great detail. The approach is directly applicable here, since the pivot table exposes its data through Excel Range objects.

Pivot Axis

Pivot axes form an integral part of the pivot table report application. The axes may contain any quantity of data in the form of pivot fields. These axes are used in combination, row versus column, to form a cross-tabulation of data. For instance, a customer name on the row axis may be cross-referenced with a

shipping method on the column axis to generate an intersection of items that were shipped for a particular day. The action of pivoting axes is commonly referred to as *slicing and dicing*. Recall that an example of slicing and dicing was presented at the start of the chapter.

Pivot Fields

Pivot fields are items that form part of the pivot table axis. Pivot fields may be added to axes through code or at design time. The end user also has the ability to add pivot fields to any axis by dragging and dropping the fields onto the appropriate drop zone axis. Pivot fields may be filtered, have formulas applied to them, or customized appropriately. Figure 7-6 shows a pivot field list object.

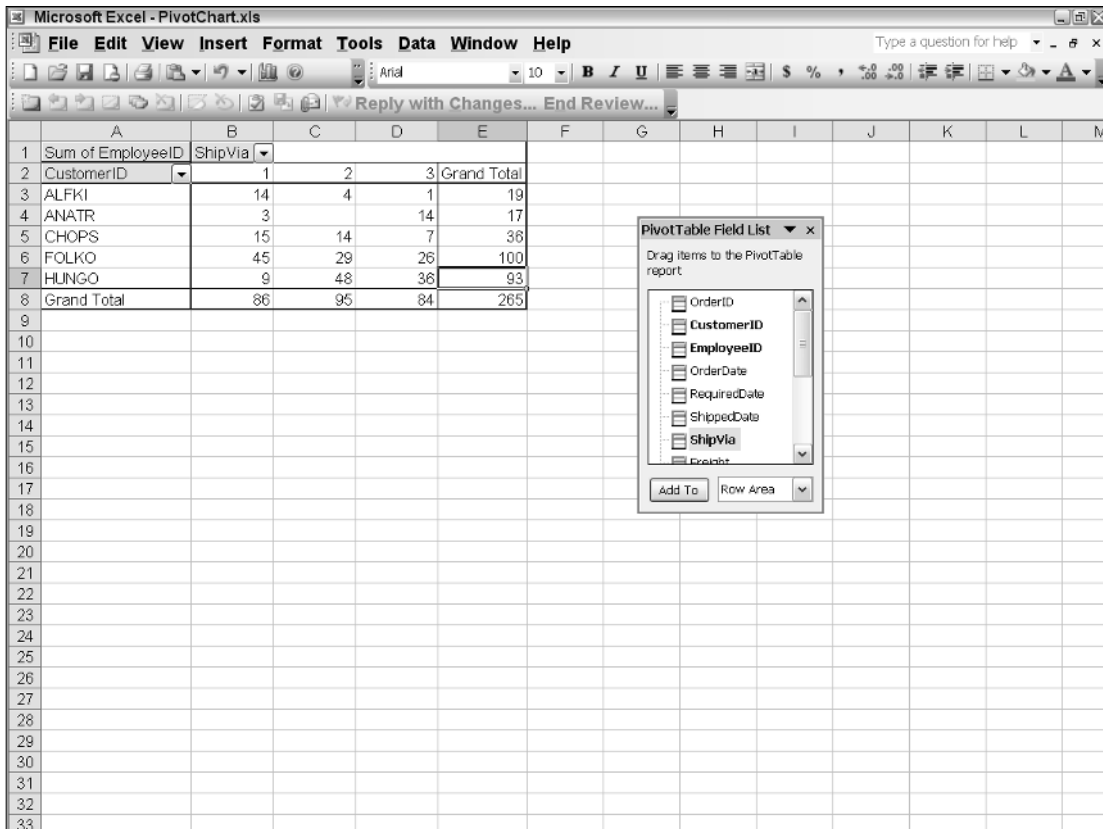


Figure 7-6

Pivot Cell Object

The smallest common denominator of a pivot table report is the pivot cell. The pivot cell holds data in the pivot data axis. The underlying implementation of a pivot cell is built on an Excel Range object. You can use `pivotcell` objects the same way you use Excel Ranges; that is, you can iterate and add customizations on a per cell basis.

Pivot Labels

The pivot labels represent the pieces of text that form part of a pivot axis caption. The underlying implementation of a pivot label is an Excel `Range` object. To iterate the collection of labels, simply obtain a reference to the appropriate label object such as the `DataLabelRange` object. You may then iterate the items in this collection or apply font customizations and styling.

Creating Pivot Tables through Code

There are two approaches to creating pivot tables in VSTO. One approach is to use the `PivotTable Wizard`. The other way is to use the pivot table object attached to the worksheet. Examining the former first, the `PivotTable Wizard` can be invoked through code with a specified parameter list. Once these requirements are satisfied, the `PivotTable Wizard` will handle all the internal plumbing details to create the pivot table. The wizard provides an easy way to build pivot tables because most of the details are handled by the wizard. The wizard is easy to use but the developer has less control over the process. You'll see an example of this in Listing 7-2.

Before presenting an example based on a `PivotTable Wizard`, you should note that the code will assume that the range A1:B4 contains data. If there is no data in the range, the `PivotTable Wizard` will fail. To prepare a range with data suitable for a pivot table, use the code in Listing 7-1.

Visual Basic

```
me.Range("A1").Value = "CustomerID"  
me.Range("A2").Value = "ALFKI"  
me.Range("A3").Value = "WILKH"  
me.Range("A4").Value = "VICTE"  
me.Range("B1").Value = "SHIPVIA"  
me.Range("B2").Value = "3"  
me.Range("B3").Value = "1"  
me.Range("B4").Value = "2"  
me.Range("C1").Value = "EmployeeID"  
me.Range("C2").Value = "23"  
me.Range("C3").Value = "17"  
me.Range("C4").Value = "39"
```

C#

```
this.Range["A1", missing].Value2 = "CustomerID";  
this.Range["A2", missing].Value2 = "ALFKI";  
this.Range["A3", missing].Value2 = "WILKH";  
this.Range["A4", missing].Value2 = "VICTE";  
this.Range["B1", missing].Value2 = "SHIPVIA";  
this.Range["B2", missing].Value2 = "3";  
this.Range["B3", missing].Value2 = "1";  
this.Range["B4", missing].Value2 = "2";  
this.Range["C1", missing].Value2 = "EmployeeID";  
this.Range["C2", missing].Value2 = "23";  
this.Range["C3", missing].Value2 = "17";  
this.Range["C4", missing].Value2 = "39";
```

Listing 7-1 Sample dataset

Listing 7-2 is an example of code to create a pivot table report using the PivotTable Wizard.

Visual Basic

```
Dim table1 as Excel.PivotTable = me.PivotTableWizard(  
    Excel.XlPivotTableSourceType.xlDatabase, me.Range("A1", "B4"),  
    , "PivotTable1", false, false, true, false, missing,  
    , , , Excel.XlOrder.xlDownThenOver)  
  
If Not table1 IsNot Nothing Then  
    MessageBox.Show("The object occupies " +  
        Table1.PivotCache().MemoryUsed.ToString() + " bytes in memory")  
End If
```

C#

```
Excel.PivotTable table1 = this.PivotTableWizard(  
    Excel.XlPivotTableSourceType.xlDatabase, this.Range["A1", "B4"],  
    missing, "PivotTable1", false, false, true, false, missing,  
    missing, false, false, Excel.XlOrder.xlDownThenOver, missing,  
    missing, missing);  
if (table1 != null)  
{  
    MessageBox.Show("The object occupies " +  
table1.PivotCache().MemoryUsed.ToString() + " bytes in memory");  
}
```

Listing 7-2 Pivot table creation using the PivotTable Wizard

As you can see, there isn't a whole lot of complexity involved in the code presented in Listing 7-2. The `Excel.XlPivotTableSourceType` enumeration informs the VSTO runtime to use the appropriate data source to generate the pivot table report. The pivot table can source from five different data sources. Of these, the `Range` object is used in the example. The pivot table report is also flexible enough to source data from a database or array by using the appropriate enumeration. You can use the help documentation to figure out the remaining parameter list.

Additionally, it's probably a better idea to use named parameters in Visual Basic for Listing 7-2. This can improve readability in the code maintenance phase. Unfortunately, there is no performance improvement to be harvested from such an approach.

Surprisingly, this is all the code that is required to generate an enterprise-level pivot table reporting application. In fact, from this point, the user takes over to massage and manipulate the data. Notice how easy such an application is to create when compared with a `GridView` or `DataGrid` application. The pivot table report contains default functionality such as filtering, sorting, drag and drop implementation, and slice and dice functionality. These features have already received hundreds of thousands of hours of testing by Microsoft Product support. There is no need to spend time testing the filtering mechanism to see if it works, for instance.

Visual Studio Tools for Office Visual Basic Edition allows edit and continue in the designer. While debugging, you may simply change the value of a variable on the fly and the Visual Studio runtime will automatically apply these changes to the running code. Visual C# does not allow this magic for VSTO-based applications. You must first stop the executing application, then perform the changes, recompile the code, and reexecute the application.

From this point, the user can choose to export or print this data or drill through it. The drill-through automatically produces a new Excel spreadsheet comprising the fields that make up the cross-section of data. As an example, double-click on any one of the pivot cells that contains data. New data will be displayed in a new spreadsheet similar to Figure 7-7.

	A	B	C	D	E	F	G	H	I	J	K	L
1	OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress	ShipCity	ShipReg
2	11011	ALFKI	3	09/04/1998	07/05/1998	13/04/1998	1	1.21	Alfreds Futterkiste	Obere Str. 57	Berlin	
3	10952	ALFKI	1	16/03/1998	27/04/1998	24/03/1998	1	40.42	Alfreds Futterkiste	Obere Str. 57	Berlin	
4	10702	ALFKI	4	13/10/1997	24/11/1997	21/10/1997	1	23.94	Alfreds Futterkiste	Obere Str. 57	Berlin	
5	10643	ALFKI	6	25/08/1997	22/09/1997	02/09/1997	1	29.46	Alfreds Futterkiste	Obere Str. 57	Berlin	
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21												
22												
23												
24												
25												
26												
27												
28												
29												
30												
31												
32												
33												

Figure 7-7

The second approach to creating a pivot table list object is to use the pivot table attached to the spreadsheet, as shown in Listing 7-3.

