



Expert

Access® 2007 Programming

Rob Cooper, Michael Tucker



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

Expert Access™ 2007 Programming

Rob Cooper and Michael Tucker



Wiley Publishing, Inc.

Expert Access™ 2007 Programming

Introduction	xxv
Part I: Programming Access Applications	1
Chapter 1: Overview of Programming for Access	3
Chapter 2: Extending Applications Using the Windows API	11
Chapter 3: Programming Class Modules	63
Chapter 4: Debugging, Error Handling, and Coding Practices	105
Part II: Data Manipulation	145
Chapter 5: Parsing Data	147
Chapter 6: Using SQL to Retrieve Data and Manipulate Objects	193
Chapter 7: Managing Data	263
Part III: Interacting with the Application	297
Chapter 8: Using Code to Add Advanced Functionality to Forms	299
Chapter 9: Using Code to Add Advanced Functionality to Reports	357
Chapter 10: Using Automation to Add Functionality	387
Chapter 11: Creating Dynamic Ribbon Customizations	441
Part IV: Finalizing the Application	489
Chapter 12: Configuration and Extensibility	491
Chapter 13: Personalization and Security	525
Chapter 14: Deployment	559
Chapter 15: Help and Documentation	613
Appendix A: Programming Tips and Tricks	637
Appendix B: Query Performance	657
Appendix C: Pattern Reference	665
Index	673

**Expert
Access™ 2007 Programming**

Expert Access™ 2007 Programming

Rob Cooper and Michael Tucker



Wiley Publishing, Inc.

Expert Access™ 2007 Programming

Published by

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

Copyright © 2008 by Wiley Publishing, Inc., Indianapolis, Indiana

ISBN: 978-0-470-17402-9

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data is available from the publisher.

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, Wrox 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. Microsoft and Access are trademarks or registered trademarks 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.

To my family, Sandi, Isabel, Gillian, and Taryn for their support and patience over the years.

— Rob Cooper

To my family, and to everyone who shares my love for data.

— Michael Tucker

About the Authors

Rob Cooper is a Test Lead on the Access team at Microsoft. He started at Microsoft as a support engineer in Charlotte, North Carolina in 1998 and joined the Access 2003 test team in Redmond in 2001. During the Access 2007 release, he led the security efforts across the test team and worked on several new features including disabled mode, database encryption, Office Trust Center, and sorting, grouping, and totals. Rob also led efforts around the Access object model and continues to provide direction around programmability and security in Access.

A long-time fan of Access, Rob is a frequent speaker at the Seattle Access Group and PNWADG meetings and has written for the Microsoft Knowledge Base and Access Advisor. Aside from writing code in Access and C#, he also enjoys spending time with his family watching movies, going to the zoo and aquarium, and hanging out in and around Seattle.

Michael Tucker is a Software Design Engineer on the Microsoft Access test team in Redmond. He joined Microsoft in 1993 as a Product Support Engineer supporting the very first release of Microsoft Access, and has been passionate about Access ever since. During the Access 2007 release, he worked on the new ACE database engine, and owned the complex data and SharePoint workflow integration features.

Michael designed and coordinated implementation of the test automation infrastructure used by the test team. He also has previous experience as a Test Lead and Program Manager on a variety of products at Microsoft. His specialties include SQL, data normalization, and object model design, and anything related to his passions for airplanes, airports, and traveling the world.

Credits

Executive Editor

Robert Elliott

Development Editor

Ed Connor

Technical Editors

Oliver Stohr and Matt Bumgarner

Production Editor

Daniel Scribner

Copy Editor

Nancy Rapoport

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

Project Coordinator, Cover

Lynsey Osborn

Compositor

Craig Woods, Happenstance Type-O-Rama

Proofreader

Sossity Smith

Indexer

Ron Strauss

Anniversary Logo Design

Richard Pacifico

Acknowledgments

The authors would like to thank the following individuals who really made this possible. First of all to Teresa Hennig for helping us get set up on this project. By giving us the opportunity to work on the last book, you helped lead to this one. To our incredible tech editors, Oli Stohr and Matt Bumgarner. Your insight and suggestions really make this book what it is and we are grateful for your hard work and keen eyes! To Kevin Nickel on the Access test team who gave us some great suggestions and considerations as we were beginning to flush out the content.

We'd also like to thank everyone at Wiley who helped bring this together, in particular Bob Elliott for getting us started and guiding us through the process. Special appreciation goes out to our development editor Ed Connor who helped us make sure that the content and tone was accurate and for keeping us on track.

Lastly, but certainly not least, to the Access team at Microsoft. The incredible amount of passion, hard work, and dedication that went into the Access 2007 release from everyone involved is nothing short of amazing.

Rob Cooper

I'd like to thank my children, Isabel and Gillian, who were even more patient and more understanding than they were for the last book—you continue to inspire me! Thanks to my wife, Sandi, for her support during all of the late nights and weekends. I'm free to do yard work again!

Thanks to my co-author Michael for keeping me in check and helping me make sure that we covered things that we felt should be covered. Thanks in particular for your great suggestions and for setting the tone for the SQL and Data chapters!

Extra thanks to Bob Elliott at Wiley for his insight and wisdom as I was taking on this role. Your patience, understanding and willingness to help were greatly appreciated.

Huge thanks to everyone on the Access product team at Microsoft — without you and your hard work on a great release of Access 2007, this wouldn't have been possible! To Sherri Duran for her continued support while I worked this project.

Lastly but certainly not least, thanks to the Access community, particularly the amazing people at utteraccess.com. It's truly an honor to be a member of a community of people who are as passionate about Access as we are, and to the Seattle Access Group and Pacific Northwest Access Developer's Group for allowing me to come listen and speak from time to time.

Michael Tucker

To my family, whom I don't see often enough, but who are never out of my thoughts: Mom and Dad, Mike and Margie, Shari, Kimberlee, Heidi, Danna, Holly, Kathleen, Rayne and Ann. Monica and Jason and their little Archer, Scott, Kittridge, and Markus: thank you for your friendship, constant kind words of encouragement, thought-provoking conversations, and nudges.

My co-author and colleague Rob Cooper has been an inspiration ever since I returned home to the Access team. His knowledge of Access is encyclopedic, and his enthusiasm and passion are infectious. I am honored and privileged to have been invited to join him on this project.

To all of the amazing professionals on the Microsoft Access product team who daily put so much energy into making an amazing product even more so, thank you for your inspiration. The most satisfying thing about working at Microsoft is being surrounded by so many amazing minds from around the world, who bring a myriad of talents and varied perspectives. To Mike Garlick and Sherri Duran who welcomed me home, I will always be in your debt. To Anderson Dupree, who was my first Best Manager; Steve Alboucq who opened doors to the product team; and Mark Langley who mentored me into management, my gratitude for your support and guidance.

Finally, my appreciation to all the Access users I've had the privilege to work with over the last 14 years. You never cease to amaze me with the unusual, often creative, and occasionally inspirational ways you use our product. I remember fondly a small and select group of particularly memorable customers who made my time in product support especially interesting: my psychic friends in Florida, a certain banker in San Diego who taught me the value of scaled integers, the OSHA man in Portland who shares my passion for aviation, and the lovely lady from AT&T in Chicago who proved once and for all that everything is relative.

Contents

Acknowledgments	x
Introduction	xxv
Part I: Programming Access Applications	1
Chapter 1: Overview of Programming for Access	3
Writing Code for Access	3
The Access Object Model	4
The DAO Object Model	4
The ADO Object Model	4
Object-Oriented Thinking	4
Windows API Programming	5
Working with Managed Code	5
What Is Managed Code?	6
Versions of Visual Studio	6
Writing Managed Code Libraries to Use with Access	6
Referencing the Access Object Model from Managed Code	7
Referencing Other Applications	7
Discovering References	8
Adding References to Office Applications	8
Summary	9
Chapter 2: Extending Applications Using the Windows API	11
Windows API Overview	11
Why Use API Functions	12
API Resources	12
Writing Declare Statements	13
Example Conversion	17
Error Messages from API Functions	18
Retrieving System Information	19
Return the User Name in Different Formats	19
Return the Computer Name in Different Formats	21
Retrieve the Command Line	21
Windows Shell Functions	24
Get the Path to a Special Folder	24
Show the Property Dialog Box for a File	25
Determine Whether a Computer Is Connected to the Internet	27
Determine Whether a User Is an Administrator	27

Contents

Controlling Processes and Threads	28
Determining the Current Process ID	28
Pausing Code Execution	29
Quitting a Process	29
Waiting for a Process to Complete	32
Windows Vista Task Dialogs	35
Introduction to Task Dialogs	35
A Simple Message Box Replacement	37
More Complex Task Dialogs	40
Icons	56
Miscellaneous API Functions	57
Creating a GUID	58
ShellExecute Tricks	59
Summary	60
Chapter 3: Programming Class Modules	63
Overview of Class Modules	63
Why Use Class Modules?	64
Class Module Members	65
Code Reuse in VBA	73
Subclassing	76
Subclassing Access Forms and Reports	76
Sinking Form Events	77
Interfaces	80
Inheritance in VBA	81
Specialization	83
Testing the Type of an Object	83
Collections	85
Writing the Standard Collection Class	86
An Alternate Implementation for the Add Method	89
Setting a Default Member	91
For Each Enumeration	91
The ICollectionEx Interface	92
Events	96
Why Write Custom Events?	96
Creating Custom Events	97
Listening for Custom Events	98
Design Considerations	99
Copying a Class Instance	100
Exposing the Interface of a Derived Type	101
Raising Errors	102
Summary	104

Chapter 4: Debugging, Error Handling, and Coding Practices	105
 Debugging	106
Using Watches	106
Using the Watches Window	108
Using the Immediate Window	109
Using the Locals Window	110
Viewing the Call Stack	110
Building a Call Stack Using Code	111
Assertions	115
Creating a Debug Build	120
 Error Handling	123
Handling Errors Inline	123
Categorizing Errors	124
System Error Codes and HRESULTS	124
Creating an Error Handler Class	125
Logging Errors	129
Using the Error Handling Class and Logging	135
 Coding Practices	136
Readability	136
Version Control	139
Refactoring	139
Code Reviews	142
 Summary	142
Part II: Data Manipulation	145
Chapter 5: Parsing Data	147
 File Input/Output	147
Using VBA I/O statements	148
Using FileSystemObject	157
Determining Whether a File Is ANSI, Unicode, or UTF-8	160
 Splitting Strings	164
The VBA String Handling Functions	165
Replacing Tokens in Strings	169
Formatted Token Replacement	170
Parsing into Data Types	171
An End-to-End Example	174
Splitting Names	178
Address Element Granularity	180
Phone Number Granularity	182