Visual Basic

```
Try
    Dim pvtCache As Excel.PivotCache =
Globals.ThisWorkbook.PivotCaches().Add(Excel.XlPivotTableSourceType.xlDatabase, "C:\
Download\VSTO\Book\Chpt_7\Data\Orders.xls!Orders")
    Dim pvtTable As Excel.PivotTable =
pvtCache.CreatePivotTable("[PivotChart.xls]Sheet1!R1C1", "MyPivotTable", True, Excel.X
lPivotTableVersionList.xlPivotTableVersion10)
    Catch ex As Exception
        MessageBox.Show(ex.Message)
End Try
```

C#

```
try
    {
        Excel.PivotCache pvtCache =
Globals.ThisWorkbook.PivotCaches().Add(Excel.XlPivotTableSourceType.xlDatabase,
@"C:\Download\VSTO\Book\Chpt_7\Data\Orders.xls!Orders");
        Excel.PivotTable pvtTable =
pvtCache.CreatePivotTable("[PivotChart.xls]Sheet1!R1C1", "MyPivotTable", true,
Excel.XlPivotTableVersionList.xlPivotTableVersion10);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Listing 7-3 Code to create a pivot table

This code approach is a bit more involved than the approach presented earlier. However, it is more flexible. A `PivotCache` object is used to store a reference to the newly created pivot table. The pivot table is created by calling the `Add()` method of the `PivotCaches()` object. The `Add` method accepts two parameters. The first indicates the type of data source and the second indicates the path to the datasource.

After the `Add()` call is executed, a `PivotCache` object is defined and added to the `PivotCache` collection. The subsequent call to `CreatePivotTable()` actually creates a `PivotTable` object and fills the pivot table with data. The `CreatePivotTable()` call enforces incorrect parameters by throwing exceptions, so you should typically wrap the code to create a pivot table in exception handling code. The exception handling code is not presented here.

The `CreatePivotTable()` method call can be used to connect a pivot table to a data source that is a database, an URI that points to an Internet source, or an OLAP cube. XML files cannot be loaded through this mechanism. XML files must first be loaded into the Excel spreadsheet. Once the data is in the spreadsheet, you can use the code in Listing 7-3 to load data into the pivot table report. Note also that for cross-domain calls or calls that must run remotely, such as loading data from an Internet resource, the correct permissions must be configured. The process responsible for running the pivot table report application on the remote server does not contain the necessary permissions to load data remotely.

In both pivot table loading scenarios, if a pivot table name identifier is not passed in, the pivot table is created with a default pivot table name identifier equal to `PivotTable1`. You can use this identifier to retrieve a reference to the existing pivot table in code. This is the preferred method. However, you can always use an index into the pivot tables object collection to retrieve a pivot table instance. It's important to note that an exception is thrown if a named identifier does not exist.

The call to `CreatePivotTable` must use this special syntax: "`[PivotChart.xls]Sheet1!R1C1`", where `PivotChart.xls` represents the name of the Excel project. The name of the Excel project is optional. This must be followed immediately by the Excel spreadsheet sheet name and row/column addressing. Deviation from this strict pattern results in an exception being thrown, indicating that the parameter is incorrect. The error message does not explicitly indicate which one of the four parameters is the culprit. It is troubling that such exceptions cannot be more specific, to say the least.

Formatting Pivot Data

Assuming that your users require a bit more information or formatting on the pivot table report, that sort of functionality is relatively simple to implement. The key to implementing this functionality successfully is to know the objects in the pivot table list that are responsible for displaying data in the pivot table report user interface widget. Several key objects were presented at the start of this chapter for this very purpose.

As an added bonus, the pivot table returns most, if not all, of its data objects as Excel spreadsheet ranges. So, you will be able to use the knowledge gathered in Chapters 2 and 3 to apply further customization when manipulating pivot table report ranges. The next few sections show how to accomplish this. For each example, you will notice a common theme: find the object of interest, retrieve it as an Excel `Range`, and iterate that `Range`.

Font Object Customizations

Building on the premise of Excel `Ranges` presented in Chapters 2 and 3, the following example demonstrates how to add a bold font to each item in the row axis. Listing 7-4 shows the code.

Visual Basic

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Dim table As Excel.PivotTable = Globals.Sheet1.PivotTables("MyPivotTable")

    If (table IsNot Nothing) AndAlso (table.RowRange IsNot Nothing) AndAlso
table.RowRange.Cells.Count > 0 Then
        Dim rng As Excel.Range = table.RowRange
        Dim i As Integer
        For i = 1 To rng.Cells.Count - 1 Step i + 1
            Dim cellRange As Excel.Range = DirectCast(rng.Cells(i, 1),
Excel.Range)
            cellRange.Font.Bold = True
        Next
    End If
End Sub
```

C#

```
private void button1_Click(object sender, EventArgs e)
{
    Excel.PivotTable table = Globals.Sheet1.PivotTables("MyPivotTable") as
Excel.PivotTable;
    if (table != null && table.RowRange != null &&
table.RowRange.Cells.Count > 0)
    {
        Excel.Range rng = table.RowRange;
        for (int i = 1; i < rng.Cells.Count; i++)
        {
            Excel.Range cellRange = (Excel.Range)rng2.Cells[i, 1];
            cellRange.Font.Bold = true;
        }
    }
}
```

Listing 7-4 Font customization in a pivot table report

By extension, you should be able to modify the code in Listing 7-3 to apply font customization to the column and data axis. All that is needed is a reference to `table.ColumnRange` instead of `table.RowRange` to cause the code to work correctly.

Label Format Customization

As the term implies, label formatting applies formatting attributes to the `Label` objects in a pivot table report application. Recall from earlier in the chapter that the `DataLabelRange` displays the data labels on a pivot table report. The approach in Listing 7-3 can be modified to iterate the `DataLabelRange` as well. Listing 7-5 shows a modification on the code presented in Listing 7-3. It may surprise you to know that label formatting is not necessarily limited to aesthetic adjustments of pieces of text.

Visual Basic

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
Dim table As Excel.PivotTable = Me.PivotTables("myPivotTable")
    If table IsNot Nothing Then
        If table.DataLabelRange IsNot Nothing Then
            Dim rng As Excel.Range = table.DataLabelRange
            rng.Value = rng.Value & "is a field on the pivot table object"
            rng.Speak()
            table.RefreshTable()
        End If
    End If
End Sub
```

C#

```
private void button1_Click(object sender, EventArgs e)
{
    Excel.PivotTable table = this.PivotTables("PivotTable1") as
Excel.PivotTable;
    if (table != null && table.DataLabelRange != null && table.
DataLabelRange.Cells.Count > 0)
    {
```

```

        Excel.Range rng = table.DataLabelRange;
        for (int i = 1; i < rng.Cells.Count; i++)
        {
            Excel.Range cellRange = (Excel.Range)rng2.Cells[i, 1];
            cellRange.Font.Bold = true;
        }
    }
}

```

Listing 7-5 Label formatting in a pivot table report

As the code in Listing 7-5 shows, a reference to the active `PivotTable` object is obtained using a named index. Then, the `DataLabelRange` property of the `PivotTable` object is used to obtain a suitable reference. However, what comes next underscores the power and flexibility of Microsoft Office. Using the reference to the `Range` object, a value is inserted into the caption. Then, Microsoft Office's speech software kicks in to voice the contents of the range. This is fantastic!

For performance reasons, you should not apply formatting attributes to axes that contain a large number of values. Instead, you should try to filter or restrict the axes to the important values before applying the filter. You may also limit the code to iterating the visible range of the Excel spreadsheet.

Speech software is more than just a nicety; it represents a growing niche market where software is customized for the hearing impaired or visual impaired. The `Speak` method causes the cells of the target range to be spoken in row or column order. Microsoft Office automatically installs the speech assemblies upon first use. If your speakers are turned up and your sound card is functioning correctly, you will hear the label being spoken. The `RefreshTable` method is called to force the column captions to display with the added text. It is not required for the `Speak` method to function correctly.

Obviously, this is only the tip of the iceberg, and a lot more is possible. However, this example is crafted to show that such technology is now possible and is easy to implement. Aside from this functionality, the caption on the data label can also be oriented or formatted appropriately. Such functionality is trivial to implement and simply involves setting the appropriate property or calling the correct method.

Style Object Customization

Style customization in the pivot table is implemented through the Excel `Range` object. This is great news because it means that you do not have to learn new material in order to implement styles. Consider the code in Listing 7-6, which applies a particular style to the row range. The actual style code is based on code presented in Chapter 3.

Visual Basic

```

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Dim table As Excel.PivotTable = DirectCast(Me.PivotTables("PivotTable1"),
Excel.PivotTable)
    If table IsNot Nothing Then
        If table.RowRange IsNot Nothing Then
            Dim rng As Excel.Range = table.RowRange

```

```
        Dim style As Excel.Style = DirectCast(rng.Style, Excel.Style)
        style.Font.Name = "Arial"
        style.Font.Size = 10
        style.Font.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Black)
        style.Interior.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Beige)
        style.Interior.Pattern = Excel.XlPattern.xlPatternSolid
    End If
End If
End Sub
```

C#

```
private void button1_Click(object sender, EventArgs e)
{
    Excel.PivotTable table = Globals.Sheet1.PivotTables("myPivotTable") as
Excel.PivotTable;
    if (table != null)
    {
        if (table.RowRange != null)
        {
            Excel.Range rng = table.RowRange;
            Excel.Style style = rng.Style as Excel.Style;
            style.Font.Name = "Arial";
            style.Font.Size = 10;
            style.Font.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Black);
            style.Interior.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Beige);
            style.Interior.Pattern = Excel.XlPattern.xlPatternSolid;
        }
    }
}
```

Listing 7-6 Style object customization in pivot table reports

The code in Listing 7-6 shows that a button's Click event contains the code that implements styles. To run this example correctly, you can simply place a button on the Excel spreadsheet. If you double-click that button, a Button1_Click event handler will be generated for you automatically. Then, you can place the code in Listing 7-6 in the body of the handler. A more sophisticated approach is to place the button in an Excel actions pane as described in Chapter 3.

Following from Listing 7-6, a style object is retrieved from the row Range object. The row Range object returns a reference for the entire row of the pivot table. Notice also that the table variable reference is retrieved for the PivotTable1 object. Consequently, formatting of the style object is applied to the rows of the pivot table. You should expect to see the customizations applied to the entire PivotTable object even though you are using a style for the row range.

Data Formatting with NumberFormats

The large majority of pivot reports deal with numeric data. When you display data, the accounting format is usually the most appropriate because it allows numerical data to be presented consistently and neatly. In addition to supporting the accounting format, the `PivotTable` object is also able to display numeric data based on a `NumberFormat` mask. Listing 7-7 provides a quick example of `NumberFormat` usage since this topic has been covered in several chapters.

Visual Basic

```
Dim table As Excel.PivotTable = DirectCast(Me.PivotTables("MyPivotTable"),
Excel.PivotTable)
    If table IsNot Nothing Then
        Dim pField As Excel.PivotField =
DirectCast(table.PivotFields("Customerid"), Excel.PivotField)
        table.AddDataField(table.PivotFields("Freight"), "Sum of Freight",
Excel.XlConsolidationFunction.xlSum)
        table.RowGrand = False
        Dim rng As Excel.Range = table.RowRange
        rng.NumberFormat = "General"
    End If
End Sub
```

C#

```
private void Formatter()
{
    Excel.PivotTable table = this.PivotTables("MyPivotTable") as
Excel.PivotTable;
    if (table != null)
    {
        Excel.PivotField pField = table.PivotFields("Customerid") as
Excel.PivotField;
        table.AddDataField(table.PivotFields("Freight"), "Sum of Freight",
Excel.XlConsolidationFunction.xlSum);
        table.RowGrand = false;
        Excel.Range rng = table.RowRange;
        rng.NumberFormat = "General";
    }
}
```

Listing 7-7 NumberFormat customization with pivot tables

Listing 7-7 shows some code to add a pivot field to a pivot table report application. As usual, a reference to the active pivot table report is retrieved using the name identifier. Then, a reference to the pivot field titled `CustomerID` is retrieved. The value is case insensitive. At this point, the code adds a data field called `freight` to the report and titles it appropriately. The data field is expected to appear in the body of the pivot table object, since it is added using a `DataField` method call.

One way to execute the code in Listing 7-7 is to first generate a pivot table report and drop the required `CustomerID` field onto the row axis. Then drop the `ShipVia` field onto the column axis. You can simply choose to invoke the code in Listing 7-7 at this point, and the data field should automatically be populated with aggregates of freight information. However, if the pivot table report application already contains data values, the new freight columns are merged in neatly for each customer. Consider Figure 7-8.

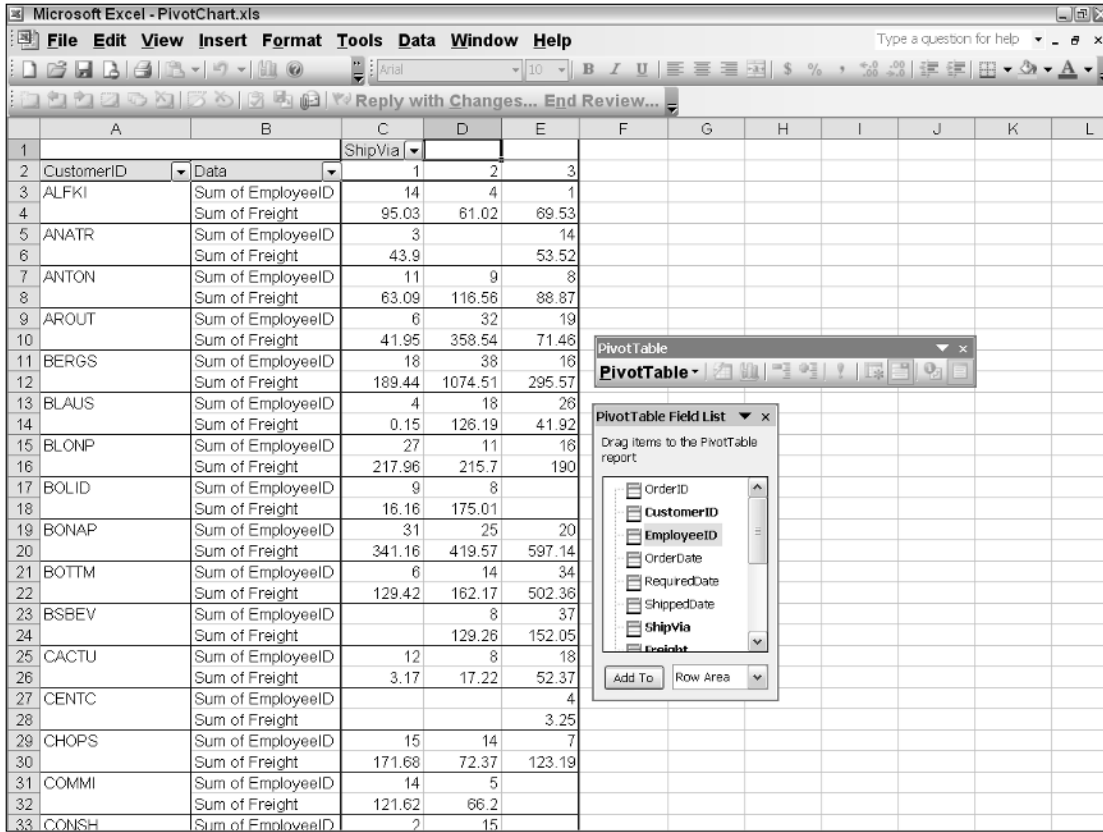


Figure 7-8

The observant reader should recognize the existing code. Notice that `NumberFormat` is set to `General`. As you may recall, the `General` type serves as a reset for the `NumberFormat` display. For a review on number formats, see Chapter 3.

The decision to reuse the Excel Range in a pivot table application is a good design choice because it serves to drastically reduce the volume of new information required to be proficient at pivot table applications.

Pivot Axis Iteration

As you know, the pivot table object consists of row and column axes. The axes are an integral part of the pivoting mechanism. A definition of pivot axis is presented at the start of the chapter. Building from this definition, you can enhance the functionality of the slicing mechanism significantly by customizing these axes. For instance, you may provide totals or formatting functionality on a specified axis. Listing 7-8 shows us some basic code to probe an axis.

Visual Basic

```

Private Sub IterateAxes()
    Dim table As Excel.PivotTable =
DirectCast(Me.PivotTables("MyPivotTable"), Excel.PivotTable)
    If ( table IsNot Nothing) Then
        Dim rng As Excel.Range = table.RowRange
        Dim cell As Excel.Range = Nothing
        Dim rowCount As Integer = rng.Rows.Count
        Dim row As Integer
        For row = 1 To rowCount- 1 Step row + 1
            cell = DirectCast(rng.Cells(row, 1), Excel.Range)
            If ( cell IsNot Nothing) Then
                cell.AddComment(" This is some text added to the field " +
cell.Text.ToString())
                cell.Interior.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Wheat)
                cell.Font.Underline = True
            End If
        Next
    End If
End Sub

```

C#

```

private void IterateAxes()
{
    Excel.PivotTable table = this.PivotTables("MyPivotTable") as
Excel.PivotTable;
    if (table != null)
    {
        Excel.Range rng = table.RowRange;
        Excel.Range cell= null;
        int rowCount = rng.Rows.Count;
        for(int row = 1; row < rowCount; row++)
        {
            cell = rng.Cells[row, 1] as Excel.Range;
            if (cell != null)
            {
                cell.AddComment(" This is some text added to the field " +
cell.Text);
                cell.Interior.Color =
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Wheat);
                cell.Font.Underline = true;
            }
        }
    }
}

```

Listing 7-8 Iterating pivot axes

In Listing 7-8, a reference to the axis is obtained using the `table.RowRange` object. If you needed to iterate the `Column`, `Data`, or `DataLabel` axes, you only need to use the appropriate reference here. The formatting and customization approaches learned in Chapter 3 are directly applicable here as well. Since we are iterating a particular column, we can hard code the column to a value of 1 and retrieve a refer-

ence to it via the `Cells` property. Once you have a reference to the `Cells` property, you can add a comment, underline the text, and add the appropriate color as necessary.

It should come as no surprise that the code to iterate the axes in a pivot table list object is modeled on the `Excel Range`. However, if you're already accustomed to pivot tables in VBA for instance, the specific axes objects that allow for pivot axis iteration are no longer available. These objects have been replaced. This is good news for new adopters of this technology, but it can be frustrating for VBA developers moving to this new platform. One issue for VBA developers is that the macro code generated in Excel is no longer directly portable. Some modifications must be made.

Pivot Filtering

One of the benefits of a pivot table is the ability to automatically filter data. The filter mechanism is already internally implemented so no work is required on your part. However, you may still require the ability to enhance that filter mechanism. The code in Listing 7-9 displays the top five items for a pivot table report.

Visual Basic

```
Private Sub Top5()  
    Dim table As Excel.PivotTable =  
DirectCast(Me.PivotTables("MyPivotTable"), Excel.PivotTable)  
    If ( table IsNot Nothing) Then  
        Dim pField As Excel.PivotField =  
DirectCast(table.PivotFields("Customerid"), Excel.PivotField)  
        pField.AutoShow(Excel.Constants.xlAutomatic, Excel.Constants.xlTop, 5,  
"Sum of EmployeeID")  
    End If  
End Sub
```

C#

```
private void Top5()  
{  
    Excel.PivotTable table = this.PivotTables("MyPivotTable") as  
Excel.PivotTable;  
    if (table != null)  
    {  
        Excel.PivotField pField = table.PivotFields("Customerid") as  
Excel.PivotField;  
        pField.AutoShow((int)Excel.Constants.xlAutomatic,  
(int)Excel.Constants.xlTop, 5, "Sum of EmployeeID");  
    }  
}
```

Listing 7-9 Pivot filtering

Listing 7-9 shows code that first retrieves a reference to the pivot table report using the named identifier. From there, the `AutoShow` method is called, indicating the number to be displayed and the column to be filtered. The filter functionality is not necessarily limited to the pivot axes; it may be used to filter on pages as well. One scenario where this is helpful is in restricting the number of items that appear on the pivot table axis by default. You should note that the user can easily reproduce this functionality by using the filter drop-down list.

The code is successfully executed if a column captioned “Sum of EmployeeID” is found; otherwise, an exception is thrown. The net effect of the code restricts the data shown in the `EmployeeID` axis to the top five values. Consequently, the Row Axis is adjusted to show five values. This is an effective way to filter away unwanted data. The filter mechanism is also powerful enough to allow a range of other functionality. Consult the help documentation frequently to improve on the filtering experience for the user.

Adding Miscellaneous Columns

One enhancement to a pivot table report is the ability to add additional columns. The additional column may represent any sort of functionality or feature that the user requires. One common request is an extra column that counts the number of items in a specific field. Listing 7-10 shows some code to achieve this.

Visual Basic

```
Private Sub OrderItems ()
    Dim table As Excel.PivotTable =
DirectCast(Me.PivotTables("MyPivotTable") , Excel.PivotTable)
    If table IsNot Nothing Then
        Dim pField As Excel.PivotField =
DirectCast(table.PivotFields("Customerid"), Excel.PivotField)
        table.AddDataField(table.PivotFields("OrderDate"), " Ship by XMas
", Excel.XlConsolidationFunction.xlCount)
    End If
End Sub
```

C#