Contents

Creating HTML	182
Exporting HTML Files	182
Why Create HTML Using Code?	187
Creating HTML Files	188
Summary	192
Chapter 6: Using SQL to Retrieve Data and Manipulate Objects	193
Where SQL Can Be Used in Access	194
Form and Report Recordsets	194
Partial SQL Properties	194
The Various Types of SQL Queries	195
The SELECT Query	196
Project Your Columns: the Field List	197
Choose Your Tables: The FROM Clause	203
Filter Your Data: The WHERE Clause	205
Sort Your Results: The ORDER BY Clause	216
Add Data from More Than One Table Using Table Joins	218
Inner Joins	219
Outer Joins	220
Joining More Than Two Tables Using Nested Joins	222
Self Joins	224
Cartesian Products	225
Prompt Users for Data with Parameters	227
Add Data from Other Databases Using the IN Clause	230
Selection Predicates	231
ALL Predicate	231
TOP Predicate	231
DISTINCT Predicate	233
Aggregating Data	234
Bucket Your Data: The GROUP BY Clause	234
Filter Your Data Based on Bucketed Data: the HAVING Clause	237
Action Queries	238
Make Table Query	239
Append Query	240
UPDATE Query	243
DELETE Query	245
Crosstab Queries	246
Ordering Column Headings	248
Common Expressions Used with Crosstab Queries	249
UNION Queries	251
Sorting a UNION Query	252
Using the Designer to Create Union Queries	253

Subqueries	253
Data Definition Queries	254
Create Table	255
Alter Table	257
Drop Table	258
Create Index	258
Drop Index	259
Alter Database Password	259
SQL Pass-Through Queries	259
Running Ad Hoc SQL Pass-Through Queries	260
ANSI Mode	261
String Pattern Matching Using ANSI-92 Syntax	261
Summary	261
 Chapter 7: Managing Data	 263
Finding Data	263
Find Methods	264
Seek Method	266
Move Methods	268
Search Optimization Tips	271
Categorization and Sorting	272
Categorization Using the Choose Function	273
Using a Custom Sort Order Field	273
Backup and Restore	276
Using Checksum Functions	278
MD5 Algorithm	279
Mod-10 Checksum	279
Access 2007 Specifics	285
Locale-Agnostic Parser for ColumnHistory Function	285
Getting a List of Attachments in an Attachment Field	289
Searching for Records with Attachments	292
Searching for Specific Attachments	294
Summary	296
 Part III: Interacting with the Application	 297
 Chapter 8: Using Code to Add Advanced Functionality to Forms	 299
Working with Form Events	299
How to Determine When Data Is Added	300
How to Determine When Data Is Changed	301

Contents

How to Determine When Data Is Deleted	301
Validating Form Data	302
Suppressing Access Error Messages	304
How to Determine If a Modifier Key Is Pressed	305
Periodic Backups Using the Timer Event	306
Moving a Form Without a Border	307
Customizing ControlTipText	308
Working with Controls	309
Validating Control Data	309
Disabling all Controls	310
Reusing a Subform Control	310
Extended List Box Functionality	311
Custom Progress Bars	319
Custom Up/Down Buttons	321
Displaying Multiple Attachments Onscreen	322
Common Forms for Your Applications	324
Dynamic Menu Forms and Dashboards	324
Splash Screens	349
About Dialog Boxes	351
Kiosk Forms	351
Custom Form Navigation	352
Navigation Bars	352
“I Need To” Drop-Down	354
Keyboard-Driven Navigation	355
Summary	356
Chapter 9: Using Code to Add Advanced Functionality to Reports	357
Interactive Reports Using Access 2007	357
Sorting a Report Using Controls on a Report	358
Filtering a Report Using Controls	360
Drill-Down	363
Navigating to a Map Dynamically	365
Report Scenarios	366
Creating a Report with Data Bars	366
Creating a Calendar Report	370
Displaying Images Using an Attachment Field	377
Displaying Images Dynamically Using a Path	378
Creating a Report Manager	379
Creating the Reports Table	380
Implementing Report Manager Features	381
Summary	386

Chapter 10: Using Automation to Add Functionality	387
Automation Basics	387
When to Automate	388
Shell Function	388
CreateObject vs. GetObject	389
Early Binding vs. Late Binding	389
Discovering Object Models Using Built-in Tools	390
Automating Windows	391
The Shell Object	392
Windows Scripting Host	395
Automating Office Applications	409
Determining If an Office Application Is Installed	409
Word: Creating a Formatted Letter with Data	412
Excel: Creating an Excel Chart with Data	421
Outlook: Create Appointments from an Events Database	429
Automating Internet Explorer	432
Opening a URL in a New Tab in IE7	432
Common Web Queries	433
Parsing HTML from Internet Explorer	435
Summary	440
Chapter 11: Creating Dynamic Ribbon Customizations	441
Overview of Ribbon Customizations in Access	441
Development Tips	442
How to Write Ribbon Customizations	444
Loading Ribbons	446
Using the USysRibbons Table	447
Using the LoadCustomUI Method	448
Programming the Ribbon	449
Ribbon Objects	449
Using Callback Routines	449
Refreshing Controls	452
Organizing Ribbon Items	453
Tabs	453
Contextual Tabs	453
Groups	454
Ribbon Controls	454
Buttons	455
Toggle Buttons	455
Check Boxes	456

Contents

Combo Boxes and Drop-Downs	457
Labels and Edit Boxes	458
Menus	458
New Types of Controls	460
Dialog Box Launcher	460
Gallery	461
Split Button	462
Dynamic Menu	462
Images	462
Images Included with Office	462
Loading Images from External Files	463
Loading Images from an Attachment Field	468
Moving Functionality into the Ribbon	469
The NotInList Event — Ribbon Style	470
Form Navigation	472
Managing Filters Using a Dynamic Menu	477
Creating a Split Button That Sticks	482
Other Ribbon Modifications	484
Modifying the Office Menu	484
Overriding Commands and Repurposing Controls	485
Summary	486
Part IV: Finalizing the application	489
Chapter 12: Configuration and Extensibility	491
Localization	491
Locale Settings	492
Determining the Current Language of Office	494
How to Create Localized Access Applications	494
Application Options	506
Storing Options in Tables	508
Storing Options in the Registry	508
Creating a Class to Work with Options	509
Creating Options Forms	513
Displaying Options in the Ribbon	516
Creating Form Themes	518
Theming Forms	518
Summary	523

Chapter 13: Personalization and Security	525
What Is Security?	525
Authentication	526
Authorization	527
What Is Personalization?	527
Overview of Security in Access 2007	527
Database Encryption	527
Disabled Mode	528
Office Trust Center	529
Digital Signatures	531
Signed Packages	531
Access 2007 Navigation Pane	533
Customizing the Navigation Pane	533
Restricting the View of the Navigation Pane	536
Hiding the Design Surface of Objects	539
Password-Protecting Objects	539
Creating the User Table	540
Creating the Login Form	540
Prompting for Password with Forms and Reports	544
Creating a Password Protected Form Object	545
Windows Integration	546
Determining the Logon Name	547
Locking the Computer	548
Receiving Notifications When the Computer Is Locked or Unlocked	548
Simulating Record Level Security	551
Limiting the View to a Particular User	551
Locking Records Based on Logon	552
Database Encryption with DAO and ADO	553
Creating an Encrypted Database with DAO	553
Changing the Database Password with DAO	553
Adding a Database Password to a Database with DAO	554
Changing the Database Password with ADO	555
Best Practices	556
Using a Client-Server Database to Protect Data	556
AllowBypassKey Property	556
Using an ACCDE or MDE File	557
Using an ACCDR File	557
File Format Selection	557
Summary	558

Contents

Chapter 14: Deployment	559
Creating an Automated Build	559
Creating the Visual Studio Project	560
Building the Application	571
Handling Application Dependencies	578
Installing Files from an Attachment Field	579
Updating References	581
Testing Reference Fix-Up	583
Late Binding	584
Licensing Your Applications	585
Creating a Limited-Use Application	585
Registering an Application	590
Miscellaneous Deployment Scenarios	596
Updating Your Applications	596
Programmatically Creating DSNs	607
Ensuring an Application Runs Locally	610
Summary	612
Chapter 15: Help and Documentation	613
Documentation via Built-in Properties	613
Using the Description Property	614
End SubSetting the Status Bar Text	619
Using the Tag Property	620
Setting the Control Tip	620
Database Documenter	621
Providing Help to Users	621
Storing Help Text in a Table	621
Creating a Help Form	622
Trapping the F1 Key	623
Alternatives to a Floating Help Window	624
External Approaches	625
Writing HTML Content for Help	625
Mapping HTML Files to Objects	627
Opening the Browser	628
Creating Compiled HTMLHelp Files	631
Summary	635

Appendix A: Programming Tips and Tricks	637
Dynamically Running Code	637
CallByName	637
Application.Run	641
Eval	641
Object Helper Functions	641
Object Initialization	642
Object Termination	643
Global Object Properties	644
Date Data Type Tricks	644
Looping Through Dates	645
Arrays of Dates	646
Determining Dates Dynamically	646
Miscellaneous Tips	647
Categorizing Constant Values	647
Comparing Class Instances	648
Add Number of Retries to a Code Block	649
Add a Timeout to a Code Block	650
Tools for the Immediate Window	651
Clearing the Immediate Window	652
Displaying the Watch Window	652
Displaying the Locals Window	652
Making a Backup of Objects	653
Closing All Code Windows	654
Appendix B: Query Performance	657
The Query Optimizer	657
Designing Tables for Performance	658
Use the Smallest Appropriate Data Type	658
Indexes	658
Designing Queries for Performance	659
Reduce the Amount of Data	659
Other Performance Considerations	662
Bulk Edits	662
Minimize Network Traffic	662
Database Maintenance	663
Hard Drive Maintenance	663
Microsoft Access Performance Analyzer	663

Contents

Appendix C: Pattern Reference	665
Introducing Patterns	665
What Is a Pattern	665
Why Use Patterns?	666
Patterns Used in This Book	666
Gang of Four Patterns	666
VBA Object Patterns	667
Other Patterns	669
Working with Strings	670
Where and How to Find Patterns	671
Index	673

Introduction

The largest release of Access in five years, Access 2007 includes many new features that you can leverage both in your own development work and as application features that you can pass along to your users. This book discusses new features such as developing for the Ribbon and the Attachment control, as well as new ways of looking at old problems such as automation, debugging, and deployment. It aims to fill the gaps between casual Access development and professional Access development. We wrote this book as a tribute to the product that we've come to use every day. It represents a labor of love for the years that we have spent working with Access both at Microsoft and prior to joining Microsoft.

Building Off-the-Shelf Applications

Throughout this book, we discuss several topics that contribute to off-the-shelf quality. Off-the-shelf is a term used to describe software that is typically available for release to the public. This doesn't necessarily mean that the software is available commercially (although it could). Even departmental or larger scale internal applications can benefit from features such as configuration (discussed in Chapter 12), and deployment (discussed in Chapter 14). We live in a global marketplace. If you are developing databases for use by people in locales other than your own, you might be particularly interested in Chapter 12, where we will also discuss localization.