```
private void OrderItems()
{
    Excel.PivotTable table = this.PivotTables("MyPivotTable") as
Excel.PivotTable;
    if (table != null)
    {
        Excel.PivotField pField = table.PivotFields("Customerid") as
Excel.PivotField;
        table.AddDataField(table.PivotFields("OrderDate"), "Ship by XMas",
Excel.XlConsolidationFunction.xlCount);
    }
}
```

Listing 7-10 Addition of custom columns

A reference to the active pivot table report is retrieved by using the name identifier that was used initially to create the report. If the object is not null, we simply add a new column called `OrderDate` with an appropriate caption “Ship by Xmas” and a predefined function that tells the pivot table how to handle the

data in this new field. In this case, the instruction is to count the number of items. If you care to run the code, you will notice that the pivot table report automatically totals each row with a grand total in the body of the pivot table report. To turn this functionality off, simply set the `table.RowGrand` to `false`.

Be aware though, that new data fields are based on columns that exist in the underlying data source, otherwise an exception is thrown. While the effect of Listing 7-10 is to add a data field to the pivot table report, you may use the `AddField` method of the pivot table object instead of the `AddDataField` method to add new columns to the row, column, or page field axes. The implementation is trivial and is left as an exercise to the reader.

The `Excel.XlConsolidationFunction` shown in Listing 7-10 contains 12 possible values. These values are applied to the fields that compose the column. In the case of Listing 7-10, each field is summed as it is rendered. You should note that the fields in a pivot table report must necessarily be unique, since these fields and their labels serve as internal identifiers. Addition of duplicate fields results in a runtime exception. Such an exception can be replicated easily by running the code in Listing 7-10 a second time.

Pivot Table Events

At the risk of sounding repetitive, you should notice that events are handled consistently across the VSTO object model. The basic approach remains the same. Find an event of interest, subscribe to it, and place code in the event handler. The event handler will be called, and the code will be executed whenever the event fires. As always, the variables that refer to events need to be declared global in scope in order for the event handler to work consistently throughout the application's lifetime.

Listing 7-11 shows one approach to pivot table events.

Visual Basic

```
Sub Worksheet1_PivotTableUpdate(ByVal Target As Excel.PivotTable) _
    Handles Me.PivotTableUpdate
    MsgBox("The PivotTable connection update event has fired.")
End Sub
```

C#

```
private void WorksheetPivotTableUpdate()
{
    this.PivotTableUpdate +=
        new Excel.DocEvents_PivotTableUpdateEventHandler(
            Worksheet1_PivotTableUpdate);
}

void Worksheet1_PivotTableUpdate(Excel.PivotTable Target)
{
    MessageBox.Show("The PivotTable connection update event has fired ");
}
```

Listing 7-11 Pivot table list object event

The `PivotTableUpdate` event fires after the pivot table report is updated on the Excel worksheet. The VB portion of the code does not need to explicitly add an event handler in code. The event can be hooked up using the event property in the spreadsheet designer. Unfortunately, the pivot table does not expose a rich event model. In fact there are only a handful of events that are exposed and most of them are tied to the worksheet as in the case of Listing 7-11.

Having said this, it's important to note that the `Range` object that forms the underpinnings of a large number of pivot table objects expose a lot of events that may be customized. You can take advantage of these events of the Excel `Range` objects to customize the pivot table further.

Creating Pivot Charts

Pivot charts first appeared with the release of Excel 2000. As the name implies, the pivot chart object represents a chart bound to a pivot table application. The data is automatically sourced from the pivot data. Chapter 6 presented the VSTO chart and several approaches to loading, customizing, and displaying the data in the user interface. You should not be surprised to learn that the VSTO chart component is simply reused to create the pivot chart. With that knowledge, it should be fairly easy to construct a chart based on a pivot table and customize the data for display. You only need to learn how to connect a chart to a pivot table.

Listing 7-12 shows how to connect a chart to a pivot table.

Visual Basic

```
Dim table As Excel.PivotTable = DirectCast(Me.PivotTables("MyPivotTable"),
Excel.PivotTable)
    Dim xlChart As Excel.Chart = CType(Globals.ThisWorkbook.Charts.Add(),
Excel.Chart)
    xlChart.SetSourceData(table.TableRange1, Excel.Constants.xlColumn)
```

C#

```
Excel.PivotTable table = this.PivotTables("MyPivotTable") as Excel.PivotTable;
    Excel.Chart xlChart =
    (Excel.Chart)Globals.ThisWorkbook.Charts.Add(missing, missing, missing, missing);
    xlChart.SetSourceData(table.TableRange1, Excel.Constants.xlColumn);
```

Listing 7-12 Creating Pivot charts

As usual, the code to create a chart based on a pivot table simply creates a stand-alone chart and then sets the source data as a pivot table range. The results are shown in Figure 7-9.

There isn't much point in repeating the concepts and strategies presented in Chapter 6. However, let's make a few additional notes before leaving. Notice that this approach does not allow us to relocate the pivot chart on the Excel spreadsheet surface. The pivot chart is created on the active sheet by default. In spite of this, you can use the `location` property of the chart object to move the chart to another worksheet after it has been rendered to the Excel spreadsheet.

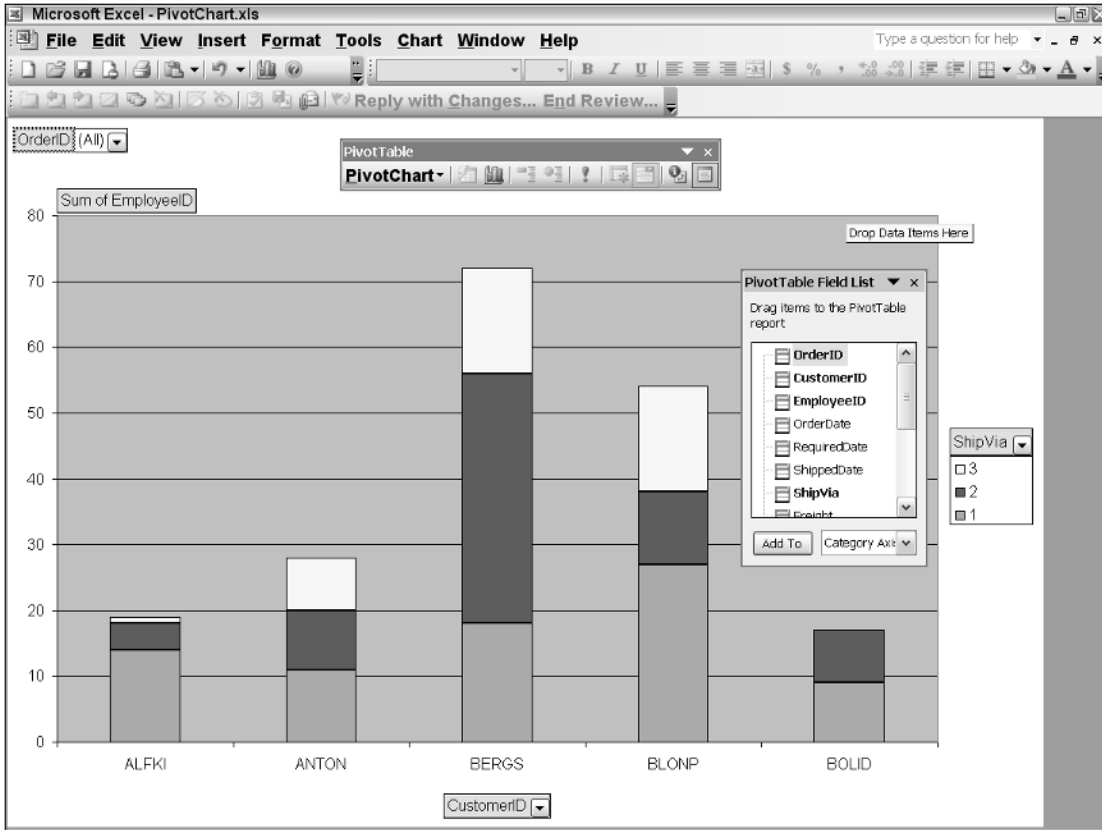


Figure 7-9

If this still does not suit your requirements, you should use the Chart Wizard to create the pivot chart. Set the source property as shown above and use the numerical parameters left, top, width, and height to position the chart appropriately on the Excel spreadsheet. Listing 7-13 contains a brief example.

Visual Basic

```
Dim table As Excel.PivotTable = DirectCast(Me.PivotTables("MyPivotTable"),
Excel.PivotTable)

Dim ChartObjects As Excel.ChartObjects = CType(Me.ChartObjects(),
Excel.ChartObjects)
Dim chartObject As Excel.ChartObject = ChartObjects.Add(1,20,250,250)
chartObject.Chart.ChartWizard(table.TableRange2,
Excel.XlChartType.xl3DColumn, ExtraTitle:="Chart")
```

C#

```
Excel.PivotTable table = this.PivotTables("MyPivotTable") as
Excel.PivotTable;

Excel.ChartObjects ChartObjects =
(Excel.ChartObjects)this.ChartObjects(missing);
```

```

Excel.ChartObject chartObject = ChartObjects.Add(1, 20, 250, 250);