Forms are essential to most Access applications. Forms are used for everything from displaying and validating data, to splash screens, to About dialog boxes. You will learn about creating different types of forms such as these in Chapter 8. It doesn't stop there, however. You will also learn about how to create custom navigation solutions for forms to provide intuitive experiences for your users. In many solutions, seeing an error message from Access is less than desirable so you will also learn how to suppress Access error messages to provide your own. We also discuss using Form events to validate data; detect when records are added, updated, or deleted; and much, much more.

But let's not forget about reports! Reports are also an integral feature of Access solutions. They provide another mechanism for presenting information to the user and are optimized for printing. Access 2007 includes a new view for reports called Report view, which allows users to interact with reports as never before. You will learn how to use Report view to create compelling scenarios for your users such as sorting and filtering. You no longer need to use a separate form to sort or filter reports. Features such as this are discussed in Chapter 9.

A Little Background

We both started our careers at Microsoft in what was then known as Product Support Services (PSS) as members of the Access support team. Learning Access from the support perspective was extremely valuable. For starters, calls didn't come in because Access was working just fine! People tend to call when something is broken or they have questions about how to implement a piece of functionality. This breadth of knowledge led to developing common methods for working with a particular task. Upon making the transition to testing Access, we learned development practices that were in place on a large software project. It is largely these practices that we implement in our Access development projects.

Whom This Book Is For

This book is primarily geared toward intermediate to advanced Access programmers using VBA and SQL. Some of the concepts apply to VBA as a language and not to one application in particular. As a result, VBA programmers using other tools such as Excel or Word may also find some of the topics useful. Familiarity with Access 2007 is helpful but not required. New features mentioned in Access 2007 are called out as such and explained throughout. You'll find that several of the concepts apply to previous versions of Access as well.

This book contains some examples written in managed code using Visual C#. A basic familiarity with this language will be helpful for those examples. The code available for download with this book includes managed code samples in both Visual C# and Visual Basic .NET.

Some experience with XML will be useful for Chapter 11.

What This Book Covers

Expert Access 2007 Programming aims to show you how to create off-the-shelf quality applications using Access that are fun to write, feature-rich, and easy to use. Several of the techniques covered in this book are used to develop Access itself at Microsoft such as using assertions, build numbers, and classes. Some techniques such as handling exceptions are also closely tied to the work we do with managed code. We have adapted these techniques to apply to the process of writing Access-based applications, and these are techniques we implement in our own solutions. Employing these techniques, even in Access-based solutions, can simplify the development process and provide consistency across multiple applications.

How This Book Is Structured

This book is structured in four parts with either three or four chapters in each part. This book is meant to walk you through the important steps in creating a full-featured application. As such, it is designed to be read sequentially from beginning to end.

Part I: Programming Access Applications

The chapters in Part I are designed to provide the foundations of programming Access applications. Chapters 2 through 4 apply to VBA developers, regardless of the application.

- ❑ Chapter 1, “Overview of Programming for Access,” describes in broad strokes what it means to write applications for Access. Often, this means writing VBA code in a particular database, but there may also be times when you are required to write applications that target Access, for example, using managed code. This chapter discusses some of the basics of working with managed code, in particular C# and VB.NET.
- ❑ Chapter 2, “Extending Applications Using the Windows API,” provides information about using API functions that are defined in Windows, and then provides many useful examples of API functions that you can use in your Access applications.

- ❑ Chapter 3, “Programming Class Modules,” gives you in-depth information about writing class modules for use in your applications. The chapter begins with an overview of writing class modules in Access, and then jumps right in to using them to extend, and even simplify, programming tasks. The chapter ends with defining events that you can use as notification when particular conditions are met.
- ❑ Chapter 4, “Debugging, Error Handling, and Coding Practices,” provides a detailed look at debugging VBA applications and extending debugging to make it work for you.

Part II: Data Manipulation

With a new version of the Access database engine (formerly known as Jet) and built-in support for other database engines, it’s an understatement to say that Access applications are well suited for data. In Part II, we move into the nitty-gritty of Access development — data. The chapters in Part II describe working with data in many different forms.

- ❑ Chapter 5, “Parsing Text and HTML,” starts out with an in-depth look at working with data in different types of files using VBA I/O statements and the `FileSystemObject`. Along the way, it shows you how to determine the byte order mark of a file to determine whether a text file is ANSI or Unicode. The chapter then discusses working with strings and how to parse data into various data types. It ends with a look at generating formatted HTML.
- ❑ Chapter 6, “Using SQL to Retrieve Data and Manipulate Objects,” shows you how to use SQL to do just about anything. It begins with a discussion about places in Access where SQL can be used and the different types of queries that can be created. It then goes into the details of the `SELECT` statement. It also shows you how to work with other query types such as crosstab and action queries, and ends with data-definition and SQL pass-through queries.
- ❑ Chapter 7, “Managing Data,” shows you how to find data in a database, as well as how to categorize data and create a custom sort field. Creating a backup of data in the database is an important task and the chapter describes a mechanism for archiving and backing up records in the database. This chapter ends with examples that target new features in Access 2007: append-only memo fields and attachment fields.

Part III: Interacting with the Application

Once data is stored in the database, your users will need a way to view the data and interact with it. This is where Part III comes in. The chapters in Part III are designed to make interfaces and functionality that pop.

- ❑ Chapter 8, “Using Code to Add Advanced Functionality to Forms,” shows you how to create reusable forms for your applications such as dynamic menu screens and dashboards, as well as splash screens and About dialog boxes. As events are critical of any form-driven application in Access, there is an in-depth look at working with form and control events.
- ❑ Chapter 9, “Using Code to Add Advanced Functionality to Reports,” begins with interactive scenarios that use an exciting new feature of Access 2007 — Report view. It then provides some other examples such as creating a calendar report and data bars à la Excel 2007, and ends with a full-featured report manager.
- ❑ Chapter 10, “Using Automation to Add Functionality,” provides samples that can be used to extend an Access application outside the boundaries of Access. It starts out with the basics of

Introduction

automation such as the `GetObject` and `CreateObject` functions, and then gives you some scenarios for automating Windows — yes, Windows. Because Access is a part of the overall Office family of applications, the chapter provides code for interacting with Word, Excel, Outlook, and Project. Web applications and social applications such as instant messaging are becoming more and more popular. To address this trend, the chapter ends with examples for automating Internet Explorer and Windows Live Messenger, formerly MSN Messenger.

- ❑ Chapter 11, “Creating Dynamic Ribbon Customizations,” goes into the fine points of Ribbon development. It begins with some development tips that we found useful and then jumps right in to the details of writing callback routines and XML for the Ribbon. The chapter also shows you how to use images in your customizations and then contains some examples of moving typical Access functionality such as the `NotInList` event of a combo box into the Ribbon. The chapter is aimed primarily at Access developers, but because the Ribbon can be used by other applications in Office, VBA developers in Word, Excel, and PowerPoint may also find this useful.

Part IV: Finalizing the Application

Okay, so you’ve developed the core functionality in your application and tested it. It’s time to start wrapping it up. Part IV covers some of the essentials for finishing an Access-based application.

- ❑ Chapter 12, “Configuration and Extensibility,” shows you how to create localized applications and detect locale information on the computer. It then creates a framework for extending an application using options and, last, shows you how to theme your Access applications.
- ❑ Chapter 13, “Personalization and Security,” describes the new security features in Access 2007. It discusses how you can create personalized solutions for your users using the navigation pane, and how to add Windows integration to an application. Chapter 13 ends with some best practices related to security and personalization.
- ❑ Chapter 14, “Deployment,” shows you how to create builds for your applications and includes a managed code tool to accomplish this. This chapter also discusses including dependencies for your applications and how to update them. It ends with a discussion about licensing and registering an Access application.
- ❑ Chapter 15, “Help and Documentation,” shows you how to add help to your applications. It describes some built-in approaches such as status bar text and tooltips, and then discusses context-sensitive help. Web-based help is also popular for many applications and the chapter shows you how to create that. The chapter ends with the details of creating a compiled HTML help file.

What You Need to Use This Book

You must have a computer that meets the minimum system requirements for Microsoft Office 2007. The minimum requirements for Office 2007, according to Microsoft, are a 500 MHz computer or higher, 256MB of RAM, and 2GB of hard-drive space.

To create the managed code examples, you will need a version of Visual Studio that targets the Microsoft .NET Framework 2.0. This includes Visual Studio 2005 Standard Edition or higher or either the Visual Basic .NET 2005 Express Edition or Visual C# 2005 Express Edition. Creating the resource DLL described in Chapter 12 requires Visual Studio 2005 Standard Edition or higher, or the Visual C++ 2005 Express Edition.

The minimum requirements for Visual Studio 2005 Standard Edition are a 600 MHz computer or higher, 192MB of RAM, and 2GB of hard-drive space.

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.

Source

This section includes the source code.

Source code

Source code

Source code

Output

This section lists the output:

Example output

Example output

Example output

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 978-0-470-17402-9.

Once you download the code, just decompress it with your favorite compression 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, like 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.

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 e-mail 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 e-mailed 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.

Part I: Programming Access Applications

Chapter 1: Overview of Programming for Access

Chapter 2: Extending Applications Using the Windows API

Chapter 3: Programming Class Modules

Chapter 4: Debugging Error Handling and Coding Practices

1

Overview of Programming for Access

In this chapter, you take a look at the different mechanisms available for programming for Access and what it means to program Access applications. You will see that with the ability to use the Access object model outside of Access, programming *for* Access is not necessarily the same as programming *in* Access. And, while the possibility is there to develop external applications that consume the Access object model, the primary focus of the book is scenarios that use the Access object model and Visual Basic for Applications (VBA) from within Access itself.

In this chapter, you:

- Learn about using managed code and how it can be used to work with Access applications
- Review object models that are often used with Access
- Learn about off-the-shelf applications in the context of Access development and what it means to develop off-the-shelf applications

Writing Code for Access

As a part of the overall Microsoft Office family of products, Access finds itself in an interesting position. It provides many tools that Access developers and programmers at all levels use to create robust database applications for themselves or their users. Access 2007 includes a database engine for storage, forms and reports for presenting data, macro objects for automating simple tasks, and a full-featured programming model using VBA. Collectively, these components make it possible for you, as the programmer or developer, to create rich solutions that can be easily deployed to your users.

As fun as it is to write Access applications, Access doesn't provide everything. For example, it doesn't provide an easy way to create e-mail messages with an arbitrary attachment such as Outlook. It doesn't provide statistical functions, such as Excel, and it doesn't provide word

Part I: Programming Access Applications

processing functionality, such as Word. It can, however, interact with these applications, as a part of the overall Office family, using each application's respective object model to participate in the larger solution. Chapter 10 provides insight into how Access fits into the larger picture when creating a solution.