chartObject.Chart.ChartWizard(table.TableRange2, Excel.XlChartType.xl3DColumn,
missing, missing, missing,
missing, missing, " Chart", missing, missing, missing);

```

Listing 7-13 Pivot chart application based on PivotChart Wizard

As you can see, this approach allows the chart position to be precisely controlled. Additional functionality through the use of properties and methods can be applied by using the `chartObject` variable.

You may notice that Listing 7-13 uses `table.TableRange2`, whereas Listing 7-12 uses `table.TableRange1` as the first parameter call to link the chart with the pivot table. Since both calls implicitly return ranges, what possible difference could there be? As it turns out, the `TableRange1` retrieves a pivot table range that does not include the page field object of the pivot table report. The `TableRange2` object returns the entire pivot table report object, including the page field.

From the user's perspective, there is no perceived difference between these two approaches to load data into the pivot table report application. Internally, one simply provides location benefits over the other.

Pivot Table Limitations

Pivot tables do have certain limitations imposed on them. However, it is interesting to note that most of these limitations are imposed by the housing Excel object. For instance, Microsoft Excel does impose a limit on the number of cells that may be rendered to its surface. Additionally, this limit also applies to the number of characters that a cell may contain. These limits necessarily constrain the pivot table object, since it runs embedded inside the Excel spreadsheet.

As pointed out in this chapter, the pivot table is implicitly tied to the Excel spreadsheet. The benefit of this implicit relationship means that the underpinnings of many of the objects that form part of the pivot table can be built on the Excel `Range` object. Several examples of row axis manipulation show that the code is greatly simplified and requires less learning. On the other hand, this relationship necessarily implies that VBA code containing pivot tables cannot be directly ported to VSTO.

Pivot tables are built on the MDX language. Although you can accomplish a lot without acknowledging MDX's importance, advanced customization of a pivot table application will require an in-depth knowledge of MDX. Your application can gain a significant performance boost by manipulating the details of a pivot table through MDX. You can also increase the complexity and display functionality of the data in the pivot table object through the MDX. However, learning to use MDX effectively is not a trivial process.

Pivot table applications can typically crunch through a large number of data items. These large datasets can sometimes strain system resources on the machine. There are also other instances when the pivot table can run out of memory or throw a runtime exception. Let's briefly examine some of these hard limits.

- ❑ The number of unique items per field cannot exceed 32,500.
- ❑ The number of page fields in a pivot table report cannot exceed 256.
- ❑ The number of data fields in a pivot table report cannot exceed 256.
- ❑ The number of reports in a pivot table report is limited only by system resources.

- ❑ The number of rows or columns in pivot table report is limited only by system resources.
- ❑ The number of calculated items in a pivot table report is limited only by system resources.
- ❑ There are no fixed maximum or minimum sizes for data in any of the axes.

Having difficulty figuring out the object hierarchy? One good approach is to use the macro recorder. From Excel, select Tools ⇨ Macro. Then select Start recording. Enter data into the spreadsheet and select the PivotTable Wizard. Use the wizard to construct a pivot chart based on the data in the range. Once the wizard has completed, stop the macro recorder by selecting Tools ⇨ Macro ⇨ Stop recording. Finally, select Tools ⇨ Macro ⇨ Visual Basic Editor and double-click the module1 node in the property page. The action should display the macro code used to enter data into the cells and to create and display a pivot chart. For Visual Basic, the code may be used as is. For C#, you will need to translate the code appropriately.

Summary

This chapter presented the `PivotTable` object. This object is responsible for generating `PivotTable` reports. As we have pointed out, `PivotTable` reports are interactive tables that summarize data. A pivot table report has distinct advantages over other reports. One of the chief advantages is that the report can be pivoted about the axes to gain a better perspective of the data. Pivot tables also contain a lot of built-in functionality such as charts, data connection wizards, and filtering and sorting functionality, as well as the ability to apply mathematical formula to each axis.

We focused on some key concepts that allow a pivot table report to stand out from the usual run-of-the-mill windows controls. In particular, the knowledge worker can simply drag and drop fields onto a `PivotTable` object to generate a customized report. From this point, the knowledge worker can filter, print, sort, and drill to uncover any anomalies in the data. And this is the purpose of a pivot table report application.

The importance of pivot filtering should not be overlooked. Imagine that a pivot table report contains a long list of field items. You may be able to appreciate how appealing a significantly reduced dataset is to the knowledge worker. The knowledge worker can simply use as many or as few fields as required to produce a report that is customized to his or her taste, not one that is imposed by the developer. To achieve this flexible list, the knowledge worker uses the Filter dialog. And that is the benefit of a pivot table list object.

Finally, we noted that the architects responsible for crafting the VSTO pivot table used the Excel `Range` object as a cornerstone of the pivot table structure. This decision results in a reduced learning curve. One other benefit is that the level of customization and formatting that may be applied to the pivot table data is greatly increased over other Office technologies, since Excel `Ranges` are very powerful and allow a great degree of flexibility.

All things considered, the pivot table report application is a formidable tool in the hands of capable knowledge workers who understand how to use it. However, this exuberance is not shared by .NET developers who seem to prefer the arcane constructs of flimsy grids and sub par controls over that of the polished, sophisticated pivot table. Hopefully, this chapter should serve as a powerful argument for adopting this lesser-known technology.

Index

SYMBOLS AND NUMERICS

#CIRC **error value, 70**
 #DIV/0! **error value, 70**
 ##### **error code, 70**
 #N/A **error value, 70**
 #NAME? **error value, 70**
 #NULL! **error value, 70**
 #NUM! **error value, 70**
 #REF! **error value, 70**
 #VALUE! **error value, 70**
3-D functional charts, 258–260, 270

A

Actions pane

Excel automation, 79–80
 Word development, 150–151

Activate **method, 159**
 ActiveExplorer **method, 161, 195**
 ActiveInspector **method, 161**
 ActiveMenuBar **object, 89, 133**

Add **method**

chart creation, 219–220
 discussed, 33
 document creation, 120
 Explorer object, 159
 pivot tables, 284

Add Web Reference dialog box, 89–90

AddComment **method, 39**
 AddDataField **method, 294**
 AddField **method, 294**

add-ins

class library project, 103
 Excel automation, 101–104
 managed code, 101
 Outlook automation, 174–175
 unmanaged code, 101

AddMyToolbar **method, 83**

address book manipulation, Outlook automation, 177–179

ADO.NET datasets, 26–30

advanced searches, Outlook automation, 187–190

AdvancedSearch **method, 187**

AllowDeletingColumns **parameter, 59**

AllowDeletingRows **parameter, 59**

allowdrop **property, 201**

AllowFiltering **parameter, 59**

AllowFormattingCells **parameter, 59**

AllowFormattingColumns **parameter, 59**

AllowFormattingRows **parameter, 59**

AllowInsertingColumns **parameter, 59**

AllowInsertingHyperlinks **parameter, 59**

AllowInsertingRows **parameter, 59**

AllowSorting **parameter, 59**

AllowUsingPivotTables **parameter, 59**

AND **construct, 186**

API (Application Programming Interface), 1, 211

Application **object**

defined, 30
 onshutdown event, 158
 onstart event, 158
 Outlook object hierarchy, 158

application objects

manipulation, 30
 Word automation, 108–109

Application Programming Interfaces (API)

Application Programming Interface (API), 1, 211

applications

- failed, 30
- hanging, 30
- loosely coupled, 64
- termination, 30

appointments (Outlook automation)

- creating, 165–167
- deleting, 170–172
- meetings, 172–173
- scheduling, 167–170
- subject properties, 172

architecture, VSTO, 2–4

AreaGroups object, 234

ArgumentException value, 84

assemblies, Word development considerations, 150

assemblyinfo file, 63–64

attachments, email, 163

author information, Web services, 91

AutoFill method, 52

AutoFormat dialog box (Excel), 42

AutoFormat method, 34

automation

- autoformatting, Excel range manipulation, 75–76
- objects, 10–11
- VSTO, 10–11

axes

- charts, 214, 226–228
- pivot tables, 279–280, 290–291

axisScale variable, 232

B

backgrounds, SetBackgroundPicture method, 42–43

backgroundWorker control, Word automation, 143, 145–146

BarGroups object, 234

BeforeClose method, 148

.bmp file extension, 270

bookmarks, Word automation, 112–114

buttons

- adding
 - to Excel menu, 88
 - to Excel toolbar, 82–84
 - to Outlook toolbars, 192–194
- click events
 - templates, 126
 - Windows applications, 203
 - Word toolbars, 127, 130

controls

- Excel Actions pane, 80
- Excel menu customization, 87
- newButton variable, 87
- removing
 - from Outlook menus, 197–198
 - from Outlook toolbars, 194–195

C

cache, pivot tables, 279

calcrange reference, 60–61

Calculate method, 68, 87–88

CallByName method, 133, 185

Caption property, 133

captions, charts, 234

case studies, Excel functionality, 46–50

categories property, 174

CDO (Collaboration Data Objects), 190

cells

- AllowFormattingCells parameter, 59
- charts, 213
- Excel ranges, 37–38
- merging, table manipulation, 119
- pivot table, 280, 292
- splitting, table manipulation, 119

Change event, 72

charts

- API (application programming interface), 211
- AreaGroups object, 234
- axes categories, 214, 226–228
- bar to cylinder conversion example, 265–266
- BarGroups object, 234
- captions, 234
- cells, 213
- Chart Wizard, 210
- ColumnGroups object, 234
- combination, 256–258
- data analysis
 - chart type selection considerations, 245