In many cases, writing code for Access is different from writing code for other Office applications, such as Word and Excel. Many Access solutions are designed specifically to work with multiple Office applications. In our day-to-day work, we use applications that we have written to enable us to send custom e-mail messages using Outlook or to create a Word document with very specific formatting. In addition, when you develop a database in Access, often you are developing a full-featured application for use by multiple users.

The Access Object Model

The code you write for Access forms, reports, and controls targets the Access object model, but to retrieve data you need to use a data access technology such as Data Access Objects (DAO), or ActiveX Data Objects (ADO). The requirement for using two object models creates an interesting dichotomy when you're writing applications for Access, which also sets Access apart from writing code for other Office applications.

The result of this separation makes it possible to write code for Access applications that can easily be reused between applications. You might be thinking, "Can't I write modular code for Excel-based applications as well?" Of course. But we think you're more likely to do so when writing an Access-based application. From the pure coding perspective, we are not suggesting that you should always separate presentation code from data access code (in different files), but rather that it's possible. However, if you think about it, we do frequently separate the presentation layer from the data access layer when we create an application with linked tables.

The DAO Object Model

Data Access Objects, or DAO, has long been used as the native data access technology for Access. Originally included in the Jet database engine with previous versions of Access, new features appear in DAO for use with Access 2007. You learn more about these new features, and other features of DAO in Chapter 7.

The ADO Object Model

ActiveX Data Objects, or ADO, is another data access technology available to use with Access. Both technologies are acceptable and can be used in conjunction with one another, although it's probably not necessary to do so. DAO, as the native application programming interface (API) for the Access database engine, has performance benefits over ADO. That said, ADO is more generic and thus has its own benefits. For example, ADO provides the ability to create Recordset objects that are not bound to a table or query, but rather are created at runtime by appending fields. In addition, you can use an ADO recordset as the data source for a form, combo box, or list box.

Object-Oriented Thinking

Object-oriented programming (OOP) organizes programming tasks into *classes*. A class is a blueprint of the thing being modeled. Often, this is something in the real world such as a customer, or an employee. Other

times, it may be something more abstract such as a log file, or a dialog box. Classes are said to be self-describing because they describe the characteristics of an entity, and its behaviors — that is, what it can do. The characteristics of a class are known as properties, and the behaviors of a class are known as methods.

A class is different from an *object*. An object is a unique instance of a class. Over the years, language changes to Visual Basic, and subsequently VBA, have provided some of the OOP features long used by C++ programmers to VB and VBA developers.

Making the decision to use classes often represents a different way of thinking from traditional procedural programming. Classes play an important role in some of the concepts discussed throughout the book. As a result, we discuss programming classes and class design in greater detail in Chapter 3. Classes form the basis of several other pieces of functionality that follow in this book.

Windows API Programming

For those tasks that VBA or the Access object model does not provide, you can use the Windows API. An API, or application programming interface, is a set of functions grouped together by technology. The Windows API contains the core set of functions for Windows itself, but there are other APIs as well. For example, the DAO API consists of the objects, properties, and methods that make up DAO. The API is discussed in depth in Chapter 2.

Working with Managed Code

At the beginning of this chapter, we mentioned *managed code* as an alternative for working with the Access object model. Managed code refers to code whose memory is managed by the Common Language Runtime (CLR) of the .NET Framework. This often refers to code written in Visual C#, or Visual Basic .NET, although this is not a requirement. For example, Visual C++ developers can use C++/CLI available with Visual Studio 2005 to write managed code.

The focus of this book is writing solutions that utilize Access for its strengths and for building high-quality applications that are based on Access. Sometimes, this means writing code using other object models such as Outlook, or Excel, and integrating them as part of an overall solution. Other times, it means writing code in a different language altogether. Several chapters in this book provide examples using C# to either drive the Access object model, or as a library that you call from VBA code inside Access. The managed code samples that are available for download with this book include both C# and VB.NET.

A great amount of material has been written about C# ranging from syntax to constructs to design. Therefore, we won't spend much time explaining the language. We don't expect you to be a C# whiz, but if you are that's great! If not, no worries. We explain the code as we go so the managed code solutions provided are straightforward and understandable. Although the managed code in this book is written using C#, we have provided the VB.NET equivalents on the corresponding Web site for this book.

Because we use managed code in this book, in addition to VBA, let's spend some time discussing managed code in a little more detail.

Part I: Programming Access Applications

What Is Managed Code?

As mentioned, managed code refers to any code that is written where memory is managed by the CLR. For the purposes of this book, we write managed code in C# — although you could just as well write in VB.NET. We chose C# because we use it in our daily work in testing Access and find it to be a nice language for many programming tasks.

The portion of the CLR that manages memory is known as the *garbage collector*. The garbage collector is responsible for detecting when objects are no longer needed, and for disposing of them appropriately. If you have written code in other languages such as C or C++, you may quickly recognize this as a powerful feature, although many C and C++ developers may prefer to manage memory themselves. If you fall into this camp, the garbage collector in the CLR provides features for tighter control when objects are cleaned up. Although VBA does not have built-in garbage collection, you are, in effect, managing your own memory in VBA when you set an object to `Nothing` such as:

```
Set objMyObject = Nothing
```

The opposite of managed code of course is *unmanaged code*. Unmanaged code is code whose memory is not managed by a runtime such as the .NET Runtime. This includes code written in C, C++, Visual Basic, or even VBA.

For more information about managed code, please refer to Appendix C of the Access 2007 VBA Programmer's Reference, ISBN 978-047004703.

Versions of Visual Studio

Managed code and the .NET Runtime were first available with Visual Studio 2002, which included version 1.0 of the .NET Framework. Version 1.1 of the .NET Framework was released with Visual Studio 1.1. The most recent release of Visual Studio, Visual Studio 2005 includes version 2.0 of the .NET Framework. For users running Windows XP or Windows 2003 Server, you can download either version of the .NET Framework from the Microsoft Web site. If you are running Windows Vista, version 3.0 of the .NET Framework is now included with the operating system and offers new libraries that further enhance managed code development.

You can even use one of the Visual Studio Express Editions to write managed code using either VB.NET or C#. However, be cautious — the Express Editions of the languages do not include the tools required to create a type library for use with Access and so you have to use a command-line tool that is included with Express Edition. We talk more about this tool in the next section.

Writing Managed Code Libraries to Use with Access

In addition to features such as garbage collection, the .NET Framework provides many libraries that you can use in managed code. Collectively, these libraries are known as the Base Class Library (BCL), and include functionality that is typically found in the Windows API. Because these libraries are available to you when writing managed code, they are nice to use in a type library that you call from VBA code in Access.

A library you create in a managed language such as C# creates a dynamic link library (DLL) file. And, while you can create references to DLL files from Access and VBA, you cannot reference a managed DLL directly. In order to set a reference to a DLL from VBA, it must expose the necessary COM interfaces that

Chapter 1: Overview of Programming for Access

are used to provide type information about a class. Managed DLL files do not include these interfaces. Therefore, in order to create a reference to a managed code library, you must first create a type library for the DLL. You can use Visual Studio to create the type library or a command-line tool called `tlbexp.exe`. This tool is available as part of the Visual Studio SDK, which is freely available on the Microsoft Web site.

So, you're probably thinking, why would we include an additional reference in the application when we can just use the Windows API? Good question. Managed code makes it pretty straight forward to write complex libraries that would have required a fair amount of API code to implement in VBA. Examples of this include:

- Using Windows common dialog boxes
- Working with the Windows Registry
- Writing to the Windows event log
- Retrieving the version number of a file

Writing managed code libraries to use in an Access solution has its benefits — namely, it's easier to write and deploy. However, it is not without its drawbacks. Installing a library written in any language requires an additional file as a part of the installation. However, we feel that the benefits of using managed code (when necessary) as part of a solution outweighs the drawbacks.

Referencing the Access Object Model from Managed Code

In addition to creating managed code libraries that you can call from VBA, you can write managed code that drives the Access object model itself. For example, you might write a report manager solution that bundles the reports in an Access application together to print multiple reports based on a timer. Such an application could use the Access object model to get a list of reports in the database and to print them.

You can add two types of references to a managed application — .NET and COM. Because Access is a COM-based application, the reference to the Access object model is a COM reference. If you add a reference to the Microsoft Access 12.0 object library in Visual Studio, you'll notice that a few additional references are given to you. These additional references are `ADODB`, `DAO`, `Microsoft.Office.Core`, and `VBIDE`. These references are all used somewhere in the Access object model and as a result are included automatically when you add the Access object model from your managed code.

In Chapter 14, you see how to create a managed application using C# to create a build of an Access application including features such as version and build numbers and release dates.

Referencing Other Applications

In order to use object models provided by applications such as Outlook or Project, you typically add a reference. However, adding references creates dependencies in an application that may not be desired. For instance, what happens if you add a reference to the Outlook 12.0 object library but your users are using Outlook 2002 (10.0)?

Issues such as these can be avoided using a technique known as *late-binding*. Late-binding enables you to write code without providing type information about an object. And while it also enables you to trap for

Part I: Programming Access Applications

error conditions at runtime instead of compile time, it tends to lead to code that is slower to execute. This lag is negligible on today's fast processors with a decent amount of memory. The opposite of late-binding is *early-binding*. Early-binding provides benefits such as compile-time checking and performance improvements. This sounds pretty nice, but it causes problems if your users do not have the same applications installed or the same version. For these times, late-binding becomes very useful.

Discovering References

The easiest way to find a reference to use in your Access application is to view the References dialog box from the Visual Basic Editor, as shown in Figure 1-1.

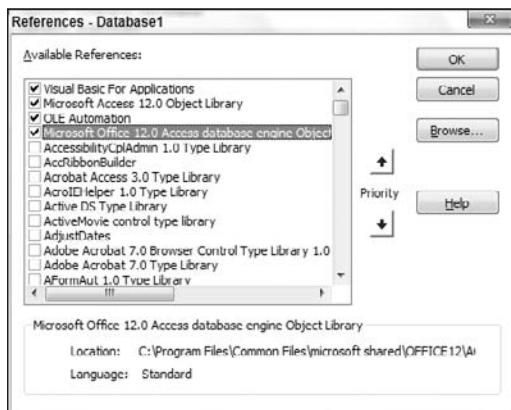


Figure 1-1

You can add references to COM objects that reside in DLL, EXE, or OCX files, or even to other databases that you create in Access. By adding references to other databases you can create reusable libraries of your own written in VBA. We discuss using references to databases in more detail in Chapter 3, and the issue of updating references in Chapter 14.

If you've opened the References dialog box, shown in Figure 1-1, and scrolled through the list, you'll quickly notice that there are a lot of files listed. Because there are probably more files listed on your development machine than those of your users, can you imagine the headache of trying to manage multiple references in an application? With all of these references to choose from, how do you know what you can or should use?

Adding References to Office Applications

The applications in the Office family of products are designed to work well together. As such, it's very common to find Access applications that include references to Office, Outlook, Excel, and the like. You can add a reference to other Office applications in the References dialog box. Adding a reference uses early-binding to an external application.