 - error bar analysis, 250–251
 - trend lines, basic functionality, 246
 - trend lines, custom generation, 248–250
 - trend lines, internally supported, 246–248
- data, inserting, 221–225
- data labels, 218
- data tables, 218
- datasets, 223–225
- design-time charting, 209
- DoughnutGroups object, 234

- embedded, 220–221
 - empty ranges, 222
 - error messages, 222
 - events, 251–254
 - font object customization, 237–238
 - formatting data, 236
 - GapDepth property, 235
 - GapWidth property, 235
 - gridlines, 215–216
 - height parameter position, 296
 - left parameter position, 296
 - legends, 216, 242–244
 - limitations, 271
 - line marker styles, 252–253
 - LineGroups object, 234
 - MajorUnit value, 228
 - MinorUnit value, 228
 - multiple instances, 254–256
 - NumberFormat object, 238–239
 - object hierarchy design, 226
 - objects, adding, 265–268
 - patterns, 242
 - perspectives, 258–260
 - pictures, adding, 267–268
 - pie, 215, 268–269
 - PieGroups object, 234
 - pivot table, 295–297
 - plotting area, 218
 - Points object, 260–264
 - polar, 268
 - radar, 215
 - rendering, 221
 - scaling, 228
 - series, 216, 226
 - seriescollection value, 235
 - special needs, 268–271
 - stand-alone, 219–220
 - subtypes, 211
 - surface area customization, 240–242
 - tables, 218
 - 3-D functional, 258–260, 270
 - tick marks, 228–232
 - time scaling, 215
 - title object, 213–214
 - titles, 232–234
 - top parameter position, 296
 - width parameter position, 296
 - chartspace **object, 213**
 - CheckSpelling **method, 122**
 - #CIRC **error value, 70**
 - class library project, add-ins, 103**
 - clearformat **method, 238**
 - click events**
 - Excel toolbar buttons, 83
 - smart tags, 142
 - Word menu automation, 133
 - client interface, VSTO, 3**
 - closing documents, Word automation, 121**
 - CLR (Common Language Runtime), 3, 5, 127**
 - code. See listings**
 - Collaboration Data Objects (CDO), 190**
 - color-coding, legends, 216**
 - ColumnGroups **object, charts, 234**
 - columns**
 - AllowDeletingColumns parameter, 59
 - AllowFormattingColumns parameter, 59
 - AllowInsertColumns parameter, 59
 - pivot tables, 293–294
 - range protection, 61–62
 - table manipulation, Word automation, 16
 - COM (Component Object Model)**
 - COM libraries, as VSTO Office system alternative, 8–9
 - discussed, 1
 - legacy frameworks, 6
 - combination charts, 256–258**
 - commandbarcontrol **event, 197**
 - Common Language Runtime (CLR), 3, 5, 127**
 - compatibility, VSTO system requirements, 6**
 - compilation errors, 7**
 - conditional formatting, 43**
 - connect.cs **file, 104**
 - Contents **parameter, 58**
 - Copy **method, 33**
 - CopyFace **method, 127**
 - copying worksheets, 33**
 - CreateFlag **method, 267**
 - CreatePivotTable **method, 284**
 - cross-referencing, pivot tables, 279–280**
 - csvdata.txt **file, 21**
 - CType **operator, 186**
 - currentchange **event, 78**
 - CurrentItem **method, 189**
- ## D
- data analysis**
 - charts
 - chart type selection considerations, 245
 - error bar analysis, 250–251
 - trend lines, 246–250
 - pivot tables, 274–275

data formatting

Excel functionality, 42–44
pivot tables, 289–290

data labels, charts, 218

data manipulation, Excel, 19–20

data tables, charts, 218

database table support, ODBC, 25

DataField **method**, 289

DataLabel **method**, 264

DataLabelRange **property**, 287

datasets

ADO.NET, 26–30

charts, 223–225

pivot tables, 281

Delete **method**, 32

deleting

Outlook appointments, 170–172

worksheets, 32

deployment, Word development considerations, 150

design-time charting, 209

design-time data loads, Excel, 20–21

design-time formatting, Excel, 42–44

design-time pivot tables, 274–275

design-time smart tagging, Word automation, 140–141

detectnewinstallation **property**, 155

development considerations, Word automation,
149–150

devtag **property**, 91

dialog boxes

Add Web Reference, 89–90

AutoFormat (Excel), 42

digital certificates, 64

DirectCast **method**, 186

#DIV/0! **error value**, 70

documents

creation, VSTO installation, 14

navigation, user interface functions, 3

processing, templates, 123–126

Word automation

closing, 121

creation, 120

discussed, 119

opening, 122

printing, 149

saving, 121

spell checking, 122

tables, adding, 115–116

wsdl, 89

DoughnutGroups **object**, 234

DoWork **event handler**, 143

drag and drop support

pivot tables, 277

Windows applications, 199–201

dragdrop **event**, 201

dragenter **event**, 201

DrawingObjects **parameter**, 58

drill-through functionality, pivot tables, 278, 283

drop zone axis, pivot tables, 280

E

EchoInput **property**, 103

email

attachments, 163

creating, 161–162

inbox folder manipulation, 164

MailItem object, 161–162

Send button, 162

embedded charts, 220–221

empty ranges, charts, 222

encrypted passwords, 57

enumerations

XlAxisGroup, 228

XlSheetVisibility, 33

error bar analysis, charts, 250–251

errors

#CIRC error value, 70

compilation, 7

#DIV/0! error value, 70

error code, 70

error handling, VBA, 8

error messages, chart data, 222

#N/A error value, 70

#NAME? error value, 70

#NULL! error value, 70

#NUM! error value, 70

#REF! error value, 70

#VALUE! error value, 70

values, Excel automation, 69–71

events

chart, 251–254

commandbarcontrol, 197

dragdrop, 201

dragenter, 201

MailLogonComplete, 181

NewExplorer, 206

newinspector, 170

NewMail, 183

onshutdown, 158

onstart, 158
 Outlook automation, 180–183
 pivot tables, 294–295
 removeMenuBar, 198
 Word automation, 147–148
 workbook
 Change event, 72
 currentchange, 78
 discussed, 71
 startup events, 72

Excel automation

Actions pane, 78–80
 add-ins, 101–104
 AutoFormat dialog box, 42
 automation considerations, 44–45
 COM libraries, as VSTO Office system alternative, 8–9
 data formatting and presentation, 42–44
 data manipulation, 19–20
 design-time data loads, 20–21
 design-time formatting, 42–44
 formulas
 design time, 66–67
 lowercasing, 104
 runtime, 68–69
 SUM, 68
 SUMIF, 68
 functionality
 case study example, 46–50
 spreadsheet functions, 51–52
 help documentation, 44
 menu bars, merging, 21
 menu customization
 adding buttons to menus, 88
 adding menus to toolbars, 86
 button creation, 87
 discussed, 85
 range manipulation
 autoformatting, 34, 75–76
 cell manipulation, 37–38
 error values, 69–71
 intersections, 39–40
 logical/physical relationship between cells, 34
 named identifiers, 35–36, 76
 offsets, 40–41
 range protection, 61–62
 unions, 38–39
 screen-updating feature, 23
 server-side automation, 95–99
 toolbars
 buttons, adding, 82–84
 click event handler, 83
 Formula Auditing, 67

style customization, 85
 workbooks
 data protection, 57–60
 events, responding to, 71–74
 list object control, 76–78
 opening, 31
 password protection, 55–57
 printing data in, 80–82
 range, autoformatting, 75–76
 Range control, 75–76
 worksheets
 adding new, 32–33
 copying, 33
 deleting, 32
 function implementation, 69
 hiding, 33, 60–61
 visibility, 33–34

Explorer **object**, 158–159

F

FaceId **property**, 83–84, 127, 133

failed applications, 30

fields, pivot tables, 280

file extensions

.bmp, 270
 .pst, 157

file loads

ADO.NET datasets, 26–30
 QueryTable object option, 25–26
 using .NET framework, 21–22
 using VTSO, 23–25
 XML, 26

FileProcessor **method**, 22

files

assemblyinfo, 63–64
 connect.cs, 104
 csvdata.txt, 21
 OfficeCodeBehind, 112, 147, 155
 VSknownIssues.rtm, 17
 XML
 file loads, 26
 schemas, 11
 server component architecture incorporation, 3
 smart tag example, 140

FileStream **object**, 23

filters

AllowFiltering parameter, 59
 Outlook search options, 185
 pivot tables, 282, 292–293

Find dialog, Outlook automation, 184

Find method

Find method, 185
FindControl method, 198
FindNext method, 186
flexibility, VSTO architecture, 4
folder manipulation, Outlook automation, 176–177
font objects
 charts, 237–238
 pivot tables, 285–286
for loop, 173, 184
FormatSheet method, 51
formatting
 chart data, 236
 formatting effects, Excel, 42–44
forms, Windows applications, 205–206
Formula Auditing toolbar (Excel), 67
formulas, Excel
 design time, 66–67
 lowercasing, 104
 runtime, 68–69
 SUM, 68
 SUMIF, 68
functions, worksheet, 69

G

GapDepth property, 235
GapWidth property, 235
General keyword, 44
GetImageResource method, 130
GetNamespace method, 181
GetType method, 169
global objects, 23
gridlines, charts, 215–216
GUID (Globally Unique Identifier), 103

H

hand-held devices, Outlook automation, 158
hanging applications, 30
HasLegend property, 244
headings, table manipulation, 119
height parameter position, charts, 296
help documentation, 44
hiding worksheets, 33, 60–61
HTML (Hypertext Markup Language), 25
hyperlinks, 59

I

icons, Word toolbar customization, 128
IDE (Integrated Development Environment), 5, 107
IIS (Internet Information Services), 65
ImageList control, 127–128
inbox folder manipulation, email, 164
Inspector object, 159–161
installation
 .NET framework, 6
 PIAs (Primary Interop Assemblies), 12
 VSTO (Visual Studio Tools for Office)
 installation problems, 16–17
 language selection, 12
 new document creation, 14
 project creation, 13–15
 project name, 14
 setup application, 11
 system requirements, 6
 template availability, 12
 VSkknownIssues.rtm file, 17

Integrated Development Environment (IDE), 5, 107
interface

 API (Application Programming Interface), 1, 211
 client interface, VSTO, 3
 MAPI (Messaging Application Programming Interface),
 154, 157–158
 UserInterfaceOnly parameter, 58
InternalStartup method, 148
Internet Information Services (IIS), 65
intersections, Excel range manipulation, 39–40
Invoke method, 147
InvokeMember method, 134, 169, 172
IsConflict property, 169
itemarray method, 29

J

JIT (Just-In-Time) compilation, 10

K

keywords
 General, 44
 ref, 7
 shortcutbar, 195
 vir, 187

L

- labels, pivot tables, 281, 286–287**
- language selection, VSTO installation, 12**
- left **parameter position, charts, 296**
- legacy frameworks, COM, 6**
- legacy libraries, 103**
- legends, charts, 216, 242–244**
- licensing restrictions, third party products, 9**
- line feed marker, Word automation, 110**
- line marker styles, charts, 252–253**
- LineGroups **object, 234**
- list object control, 76–78**
- listings**
 - charts
 - adding objects to, 265–267
 - axes, 227–228
 - combination, 257
 - data, inserting, 221, 223–224
 - default groups, 234–235
 - embedded, 220–221
 - error bar analysis, 250–251
 - events, 252–253
 - font objects, 237–238
 - legends, 243–244
 - line marker styles, 252–253
 - multiple instances, 255
 - NumberFormat **object, 238–239**
 - perspective, 259
 - pie, 269
 - Points **object, 261–263**
 - rendering, 221
 - scaling, 228
 - series manipulation, 226