Chapter 1: Overview of Programming for Access

To use late-binding to another application, you must know its programmatic identifier, or *ProgID*. The ProgID is a string that identifies an application for use by another application. The following table provides the ProgID values for Office applications:

Application	ProgID
Access	Access.Application
Excel	Excel.Application
InfoPath	InfoPath.Application
OneNote	OneNote.Application
Outlook	Outlook.Application
PowerPoint	PowerPoint.Application
Project	Project.Application
Publisher	Publisher.Application
Visio	Visio.Application
Word	Word.Application

Summary

In this chapter, we provided some thoughts about what it means to write an Access-based solution. We examined how writing applications for Access is quite different from writing VBA code in other Office applications. Because Access is used to develop applications, sometimes off-the-shelf, it is often viewed as a development tool when compared to other applications in Office such as Word and Excel.

To support this view of Access as a development tool, we discussed the following:

- Object models commonly used while writing Access applications
- Programming tools and techniques such as the Windows API and object-oriented programming
- Using managed code that targets the Access object model, as well as using managed code as a library inside of an Access application

You also saw a glimpse into what lies ahead in the rest of this book. Features that are often found in commercial applications such as configuration, deployment, and help also have their place in Access applications. More important, you can standardize them across many applications. Over the course of the book we introduce code that you can use to integrate features such as these into your own applications.

Coming up next, you learn how to use the Windows API and why it can be useful in your applications.

2

Extending Applications Using the Windows API

VBA simplifies certain tasks, such as reading data from files, asking the user for information, or even launching an external application. As a programming language, it also does a good job at preventing you from shooting yourself in the foot. As a result, a fair amount of functionality to developers of other languages, such as C++ or C#, is not available natively. Therefore, you utilize the Windows API to achieve some of this same functionality.

In this chapter, you learn how you can leverage the Windows API in your applications. More specifically, you learn the following:

- How to translate an API function declaration written in C/C++ to VBA
- How to use the API to retrieve information about the system
- How to control processes and threads, and work with the Windows shell
- How to use new task dialogs in Windows Vista that allow you to create exciting user interfaces for your applications running on Vista

Windows API Overview

The Windows API contains literally thousands of functions that can be used to perform a wide variety of tasks ranging from retrieving information about the system, to printing, to creating processes, to well, you name it. Virtually everything that Windows does as an operating system can be accomplished using the API.

Part I: Programming Access Applications

Why Use API Functions

VBA is a great language, and along with C#, it is certainly our favorite. It contains many powerful features that we as developers use to provide very rich applications to our users. That said, it doesn't do everything, and that's okay. You can use the API to take it that extra mile and to obtain the most amount of control over the experience provided by your application. For some tasks, such as showing a Windows common dialog box, using the API can even prevent you from having to deploy extra components with your application such as an ActiveX control.

We also mentioned obtaining control, and the Windows common dialog box scenario is another good example of this. The Access object model includes a property called `FileDialog`, which wraps the Windows common dialog box. This property alone allows you to keep from using the `comdlg32.ocx` ActiveX control, but it still doesn't provide everything that the API itself can do. For example, using the API you can include a Help button in the common dialog box and call a function in your VBA code when it is clicked. Neither the ActiveX control nor the `FileDialog` object does this.

Finally, there are just certain things that VBA cannot do. For example, it cannot retrieve certain information about the system, or create a Globally Unique Identifier (GUID). It cannot create a new task dialog provided by Windows Vista, and for that matter, cannot determine which operating system is running. For all of these things you'll need the API.

API Resources

Knowing how to translate the MSDN documentation to VBA is important but in many cases you actually won't have to. There are several resources available online where these functions have already been translated to VB for you. The ones we use most often are as follows:

- ❑ <http://vbnet.mvps.org>
- ❑ www.pinvoke.net

The term pinvoke refers to P/Invoke, or platform invoke which is the mechanism used in managed code to refer to API functions. For more information about P/Invoke, refer to the C# or VB.NET documentation on MSDN.

Both sites include documentation for the constants, types, and declarations used for many scenarios where you want to use the API. Here are some other tools that you can use when working with the Windows API:

- ❑ **Platform SDK** — An SDK, or software development kit, includes tools and documentation for working with a particular set of API functions. The Platform SDK is the core documentation for the Windows API and can be downloaded freely from the Microsoft Web site. This would take the place of the MSDN library online and would run locally on your computer.
- ❑ **dumpbin.exe** — A command-line tool available with the Platform SDK and Visual C++. You can use the `dumpbin` command with the `/exports` command-line switch to get a list of functions that are exported from a library. These functions can potentially be used in a `Declare` statement.
- ❑ **depends.exe** — A graphical tool available with the Platform SDK and Visual C++. Displays library dependencies for a module, but also includes the list of exported functions for a library.

Writing Declare Statements

In order to use the API, you'll have to write a `Declare` statement. The `Declare` statement in VBA contains the following parts:

```
[Public | Private] Declare [Function | Sub] "RoutineName" Lib "LibraryName.dll"  
[Alias "AliasName"] ( Arguments ) [As Type]
```

Let's break this down a little:

1. First, the access modifier: In a standard module, you can specify a `Declare` as being either `Public` or `Private`. In a class module, you have to define a `Declare` statement as `Private`.
2. Next comes the `Declare` keyword, followed by either `Function` or `Sub` depending on the return type of the API you are calling. After that, assign the `Declare` a name using the same rules for naming other types of routines.
3. Specify the location for the library that contains the API function using the `Lib` keyword. The library can be a DLL, OCX, or EXE file.
4. You can optionally define the `Alias` for the API. If you specified your own name as the `RoutineName`, you must use the `Alias` keyword. If you use the name of the API itself as the `RoutineName`, you can skip this keyword in many cases.
5. Define the arguments that the API accepts, followed by the return type if any.

There are two considerations to keep in mind when using the `Alias` keyword. First, keep in mind that the names of API functions are case-sensitive. This is because most API functions are written in C, which is a case-sensitive language (VBA of course is not case-sensitive). Second, when the API you are calling contains a C-based string argument, you typically need to use the `Alias` keyword.

Why? Consider the following declaration for the `ShellExecute` API from the `ShellAPI.h` header file. These declarations have been reformatted for readability.

```
SHSTDAPI_(HINSTANCE) ShellExecuteA(HWND hwnd, LPCSTR lpOperation, LPCSTR lpFile,  
LPCSTR lpParameters, LPCSTR lpDirectory, INT nShowCmd);  
SHSTDAPI_(HINSTANCE) ShellExecuteW(HWND hwnd, LPCWSTR lpOperation, LPCWSTR lpFile,  
LPCWSTR lpParameters, LPCWSTR lpDirectory, INT nShowCmd);  
#ifdef UNICODE  
#define ShellExecute ShellExecuteW  
#else  
#define ShellExecute ShellExecuteA  
#endif // !UNICODE
```

The header file actually defines two API functions: `ShellExecuteA` and `ShellExecuteW`. The difference between the two is the use of *wide* characters for parameters. A wide character is a character that utilizes two bytes of space instead of one. Following the two declarations, the header file then defines `ShellExecute` as calling either `ShellExecuteW` or `ShellExecuteA`, depending on whether Unicode support is available. As a result, there is no function called `ShellExecute` defined in the API! Therefore, if you try to call `ShellExecute` from a `Declare` statement in VBA, you receive an error that says the function cannot be found. Instead, you need to use the `Alias` keyword to indicate whether you want `ShellExecuteA` or `ShellExecuteW`.

Part I: Programming Access Applications

How to Find API Functions

The documentation for the Windows API is largely found on the MSDN Web site: <http://msdn.microsoft.com>. If you have installed Visual Studio or the Platform SDK you can also use the Windows header files (.h) to find information about an API. The API documentation on MSDN refers to the header file that declares the function.

A header file in C and C++ is used to declare functions, structs, constants, and classes. Sometimes the header file is the best place to see how an API is declared.

C to VBA Data Type Conversions

In order to successfully convert an API declaration written in C to a `Declare` statement in VBA, you'll need to be able to recognize the data types used by the APIs and translate them to their VBA equivalents. VBA has a limited set of data types. The C language also has a limited set of data types, but variations on a given data type in effect make it a new data type. Some examples of this in C are when you use a *pointer* to a data type, or a constant in the data type. A pointer is simply the address of a variable in memory. Data types used with the API are often aliased into new data types to make them easier to work with. Each alias contains characters that describe the data type. The following table lists some of the characters you'll see in data types used with the API.

Character	Refers To	Used With
C	Constant	Pointers, values
DW	Double word	Numbers
H	Handle	Windows, device context, files
LP	Long Pointer	Pointers to data types
P	Pointer	Pointers to data types
STR	String	Strings (a string in C or C++ is a null-terminated array of characters)
T	String	The character T stands for <code>TCHAR</code> , which could be an ANSI or wide character depending on the platform
U	Unsigned	Numeric data types
W	Wide	Character arrays (strings)

Now you can interpret a data type that contains these characters to figure out what the data type actually means. After you know that, you can convert it to VBA. The following table lists some of the common types you'll come across with their VBA counterpart.

Chapter 2: Extending Applications Using the Windows API

API Data Type	VBA Data Type	Description
DWORD	Long	A DWORD, or double word is a 32-bit number
HDC	Long	Handle to a device context
HICON	Long	Handle to an icon
HWND	Long	Handle to a window
INT	Long	An int in C and C++ is 4-bytes, which corresponds to a Long in VBA
LONG	Long	
LPCWSTR	String	Long pointer to a constant wide string
LPTSTR	String	Long pointer to a wide or ANSI string
LPWSTR	String	Long pointer to a wide string
PULONG	Long	Pointer to an unsigned long
UINT	Long	Unsigned integer
void	None	A return type of void is a Sub procedure in VBA
void*/PVOID	Any	A void* is defined in the C language to accept any data type.
WCHAR	String	Wide character
WORD	Integer	A WORD is a 16-bit number

Given this table, you can see that most data type conversions from the API result in a Long or a String in VBA. In fact, this is usually the case. The tricky part when creating a Declare statement is getting the ByVal correct. More on that in the next section.

ByRef vs. ByVal

Unless otherwise specified, VBA passes arguments by reference. This means that you are given a pointer to the underlying data that enables you to change the data in the argument. C and C++, on the other hand, pass data by value by default meaning that data cannot be changed. Because most API functions were written for C and C++, most API functions expect you to pass arguments by value. In VBA you can do this using the ByVal keyword.

Strings

Strings in C and C++ are typically arrays of characters that end with a null-termination character. This character is defined as vbNullChar or Chr(0) in VBA. There are also two types of strings you will come across when working with API functions: ANSI and Wide.