 - special needs, 269
 - stand-alone, 219
 - surface area customization, 241–242
 - tick marks, 230–231
 - titles, 233–234
 - trend lines, 247, 249–250
 - digital certificates, 64
 - Excel functionality
 - Actions pane, 79–80
 - add-ins, 102–103
 - autoformatting, 34
 - case study example, 46–50
 - cell manipulation, 37–38
 - conditional formula usage, 68
 - error references, 70–71
 - intersections, 39–40
 - menu customization, 86–88
 - named identifiers, 35–36
 - offsets, 40
 - range protection, 61–62
 - runtime formula usage, 68
 - server-side automation, 95–99
 - spreadsheet functions, 51–52
 - toolbar customization, 82–85
 - unions, 38–39
 - file loads
 - ADO.NET datasets, 27–29
 - QueryTable **object, 25**
 - using .NET framework, 21–22
 - using VSTO, 23–24
 - XML, 26
 - formatting, 43–44
 - Outlook automation
 - address book manipulation, 178–179
 - advanced searches, 187–190
 - appointments, creating, 166–167
 - appointments, deleting, 170–172
 - appointments, scheduling, 168–169
 - email attachments, 163
 - email creation, 161–162
 - events, 180–183
 - Explorer **object, 159**
 - Find dialog, 184
 - Find method, 185
 - FindNext method, 186
 - folder manipulation, 177
 - Inspector **object, 160**
 - MAPI folder manipulation, 157
 - meetings, 172–173
 - menu customization, 196–198
 - note items, 173–176
 - OMG (Object Model Guard), 190–191
 - toolbar customization, 192–194
 - pivot tables
 - axes, 291
 - charts, 295–297
 - columns, 293
 - datasets, 281
 - events, 294–295
 - filters, 292–293
 - font objects, 285–286
 - labels, 286–287
 - NumberFormat **object, 289–290**
 - style customization, 287–288
 - strongly names assemblies, 63–64
 - Web services
 - query implementation, 92–93
 - search objects, 91

listings (continued)

- Windows applications
 - drag and drop support, 199–201
 - forms, 205–206
- Word automation
 - application objects, 108–109
 - bookmarks, 112–114
 - document manipulation, 120–122
 - documents, printing, 149
 - events, 147–148
 - long-running tasks, 143, 144.145
 - menu customization, 131–134
 - Office application executable example, 137–139
 - PowerPoint example, 134–137
 - range manipulation, 109–111
 - search functionality, 111
 - Selection object, 114–115
 - smart tags, 140–141
 - table styles, 117–119
 - tables, adding to documents, 115–116
 - template manipulation, 125–126
 - toolbar customization, 126–130
- workbooks
 - data in, printing, 81
 - data protection, 57
 - events, responding to, 72–74
 - functionality, 31
 - list object control, 76
 - password protection, 56–57
- worksheets
 - copying, 33
 - deleting, 32
 - function implementation, 69
 - hiding, 33, 60
 - visibility, 33–34
- listobject **value**, 94
- loading**
 - design-time data loads, 20–21
 - file loads
 - ADO.NET datasets, 26–30
 - QueryTable object option, 25–26
 - using .NET framework, 21–22
 - using VTSO, 23–25
 - XML, 26
- locked **property**, 250
- logical/physical relationship between cells, Excel range manipulation**, 34
- long-running task execution, Word automation**, 45, 143–144
- loosely coupled applications**, 64
- lowercasing, Excel formulas**, 104

M

macros

- functionality, 45
- malicious code, 8

MailItem **object**, 161–162

MailLogonComplete **event**, 181

MajorUnit **value**, 228

makecert.exe **utility**, 64

malicious code, macros, 8

managed code, add-ins, 101

MAPI (Messaging Application Programming Interface), 154, 157–158

maximum **property**, 250

MDX (Multidimensional Expressions), 278

meetings, Outlook appointments, 172–173

menu bars, merging, 21

menu customization

- Excel automation
 - adding buttons to menus, 88
 - adding menus to toolbars, 86
 - button creation, 87
 - discussed, 85
- Outlook automation
 - adding items to menus, 195–197
 - buttons, removing from menus, 197–198
- Word automation
 - built-in document statistics functionality, 131–132
 - click events, 133
 - late binding code sample, 133–134

MessageClass **argument**, 169

Messaging Application Programming Interface (MAPI), 154, 157–158

methods. See also objects

- Activate, 159
- ActiveExplorer, 161, 195
- ActiveInspector, 161
- Add
 - chart creation, 219–220
 - discussed, 33
 - document creation, 120
 - Explorer object, 159
 - pivot tables, 284
- AddComment, 39
- AddDataField, 294
- AddField, 294
- AddMyToolbar, 83
- AdvancedSearch, 187
- AutoFill, 52
- AutoFormat, 34
- BeforeClose, 148
- Calculate, 68, 87–88

CallByName, 133, 185
 CheckSpelling, 122
 clearformat, 238
 Copy, 33
 CopyFace, 127
 CreateFlag, 267
 CreatePivotTable, 284
 CurrentItem, 189
 DataField, 289
 DataLabel, 264
 Delete, 32
 DirectCast, 186
 FileProcessor, 22
 Find, 185
 FindControl, 198
 FindNext, 186
 FormatSheet, 51
 GetImageResource, 130
 GetNamespace, 181
 GetType, 169
 InternalStartup, 148
 Invoke, 147
 InvokeMember, 134, 169, 172
 itemarray, 29
 NumberFormatting, 37, 43–44
 Open, 23
 OpenText, 24
 OpenXML, 26
 PrintOut, 149
 RefreshTable, 287
 ReleaseComObject, 139, 170
 runWorkerAsync, 143
 SetBackgroundPicture, 42
 ShrinkToFit, 71
 Shutdown, 148
 Speak, 287
 Split, 119
 Startup, 92, 148
 ToString, 173
 TryCast, 162, 164, 186
 TryParse, 264
 writeXML, 29
Microsoft Excel. See Excel automation
Microsoft Office. See Office (Microsoft)
Microsoft Outlook. See Outlook automation
 minimum **property**, 250
 MinorUnit **value**, 228
Mono framework, 24
MOSTL (Microsoft Office Smart Tag List), 140, 142

Multidimensional Expressions (MDX), 278
multiple instances, chart data, 254–256
MYSQL database, 19

N

#N/A **error value**, 70
 #NAME? **error value**, 70
named identifiers, Excel range manipulation, 35–36, 76
namespace
 GetNamespace method, 181
 System.Globalization, 264
 System.IO, 21
 System.Reflection, 169
 System.Runtime.InteropServices, 169
.NET framework
 file loads, 21–22
 installation, 6
 security through, 62–66
 VSTO disadvantages, 9
 newButton **variable**, 87
 NewExplorer **event**, 206
 newinspector **event**, 170
 NewMail **event**, 183
 NGEN.exe **utility**, 10
 NOT **construct**, 186
note items, Outlook automation, 173–176
 #NULL! **error value**, 70
 #NUM! **error value**, 70
 NumberFormat **object**, 238–239, 289–290
 NumberFormatting **method**, 37, 43–44

O

object hierarchy design
 charts, 226
 Word automation, 107–108
Object Model Guard (OMG), 190–191
objects. See also methods
 ActiveMenuBar, 89, 133
 adding to charts, 265–268
 Application
 defined, 30
 onshutdown event, 158
 onstart event, 158
 Outlook object hierarchy, 158
 application object manipulation, 30
 AreaGroups, 234

objects (continued)

- automation, 10–11
- BarGroups, 234
- CDO (Collaboration Data Objects), 190
- chartspace, 213
- ColumnGroups, charts, 234
- COM (Component Object Model)
 - COM libraries, as VSTO Office system alternative, 8–9
 - discussed, 1
 - legacy frameworks, 6
- DoughnutGroups, 234
- Explorer, 158–159
- FileStream, 23
- global, 23
- Inspector, 159–161
- LineGroups, 234
- MailItem, 161–162
- NumberFormat, 238–239, 289–290
- OMG (Object Model Guard), 190–191
- PieGroups, charts, 234
- PivotCache, 279
- pivotdata, 279
- Points, 260–264
- QueryTable
 - file loads, 25–26
 - Web services, 89
- Selection, 114–115
- StringBuilder, 94
- ThisApplication, 108–109
- title, charts, 213–214
- WebRequest, 94

ODBC (Open Database Connectivity)
database table support, 25
discussed, 20

Office (Microsoft)
Professional version, VSTO compatibility, 6
server-side processing, 3
XML schemas, 11

Office web components (OWC), 8
OfficeCodeBehind file, **112, 147, 155**

offsets, Excel range manipulation, 40–41

ol variable, Outlook automation, 160

OMG (Object Model Guard), 190–191

Online Analytical Processing (OLAP), 273, 279

onshutdown event, 158

onstart event, 158

OOM (Outlook Object Model), 154, 158

Open Database Connectivity (ODBC)
database table support, 25
discussed, 20

Open method, 23–24

opening
documents, Word automation, 122
workbooks, 31

OpenText method, 24

OpenXML method, 26

OR construct, 186

Oracle database, 19

Outlook automation
add-ins, 174–175
address book manipulation, 177–179
Application object, 158
appointments

- creating, 165–167
- deleting, 170–172
- meetings, 172–173
- scheduling, 167–170
- subject properties, 172

- discussed, 154
- email
- attachments, 163
- creating, 161–162
- inbox folder manipulation, 164
- Send button, 162
- events, 180–183
- Explorer object, 158–159
- folder manipulation, 176–177
- hand-held devices, 158
- Inspector object, 159–161
- MAPI (Messaging Application Programming Interface), 154, 157–158
- menu customization
- adding items to menus, 195–197
- buttons, removing from menus, 197–198
- note items, 173–176
- ol variable, 160
- OMG (Object Model Guard), 190–191
- .pst file extension, 157
- search options
- advanced searches, 187–190
- filters, 185
- Find dialog, 184
- Find method, 185
- FindNext method, 186
- search dialog, 183
- SMTP (Simple Mail Transfer Protocol), 157
- toolbar buttons
- adding, 192–194
- removing, 194–195

Windows application integration
 button clicks, 203
 discussed, 198
 drag and drop support, 199–201
 forms, 205–206

Outlook Object Model (OOM), 154, 158

out-sourcing obstacles, 9

OWC (Office web components), 8

P

PageWidth **property, 133**

paragraph reference, Word automation, 110–111

parameters

AllowDeletingColumns, 59
 AllowDeletingRows, 59
 AllowFiltering, 59
 AllowFormattingCells, 59
 AllowFormattingColumns, 59
 AllowFormattingRows, 59
 AllowInsertingColumns, 59
 AllowInsertingHyperlinks, 59
 AllowInsertingRows, 59
 AllowSorting, 59
 AllowUsingPivotTables, 59
 Contents, 58
 DrawingObjects, 58
 height parameter position, charts, 296
 left parameter position, charts, 296
 Scenarios, 58
 top parameter position, charts, 296
 UserInterfaceOnly, 58
 width parameter position, charts, 296

passwords

encrypted, 57
 Password parameter, 58
 password-cracking applications, 57
 workbook, 55–57

patterns, charts, 242

performance, VSTO disadvantages, 10

perspective, charts, 258–260

PIAs (Primary Interop Assemblies)

installation problems, 16
 installing, 12
 reasons for, 5
 VSTO system requirements, 7
 Word development considerations, 149–150

pictures, adding to charts, 267–268

pie charts, 215, 268–269

PieGroups **object, 234**

pivot tables

Add method, 284
 AllowUsingPivotTables parameter, 59
 axes, 279–280, 290–291
 cache, 279
 cells, 280, 292
 charts, 295–297
 columns, 293–294
 CreatePivotTable method, 284
 cross-referencing, 279–280
 data analysis, 274–275
 data formatting, 289–290
 data in, printing, 277
 DataField method, 289
 datasets, 281
 design-time, 274–275
 drag and drop support, 277
 drill-through functionality, 278, 283
 drop zone axis, 280
 events, 294–295
 fields, 280
 filters, 282, 292–293
 font objects, 285–286
 labels, 281, 286–287