Part I: Programming Access Applications

ANSI Strings

Characters in an ANSI string are 1 byte in length and typically consist of characters in the ANSI *character set*. A character set is the set of characters that are found in a given code page for a language. The ANSI character set contains Latin characters found in English, Spanish, French, and Italian, just to name a few.

API functions that accept string arguments should be aliased with the name of the function followed by an "A" in the Alias keyword. For example:

```
Private Declare Function MyFunction Lib "MyLibrary.dll" Alias "MyFunctionA" _  
    (ByVal szData As String)
```

Wide Strings

Characters in a wide string are made up of 2 bytes and can hold characters from languages that contain non-Latin characters, such as Cyrillic or Greek or Japanese or Chinese characters. Internally, the String data type in VBA stores strings as a Byte array with two-bytes per character. This means that you can assign a String value to a Byte array without conversion, as shown here:

```
Sub StringTest()  
    Dim strTest As String  
    Dim bytData() As Byte  
  
    strTest = "ABC"      ' A = Chr(65)  
    bytData = strTest  
  
    Stop  
  
End Sub
```

If you add a watch statement on the values of strTest and bytData, you'll see two bytes stored for each character in the array, as shown in Figure 2-1. The second byte of each character is a zero because the string contains ANSI characters.

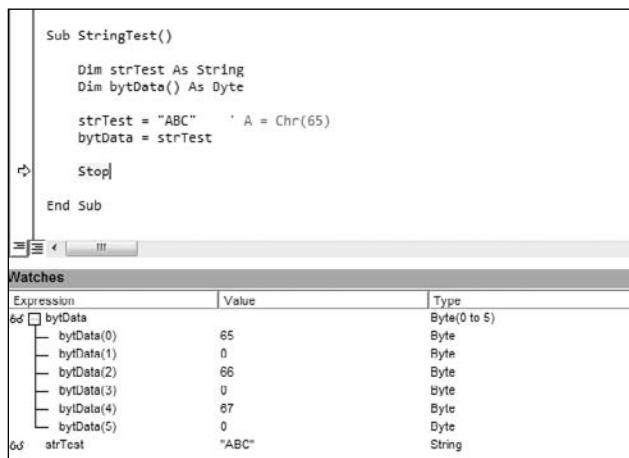


Figure 2-1

Chapter 2: Extending Applications Using the Windows API

API functions that accept string arguments should be aliased with the name of the function followed by a "W" in the Alias keyword. For example:

```
Private Declare Function MyFunction Lib "MyLibrary.dll" Alias "MyFunctionW" _  
    (ByVal szData As String)
```

When you use the function that returns a wide string you will likely have to use the StrConv function in VBA with the vbFromUnicode argument to read the string.

Structures

A structure in C or C++ is the equivalent of a user-defined type in VBA. These are declared using the `typedef struct` keywords.

Unions

Certain structures that you come across for use with API functions may include what is called a union. A union contains only one of its members at a given time. A union is useful when you want to pass data of a different data type to a function, or if you have an API feature that can accept one feature over another. For example, the TaskDialogIndirect API function uses a structure called `TASKDIALOGCONFIG`. This structure contains a union that allows you to specify either an icon defined in a file on the hard drive, or an icon in either a resource DLL or the system, but not both. The union defines which member to use, and there is a flag that tells the API to use the file on the hard drive. You can find out more about task dialogs and how to use them in your Access applications in the section "Windows Vista Task Dialogs."

VBA does not support the concept of a union, and as such, you can choose only one of the members of a union when converting a `struct` to a `Type` statement in VBA. Read the documentation for the API in question to determine which member you should use along with how to specify that member.

Example Conversion

With the data types behind us, let's take a look at an example conversion between C and VBA. The following function is one that you'll see later on so let's explain how to convert it. Take a look at the definition of the `ShellExecute` function as defined on MSDN:

```
HINSTANCE ShellExecute(  
    HWND hwnd,  
    LPCTSTR lpOperation,  
    LPCTSTR lpFile,  
    LPCTSTR lpParameters,  
    LPCTSTR lpDirectory,  
    INT nShowCmd  
) ;
```

The type declaration appears before the data type variable declaration in C-based languages, such as C, C++, and C#. Therefore, in the preceding declaration, `HINSTANCE` is the return type of the function. `HINSTANCE` refers to a handle to an instance. Because this is a handle, the return type is a `Long` in VBA.

The first parameter is defined as `HWND`, which we know is a handle to a window. Again, this is a `Long`. The next four parameters are defined as `LPCTSTR`. This refers to a long pointer to a constant string. Therefore, in VBA this becomes a `String`. Last, the `nShowCmd` parameter is defined as an `INT`, which is a `Long` in VBA.

Part I: Programming Access Applications

Once you've translated the data types, you need to know in which file this function is declared. Luckily for us, the MSDN documentation provides this information as well. Figure 2-2 shows additional information as taken from the MSDN documentation for `ShellExecute`.

Function Information	
Minimum DLL Version	shell32.dll version 3.51 or later
Custom Implementation	No
Header	shellapi.h
Import library	shell32.lib
Minimum operating systems	Windows NT 3.1, Windows 95
Unicode	Implemented as ANSI and Unicode versions.

Figure 2-2

In this description, we are interested in the information listed in the table that follows.

MSDN Description	Corresponding VBA Keyword
Minimum DLL Version	Lib
Unicode	Alias

Looking at the information for `ShellExecute` that appears in Figure 2-2, we know that the `Lib` keyword should be set to `shell32.dll`. Because this function accepts string arguments, and is implemented as both ANSI and Unicode, we can use either `ShellExecuteA` or `ShellExecuteW` as the `Alias`.

When you put it all together, the `Declare` statement for the `ShellExecute` API function becomes:

```
Declare Function ShellExecute Lib "shell32.dll" Alias "ShellExecuteA" ( _  
    ByVal hWnd As Long, _  
    ByVal lpOperation As String, _  
    ByVal lpFile As String, _  
    ByVal lpParameters As String, _  
    ByVal lpDirectory As String, _  
    ByVal nShowCmd As Long) As Long
```

Error Messages from API Functions

When an API function returns an error, you can check the `LastDLLError` property of the `Err` object in VBA. This property contains the most recent error code from a function that is defined with the `Declare` statement. Most API functions return an error code to indicate pass or fail. When these functions fail, they will not raise an error, which means you cannot include error handling for an API function. As a result, the `Description` property of the `Err` object is also empty because an error hasn't occurred! So, how can you get more information about the actual error that occurred?

Chapter 2: Extending Applications Using the Windows API

Windows contains the `FormatMessage` API function for exactly this purpose. This function returns the error message for a given error as indicated by the `LastDLLError` property. The declaration for this function is as follows:

```
Private Declare Function FormatMessage Lib "kernel32" Alias _  
    "FormatMessageA" (ByVal dwFlags As Long, lpSource As Long, _  
    ByVal dwMessageId As Long, ByVal dwLanguageId As Long, _  
    ByVal lpBuffer As String, ByVal nSize As Long, Arguments As Any) _  
    As Long
```

To tell the API to look up a system error message, include the following constant value:

```
Private Const FORMAT_MESSAGE_FROM_SYSTEM = &H1000
```

Now, you can use this function to retrieve the system error message when an API function fails:

```
Public Function GetAPIErrorMessage(lngError As Long) As String  
    Dim strMessage As String  
    Dim lngReturn As Long  
    Dim nSize As Long  
    strMessage = Space(256)  
    nSize = Len(strMessage)  
    lngReturn = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, 0, _  
        lngError, 0, strMessage, nSize, 0)  
    If lngReturn > 0 Then  
        GetAPIErrorMessage = Replace(Left(strMessage, lngReturn), vbCrLf, "")  
    Else  
        GetAPIErrorMessage = "Error not found."  
    End If  
End Function
```

Retrieving System Information

There will be times in your applications where you want to retrieve certain information about the system that cannot be obtained directly in VBA or Access. The classic example of such a scenario is retrieving the name of the user who is currently logged on to the computer. We'll first take a look at this example, but using the newer version of the function.

Return the User Name in Different Formats

Asking Windows for the name of the currently logged on user is a common request. This is useful for Access applications to restrict access to individual records or objects in a database. You may be familiar with this function that returns the name of the user in the format: `DOMAIN\UserName`, but there are other formats available as well that you can retrieve using the `GetUserNameEx` API function. Start with the `Declare` statement for the function:

```
Public Declare Function GetUserNameEx Lib "Secur32.dll" Alias "GetUserNameExA" ( _  
    ByVal NameFormat As EXTENDED_NAME_FORMAT, _  
    ByVal lpNameBuffer As String, _  
    ByRef lpnSize As Long) As Long
```

Part I: Programming Access Applications

Notice that the last argument is passed by reference using the `ByRef` modifier because the argument, `lpszName`, is a long pointer to a number. This means the size of the buffer is filled in by the API function. In order for the function to fill in the value it must receive a pointer. In VBA this is done by passing a variable by reference using the `ByRef` keyword. Access is likely to crash if you try to pass this argument by value using the `ByVal` modifier. This is because Windows cannot write to the memory that is specified by the variable.

The first argument, `EXTENDED_NAME_FORMAT`, is an `Enum` that is also defined on MSDN. We talk more about `Enums` in Chapter 3. For now, here is the declaration of the `Enum`:

```
Public Enum EXTENDED_NAME_FORMAT
    NameUnknown = 0
    NameFullyQualifiedDN = 1
    NameSamCompatible = 2
    NameDisplay = 3
    NameUniqueId = 6
    NameCanonical = 7
    NameUserPrincipal = 8
    NameCanonicalEx = 9
    NameServicePrincipal = 10
    NameDnsDomain = 12
End Enum
```

With the `Declare` and the `Enum` written, we just need to use the function. The following VBA function wraps the API to retrieve the user name in the specified format. Start with the declarations of the function:

```
Function GetUserNameHelper(NameFormat As EXTENDED_NAME_FORMAT) As String
    Dim buffer As String      ' space to receive data
    Dim rc      As Long       ' return code
    Dim pSize   As Long       ' pointer to a long
                            ' this is the number of characters which is
                            ' returned by the function
```

Because most API functions return error codes, the majority of them work by passing in a buffer (of spaces) and filling in the buffer. With that in mind, fill in the buffer with some spaces using the `Space` function in VBA. Also, set the size of the buffer using the `Len` function in VBA.

```
' allocate some space
buffer = Space(255)
pSize = Len(buffer)
```

Now we need to call the function. We'll pass in the format, buffer, and size to the `GetUserNameEx` API function. If it succeeds, `pSize` will contain the number of characters used in the user name. Using the `Left` function in VBA we can return that many characters from the buffer.

```
' return data...
If (GetUserNameEx(NameFormat, buffer, pSize)) Then
    GetUserNameHelper = Trim(Left(buffer, pSize))
End If

End Function
```

This function is most useful when you are logged on to a domain.

Return the Computer Name in Different Formats

As with the user name, there are different formats for the computer name that can also be retrieved. Again, we'll start with the Declare statement:

```
Public Declare Function GetComputerNameEx Lib "kernel32.dll" Alias  
"GetComputerNameExA" ( _  
    ByVal NameType As COMPUTER_NAME_FORMAT, _  
    ByVal lpBuffer As String, _  
    ByRef lpnSize As Long) As Long
```

Followed by the COMPUTER_NAME_FORMAT Enum:

```
Public Enum COMPUTER_NAME_FORMAT  
    ComputerNameNetBIOS  
    ComputerNameDnsHostname  
    ComputerNameDnsDomain  
    ComputerNameDnsFullyQualified  
    ComputerNamePhysicalNetBIOS  
    ComputerNamePhysicalDnsHostname  
    ComputerNamePhysicalDnsDomain  
    ComputerNamePhysicalDnsFullyQualified  
    ComputerNameMax  
End Enum
```

And the VBA function:

```
Private Function GetComputerNameHelper(NameType As COMPUTER_NAME_FORMAT) As String  
    Dim buffer As String      ' space to receive data  
    Dim rc      As Long       ' return code  
    Dim pSize   As Long       ' pointer to a long  
                            ' this is the number of characters which is  
                            ' returned by the function  
  
    ' allocate some space  
    buffer = Space(255)  
    pSize = Len(buffer)  
  
    ' return data...  
    If (GetComputerNameEx(NameType, buffer, pSize)) Then  
        GetComputerNameHelper = Trim(Left(buffer, pSize))  
    End If  
  
End Function
```

This function is also most useful when you are logged on to a domain.

Retrieve the Command Line

VBA contains a function called Command, which is used to return the portion of the command line that follows the /cmd command-line switch. This is very useful, but what if you want the entire command line? For example, if you wanted to know whether your application was started using a shortcut or was opened with the Open dialog box, you would need to parse the entire command line. To do this, you can use the GetCommandLine API function.

Part I: Programming Access Applications

Let's start with the Declare statement for GetCommandLine:

```
Private Declare Function GetCommandLine Lib "kernel32.dll" Alias _  
    "GetCommandLineA" () As Long
```

This appears deceptively simple. The function accepts no parameters and simply returns a Long. However, the return value of the function is the actual pointer to the string containing the command line used to start the application. It does not fill in a buffer. Therefore, we'll have to use some other API functions to get the data out of memory.

The next API function we'll declare is called CopyMemory. This function is actually defined in Windows using the name RtlMoveMemory and is used to copy a block of memory from one location to another. Here is the Declare statement:

```
Private Declare Function CopyMemory Lib "kernel32.dll" Alias "RtlMoveMemory" ( _  
    ByVal Destination As Any, _  
    ByVal pvSource As Any, _  
    ByVal Length As Long) As Long
```

Notice the use of the Any data type. This tells VBA not to check the data type for a given variable. The following table lists the arguments for the function and what they mean.

Argument	Meaning
Destination	This is the destination for data that will be copied from memory.
pvSource	Source of the data that will be copied. In the section "C to VBA Data Type Conversions," we defined a void* as being any type of data.
Length	The length of data to copy.

Because the GetCommandLine function returns a pointer to the string, we won't know how long the string is to pass to CopyMemory. So, we need one more API function to determine the length of data given a pointer. This function is called lstrlenW and is declared as follows:

```
Private Declare Function lstrlenW Lib "kernel32.dll" (lpString As Any) As Long
```

This function takes a single parameter that is the pointer to a string and is also defined using the Any data type.

Now that you've defined the API functions needed to return the command line, we need to put them together. It's actually quite simple once you see how it works. Start with the following declaration for a new VBA function:

```
Public Function CommandLine() As String  
    Dim pcl As Long      ' pointer to the command-line  
    Dim n    As Long      ' length of command-line  
    Dim dest As String   ' destination
```

Chapter 2: Extending Applications Using the Windows API

Next, we need to get the pointer to the command line using `GetCommandLine`. If the API function returns 0, then there was a problem and we exit the routine.

```
' read the command-line
pcl = GetCommandLine()
If (pcl = 0) Then
    MsgBox "Unable to read the command-line", vbExclamation
    Exit Function
End If
```

Good. Next, get the length of the command line. Here's where it gets a little tricky.

```
' find the length of the command-line
n = (lstrlenW(ByVal pcl) * 2) - 1
```

The `lstrlenW` API function returns the number of *wide* characters. A wide character is defined as two bytes instead of one character for ANSI. Therefore, there are half as many wide characters as there are ANSI characters. So, multiply the number of wide characters by 2, and then subtract 1 to account for the null-terminator in the string.

Of course, we want to make sure that we received something that was valid. If the length is 0, we know we have a problem and we exit the routine.

```
If (n = 0) Then
    MsgBox "Unable to determine the length of the command-line", vbExclamation
    Exit Function
End If
```

Once you've figured out the length, you need to copy the command line into your buffer. For this, allocate some space using the `Space` function with the number of characters calculated earlier.

```
dest = Space(n)
```

After that, copy the data in memory and return.

```
CopyMemory dest, ByVal pcl, n
CommandLine = Trim(dest)
End Function
```

Great! You have successfully copied data out of memory using a pointer. And who says that VBA can't use pointers?

Because a string in C and C++ is really just an array of characters, strings in these languages typically end with a null-terminator character to signify the end of the string. Oftentimes when working with C-strings in VB or VBA, you will see a +1 or -1 to account for this character. The null-terminator character is defined as Chr(0) or vbNullChar in VB and VBA.

Windows Shell Functions

The Windows Shell is the container that displays all user interface elements in Windows and is defined in `shell32.dll`. The Shell includes such items as the Desktop, Start menu, Windows Explorer, task bar, dialog boxes and message boxes, and so on. In addition to being at the core of the Windows operating system, it exposes some really cool API functions!

The following sections contain a very small subset of functions that are defined in the Shell. Some of this functionality is also defined in the `FileSystemObject` object, which is defined as a part of the Microsoft Scripting Runtime reference. Using API functions to create the same functionality allows you to remove a reference to this library!

Get the Path to a Special Folder

A special folder is a built-in folder in Windows such as the My Documents folder or Application Data folder. Because Windows knows where these folders are stored, it makes sense that there is a separate API function to retrieve the path to a special folder. The function is called `SHGetFolderPath` and it is defined here:

```
Public Declare Function SHGetFolderPath Lib "shell32.dll" _
    Alias "SHGetFolderPathA" _
    (ByVal hwndOwner As Long, _
     ByVal nFolder As Long, _
     ByVal hToken As Long, _
     ByVal dwFlags As Long, _
     ByVal pszPath As String) As Long
```

Windows identifies special folders using an ID value called a `CSIDL`. There are many `CSIDL` values defined by Windows but only a subset of them is supported by this function. To make these values easy to work with, create an `Enum` called `CSIDL` to contain the values:

```
Public Enum CSIDL
    CSIDL_ADMINTOOLS = &H30
    CSIDL_COMMON_ADMINTOOLS = &H2F
    CSIDL_APPDATA = &H1A
    CSIDL_COMMON_APPDATA = &H23
    CSIDL_COMMON_DOCUMENTS = &H2E
    CSIDL_COOKIES = &H21
    CSIDL_HISTORY = &H22
    CSIDL_INTERNET_CACHE = &H20
    CSIDL_LOCAL_APPDATA = &H1C
    CSIDL_MYPICTURES = &H27
    CSIDL_PERSONAL = &H5
    CSIDL_PROGRAM_FILES = &H26
    CSIDL_PROGRAM_FILES_COMMON = &H2B
    CSIDL_SYSTEM = &H25
    CSIDL_WINDOWS = &H24
End Enum
```

These values will be used in the `nFolder` argument of `SHGetFolderPath`. The API function also expects a flag value that is used to determine whether to return the current path or the default path for a folder.

Chapter 2: Extending Applications Using the Windows API

This flag applies to folders that can be moved by the user such as the My Documents folder or the Temporary Internet Files directory. The flag values can also be represented in an `Enum`:

```
Public Enum SHGFP_TYPE
    SHGFP_TYPE_CURRENT = 0      ' // current value for user, verify it exists
    SHGFP_TYPE_DEFAULT = 1      ' // default value, may not exist
End Enum
```

Now, all you need to do to call the API function is to use this code:

```
Public Function GetSpecialFolderPath(id As CSIDL) As String
    Const MAX_PATH As Long = 260
    Dim rc      As Long
    Dim stPath As String

    ' allocate some space for the path
    stPath = Space(MAX_PATH)

    ' call the API
    rc = SHGetFolderPath(0, id, 0, SHGFP_TYPE.SHGFP_TYPE_CURRENT, stPath)
    If (rc = 0) Then
        GetSpecialFolderPath = Left(stPath, InStr(stPath, vbNullChar) - 1)
    End If
End Function
```

Show the Property Dialog Box for a File

Any time we learn a new programming language or a new version of Access is released, we rewrite the same example application — a photo manager application to catalog digital photos that we take. Digital photos have a lot of metadata associated with them, such as the camera maker and model, exposure time, ISO speed, and so on. Rather than parse this data from the image file, we just decided to show the Properties dialog box for the file because Windows already knows about all of this information. Figure 2-3 shows the Properties dialog box for a JPG file in Windows Vista.

When you open a file in the Windows Shell, you are sending the application associated with the file a *verb*. A verb for a file is defined as an action that can be taken upon the file such as *open* or *print*. It turns out that in order to view the Properties dialog box for a file, you simply need to send it the *properties* verb. So, how do you send a verb to a given file you ask? For that, we can use the `ShellExecuteEx` API function defined here:

```
Private Declare Function ShellExecuteEx Lib "shell32.dll" _
(lpExecInfo As SHELLEXECUTEINFO) As Long
```

The `ShellExecuteEx` function accepts one argument — a variable of type `SHELLEXECUTEINFO`. This is a user-defined type defined as follows:

```
Private Type SHELLEXECUTEINFO
    cbSize      As Long
    fMask       As Long
    HWND        As Long
    lpVerb      As String
```

Part I: Programming Access Applications

```
lpFile      As String
lpParameters As String
lpDirectory As String
nShow       As Long
hInstApp    As Long
lpIDList    As Long
lpClass     As String
hkeyClass   As Long
dwHotKey   As Long
hIcon       As Long
hProcess    As Long
End Type
```

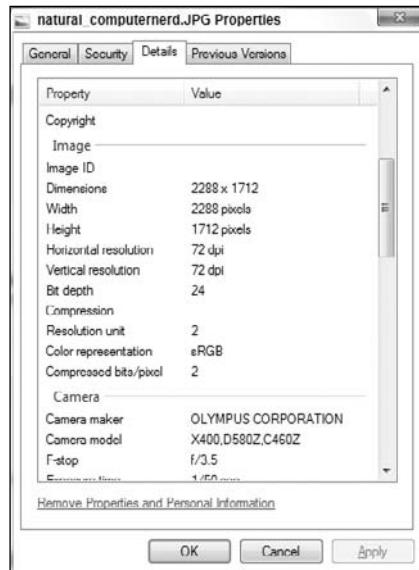


Figure 2-3

In order to view the Properties dialog box, you must tell the API function to invoke the verb from the shortcut menu for the file, not the registry. Do this by setting the fMask parameter of SHELLEXECUTEINFO to the following constant:

```
Private Const SEE_MASK_INVOKEIDLIST As Long = &H
```

In the code that follows, we set the size of the SHELLEXECUTEINFO Type, along with the file name, verb, and mask.

```
Public Sub OpenPropertiesDialog(stFile As String)
    Dim rc As Long
    Dim ei As SHELLEXECUTEINFO

    ' set some data in the Type
    ei.HWND = hWndAccessApp()
    ei.lpFile = stFile
```

```
ei.lpVerb = "properties"
ei.fMask = SEE_MASK_INVOKEIDLIST
ei.cbSize = Len(ei)

' call the API
rc = ShellExecuteEx(ei)
End Sub
```

After calling the code, the Properties dialog box should be displayed for the specified file, as shown earlier in Figure 2-3.