 limitations, 296–297
 MDX (Multidimensional Expressions), 278
 PivotCache object, 279
 pivotdata object, 279
 queries, 275
 RefreshTable method, 287
 rows, 277
 slicing and dicing concept, 280
 style customization, 287–288
 uses for, 273

PivotCache **object, 279**

pivotdata **object, 279**

plotting area, charts, 218

Points **object, chart data, 260–264**

polar charts, 268

PowerPoint automation listing example, 134–137

Primary Interop Assemblies. See PIAs

printing

pivot table data, 277
 Word documents, 149
 workbook data, 80–82

PrintOut **method, 149**

procedural programming language, VBA as, 8

Professional version (Microsoft Office), 6

properties

- allowdrop, 201
- Caption, 133
- categories, 174
- DataLabelRange, 287
- detectnewinstallation, 155
- devtag, 91
- EchoInput, 103
- FaceId, 83–84, 127, 133
- GapDepth, 235
- GapWidth, 235
- HasLegend, 244
- IsConflict, 169
- locked, 250
- maximum, 250
- minimum, 250
- PageWidth, 133
- retVal, 187
- Shapes, 267
- Tag, 133
- userRange, 60

properties pages, Web services, 89

Q

queries

- pivot tables, 275
- Web services, 92–93

QueryTable **object**

- file loads, 25–26
- Web services, 89

R

radar charts, 215

random key pairs, strongly named security, 63

range manipulation

- Excel automation
 - autoforamtting, 34, 75–76
 - cell manipulation, 37–38
 - error values, 69–71
 - intersections, 39–40
 - logical/physical relationship between cells, 34
 - named identifiers, 35–36, 76
 - offsets, 40–41
 - range protection, 61–62
 - unions, 38–39
- Word automation, 109–111

RCW (Runtime Callable Wrapper)

- discussed, 5, 101
- Word development considerations, 149–150

#REF! error value, 70

ref keyword, 7

RefreshTable method, 287

regular expressions, smart tags, 142

relative positioning, Excel cell manipulation, 38–39

ReleaseComObject method, 139, 170

removeMenuBar event, 198

rendering charts, 221

retVal property, 187

rows

- AllowDeletingRows parameter, 59
- AllowFormattingRows parameter, 59
- AllowInsertingRows parameter, 59
- pivot tables, 277
- table manipulation, Word automation, 116

runtime

- smart tagging, 141
- using Excel formulas at, 68–69

Runtime Callable Wrapper (RCW)

- discussed, 5, 101
- Word development considerations, 149–150

runtime exception avoidance, table manipulation, 119

runWorkerAsync method, 143

S

saving documents, Word automation, 121

scaling, charts, 228

Scenarios parameter, 58

scheduling appointments, 167–170

schemas, XML, 111

screen-updating feature, Excel, 23

search functionality

- Outlook automation
 - advanced searches, 187–190
 - filters, 185
 - Find dialog, 184
 - Find method, 185
 - FindNext method, 186
 - search dialog, 183
- Web services, 91
- Word automation, 111

security

- digital certificates, 64
- strongly named assemblies, 63–64
- through .NET framework, 62–66

- VSTO approaches to, 2
 - VSTO disadvantages, 9–10
 - workbooks
 - data protection, 57–60
 - password protection, 55–57
 - Selection **object**, **114–115**
 - Send button, email, 162**
 - serialization, Excel range manipulation, 34**
 - series, charts, 216, 226**
 - seriescollection **value, charts, 235**
 - server component, VSTO, 3–4**
 - ServerDocument **class, 100**
 - server-side automation, Excel automation, 95–99**
 - server-side processing, Microsoft Office, 3**
 - SetBackgroundPicture **method, 42–43**
 - setup application, VSTO installation, 11**
 - Shapes **property, 267**
 - shortcutbar **keyword, 195**
 - ShrinkToFit **method, 71**
 - Shutdown **method, 148**
 - signing digital certificates, 64**
 - Simple Mail Transfer Protocol (SMTP), 157**
 - slicing and dicing concept, pivot tables, 280**
 - smart tags, Word automation**
 - click events, 142
 - design-time, 140–141
 - labeling text with, 139
 - MOSTL (Microsoft Office Smart Tag List), 140, 142
 - regular expressions, 142
 - runtime implementation, 141
 - sample XML file, 140
 - SMTP (Simple Mail Transfer Protocol), 157**
 - sorting, AllowSorting parameter, 59**
 - SPCs (software publisher certificates), 64**
 - Speak **method, 287**
 - special needs charts, 268–271**
 - spell checking documents, Word automation, 122**
 - Split **method, 119**
 - spreadsheets. See Excel automation**
 - SQL database, 19**
 - stand-alone charts, 219–220**
 - stand-alone version, VSTO, 4**
 - startup events, 72**
 - Startup **method, 92, 148**
 - StringBuilder **object, 94**
 - strongly named assemblies, 63–64**
 - style customization**
 - Excel toolbars, 85
 - pivot tables, 287–288
 - Word table automation, 117–119
 - subject properties, Outlook appointments, 172**
 - subtypes, charts, 211**
 - SUM **Excel formula, 68**
 - SUMIF **Excel formula, 68**
 - surface area customization, charts, 240–242**
 - system clipboard, Word toolbar customization, 127–128**
 - system requirements, VSTO, 6–7**
 - System.Globalization **namespace, 264**
 - System.IO **namespace, 21**
 - System.Reflection **namespace, 169**
 - System.Runtime.InteropServices **namespace, 169**
- ## T
- tables**
 - charts, 218
 - table manipulation, Word automation
 - adding tables to documents, 115–116
 - cell merging, 119
 - cell splitting, 119
 - headings, 119
 - row implementation, 116
 - runtime exception avoidance, 119
 - table styles, 117–119
 - Tag **property, 133**
 - templates**
 - availability, VSTO installation, 12
 - Word automation
 - button click events, 126
 - document processing, 123–126
 - termination, applications, 30**
 - testing digital certificates, 64**
 - third party products, as VSTO Office system alternative, 9**
 - ThisApplication **object, 108–109**
 - 3-D functional charts, 258–260, 270**
 - tick marks, charts, 228–232**
 - time scaling, charts, 215**
 - time-saving techniques, design-time charting, 209**
 - title **object, charts, 213–214**
 - toolbars**
 - Excel
 - buttons, adding, 82
 - click event handler, 83
 - Formula Auditing, 67
 - style customization, 85
 - Outlook
 - buttons, adding, 192–194
 - buttons, removing, 194–195

toolbars (continued)

Word

- button events, 127, 130
- discussed, 126
- icons, 128
- ImageList control, 127–128
- system clipboard, 127–128

top **parameter position, charts, 296**

ToString **method, 173**

treeview **control, 201**

trend lines, chart data

- basic functionality, 246
- custom generations, 248–250
- internally supported, 246–248

TryCast **method, 162, 164, 186**

TryParse **method, 264**

U

UDF (user defined functions), 101

unions, Excel range manipulation, 38–39

unmanaged code, add-ins, 101

URIs (Universal Resource Identifiers), 92

user interface, VSTO, 3

UserInterfaceOnly **parameter, 58**

userRange **property, 60**

utilities, NGEN.exe, 10

V

validation, user interface functions, 3

#VALUE! **error value, 70**

VBA (Visual Basic for Applications)

- advantages, 8
- discussed, 2
- error handling, 8
- macros, 8
- as procedural programming language, 8
- unmanaged code, 8
- as VSTO Office system alternatives, 7–8

vir **keyword, 187**

VSkknownIssues.rtm **file, 17**

VSTO (Visual Studio Tools for Office)

- alternatives to, 7–9
- architecture, 2–4
- automation, 10–11
- client interface, 3
- disadvantages, 9–10

- functionality, 4
- installation
 - language selection, 12
 - new document creation, 14
 - PIA installation, 12
 - problems with, 16–17
 - project creation, 13–15
 - project name, 14
 - setup application, 11
 - template availability, 12
 - VSkknownIssues.rtm file, 17
- Microsoft Office Professional Edition compatibility, 6
- new features, 1–2
- security approaches, 2
- server component, 3–4
- stand-alone version, 4
- system requirements, 6–7
- user interface, 3
- Visual Studio add-in, 5

W

watchdog application, Excel server-side automation, 97

Web services

- author information, 91
- listobject value, 94
- property pages, 89
- queries, 92–93
- QueryTable object, 89
- search objects, 91
- URIs (Universal Resource Identifiers), 92

WebRequest **object, 94**

While **statement, 97**

width **parameter position, charts, 296**

Windows application integration, with Outlook

- button clicks, 203
- discussed, 198
- drag and drop support, 199–201
- forms, 205–206

Word automation

- application objects, 108–109
- backgroundWorker control, 143, 145–146
- bookmarks, 112–114
- column manipulation, 116
- development considerations, 149–150
- documents
 - closing, 121
 - creating, 120
 - discussed, 119

- opening, 122
- printing, 149
- saving, 121
- spell checking, 122
- events, 147–148
- line feed marker, 110
- long-running task execution, 45, 143–144
- menu customization
 - built-in document statistics functionality, 131–132
 - click events, 133
 - late binding code sample, 133–134
- object hierarchy design, 107–108
- Office application executable example, 137–139
- paragraph reference, 110–111
- PowerPoint automation example, 134–137
- range manipulation, 109–111
- search functionality, 111
- Selection object, 114–115
- smart tags
 - click events, 142
 - design-time, 140–141
 - labeling text with, 139
 - MOSTL (Microsoft Office Smart Tag List), 140, 142
 - regular expressions, 142
 - runtime implementation, 141
 - sample XML file, 140
- table manipulation
 - adding tables to documents, 115–116
 - cell merging, 119
 - cell splitting, 119
 - headings, 119
 - row implementation, 116
 - runtime exception avoidance, 119
 - table styles, 117–119
- templates
 - button click events, 126
 - document processing, 123–126
- ThisApplication object, 108–109
- toolbar customization
 - button events, 127, 130
 - discussed, 126
 - icons, 128
 - ImageList control, 127–128
 - system clipboard, 127–128

workbooks

- data protection, 57–60
- events
 - Change, 72
 - currentchange, 78
 - discussed, 71
 - startup, 72
- list object control, 76–78
- opening, 31
- password protection, 55–57
- printing data in, 80–82
- range, autoformatting, 75–76
- Range control, 75–76

worksheets

- adding new, 32–33
- copying, 33
- deleting, 32
- function implementation, 69
- hiding, 33, 60–61
- visibility, 33–34
- writeXML **method**, 29
- wsdl **document**, 89

X**X axes, charts, 214**

- XlAxisGroup **enumeration**, 228

- XlSheetVisibility **enumeration**, 33

XML files

- file loads, 26
- schemas, 11
- server component architecture incorporation, 3
- smart tag example, 140

Y**Y axes, charts, 214**

powered by

books24x7[®]

Programmer to Programmer™



Take your library wherever you go.

Now you can access more than 70 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the **Wrox Reference Library**. For answers when and where you need them, go to wrox.books24x7.com and subscribe today!

Find books on

- ASP.NET
- C#/C++
- Database
- General
- Java
- Mac
- Microsoft Office
- .NET
- Open Source
- PHP/MySQL
- SQL Server
- Visual Basic
- Web
- XML



www.wrox.com