Occasionally, you may see API functions documented on MSDN that use the prefixes cch or cb in parameter names. These prefixes stand for count of characters, or count of bytes respectively. On newer versions of Windows that support Unicode, one character is 2 bytes long, even for ANSI characters. Therefore, if you pass in a count of characters to a function that expects a count of bytes, you have passed in only one-half of the required amount of space. If you pass in a count of bytes to a function that expects a count of characters, you've passed in double the amount of space!

Determine Whether a Computer Is Connected to the Internet

Let's say that you have a registration process built in to your application and need to download a file for registration. This process would fail if the user wasn't connected to the Internet. You can determine whether the user is connected to the Internet using the `InetIsOffline` function defined in `url.dll`. This allows you to give the user a nice error message when they are not connected.

We'll implement a `Property` procedure called `IsConnected` to wrap this function:

```
Private Declare Function InetIsOffline Lib "url.dll" _
    (ByVal dwFlags As Long) As Long
Public Property Get IsConnected() As Boolean
    IsConnected = (Not InetIsOffline(0))
End Property
```

You can read more about adding registration to your application in Chapter 14.

Determine Whether a User Is an Administrator

There may be times in your applications when you want to know if the user who is logged on to the machine is an administrator on the machine. For example, say you want to tie a process such as connecting to a SQL Server in the database to the Windows security on the machine. To do so, you can call the `IsUserAnAdmin` API function. This one is simple and is defined as follows:

```
Public Declare Function IsUserAnAdmin Lib "shell32.dll" () As Boolean
```

Because this function doesn't take any arguments, it is acceptable to declare it as `Public` and call it directly. As a reminder, if you are wrapping these functions in a class module you'll need to declare it as `Private`.

On Windows Vista, this function returns True if the process is running under elevated privileges. Even as a member of the local Administrators group on the computer it may return False.

Controlling Processes and Threads

You may want to control external processes within an application. For example, you might want to create another process and wait for it to finish or even shut down another application. To do this, Windows provides many functions to control *processes* and *threads*. A process typically refers to an executable running on the machine. The executable can be in the form of a .exe file that runs in its own process space (out-of-process), or a .dll or .ocx file that runs in the context of another application's process space (in-process).

A thread is a single unit of work within a process. Processes often create additional threads to provide responsive user interfaces or to improve performance.

Determining the Current Process ID

Processes are identified using a process identifier, or PID. You can use Task Manager to display the PID along with other information associated with a process, as shown in Figure 2-4.

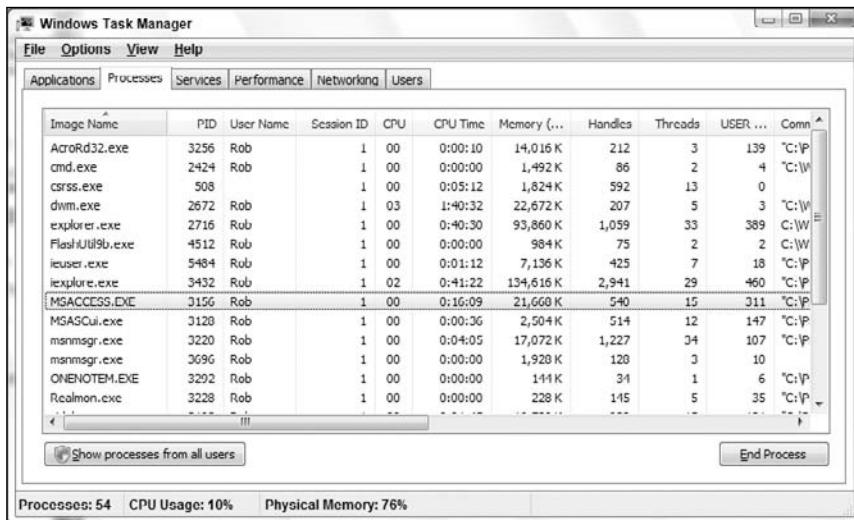


Figure 2-4

When you want to control the current Access process itself, you'll need to get the PID for the running instance of Access. The easiest way to do this is to use the `GetCurrentProcessId` function defined by Windows:

```
Public Declare Function GetCurrentProcessId Lib "kernel32.dll" () As Long
```

This function returns a `Long` integer that is the PID for the current process.

Pausing Code Execution

There may be times when you want to pause code execution for a specified amount of time. You can do this using the `Timer` function in a loop, but an easier way is to use the `Sleep` function defined by Windows:

```
Public Declare Sub Sleep Lib "kernel32.dll" (ByVal dwMilliseconds As Long)
```

This function accepts a `Long` integer value as a parameter that is the number of milliseconds to sleep. This function blocks the calling code until the delay is complete.

This function does not return a value and is declared as a Sub routine.

Quitting a Process

Have you ever automated an application only to have its process hang around? Or, have you ever started an application and wondered how to shut it down? Generally speaking, you can do this using the `TerminateProcess` API function, but let's take it a step further. The `TerminateProcess` function ends an application and is the equivalent of using the `End Process` command in Task Manager. This is a rather forceful way of exiting an application. As an alternative to this function, you can use the `SendMessage` API and ask the application in question to close. For this, we'll use a few different functions, as follows:

- ❑ `GetWindowThreadProcessId` — Returns the process ID for the specified window handle.
- ❑ `OpenProcess` — Returns the process handle for an existing process.
- ❑ `FindWindow` — Searches for a window handle with specified criteria.
- ❑ `SendMessage` — Sends a Windows message to an application. (For more information about Windows messages, see the article titled “About Messages and Message Queues” in MSDN.)
- ❑ `TerminateProcess` — Forces a process to exit.
- ❑ `CloseHandle` — Closes the handle for an object.

Okay, let's get started with the `Declare` statements for each of these. We explain each one as we use them.

```
' Returns the PID for a process with the specified window handle
Private Declare Function GetWindowThreadProcessId Lib "user32.dll" ( _
    ByVal hWnd As Long, _
    ByRef lProcessID As Long) As Long

' Opens an existing process
Private Declare Function OpenProcess Lib "kernel32.dll" ( _
    ByVal lDesiredAccess As Long, _
    ByVal bInheritHandle As Boolean, _
    ByVal lProcessID As Long) As Long

' Find a window handle based on a class name or window title
Private Declare Function FindWindow Lib "user32.dll" Alias "FindWindowA" ( _
    ByVal lpClassName As String, _
    ByVal lpWindowName As String) As Long

' send a windows message to an application
Private Declare Function SendMessage Lib "user32.dll" Alias "SendMessageA" ( _
    ByVal hWnd As Long, _
    ByVal wMsg As Long, _
```

Part I: Programming Access Applications

```
ByVal wParam As Long, _
ByVal lParam As Long) As Long
' Terminates a process
Private Declare Function TerminateProcess Lib "kernel32.dll" ( _
    ByVal hProcess As Long, _
    ByVal uExitCode As Long) As Long
' closes a process handle
Private Declare Function CloseHandle Lib "kernel32.dll" ( _
    ByVal hObject As Long) As Boolean
```

Also, declare the constants that we will use for this exercise:

```
Private Const WM_CLOSE As Long = &H10
Private Const PROCESS_TERMINATE As Long = 1
```

You will actually write two routines to exit processes:

- KillProcessFromWindow — Exits a process using the specified window handle
- KillProcessFromClassName — Exits a process using the specified window class

The first routine, KillProcessFromWindow is called from the second one so let's start there, beginning with the declarations. Notice that we've added an optional argument called Force, which defaults to False. This is used to determine whether to call the SendMessage function or the TerminateProcess function.

```
Public Sub KillProcessFromWindow(hWin As Long, Optional Force As Boolean = False)
    Dim lProcessID As Long
    Dim hProcess As Long
```

If you are not forcing the application to close, you ask it to close by sending it a WM_CLOSE Windows message using the SendMessage function.

Add the following code to the routine:

```
' If we are forcing a close, call TerminateProcess API,
' otherwise, send the window a WM_CLOSE message
If (Not Force) Then
    ' shut down the application gracefully by asking it to close
    SendMessage hWin, WM_CLOSE, 0, 0
Else
```

If you are asking the application to forcefully close, you call the TerminateProcess API. For that, you first need the process ID that you'll get from the GetWindowThreadProcessId function:

```
' get the process ID for the specified window
GetWindowThreadProcessId hWin, lProcessID
```

Now, with the process ID, you can get the process handle for the process using the OpenProcess function. In order to terminate the process, you need to ask for PROCESS_TERMINATE rights:

```
' get the process handle for the PID
hProcess = OpenProcess(PROCESS_TERMINATE, False, lProcessID)
```

Chapter 2: Extending Applications Using the Windows API

Once you have the process handle, you just need to tell it to exit using `TerminateProcess`. Add the following code to exit the process and close the routine:

```
' terminate the process
TerminateProcess hProcess, 0

' close the process handle
CloseHandle hProcess
End If
End Sub
```

As you can see, this routine does the work of actually closing the application, but what if you don't have the window handle? What if you can get something else such as the *window class* name for the application? The window class is a name that is registered for an application with Windows. For Access, the window class is called `OMAIN`. For Excel, the window class is `XLMMAIN`.

You can get the window class for an application using a tool such as SpyXX that is included with Visual Studio or the GetClassName API function.

One way or another, you need the window handle to exit the application. This is required for the case where you call the `SendMessage` API or in the case where you call the `TerminateProcess` API to retrieve the process handle. If you only know the window class or window title, you can use the `FindWindow` API function to get the window handle. This is what the `KillProcessFromClassName` routine does. Start by adding the declarations:

```
Public Sub KillProcessFromClassName(ClassName As String, _
    Optional Force As Boolean = False, _
    Optional AllProcesses As Boolean = True)

    Dim lProcessID As Long
    Dim hProcess As Long
    Dim hWin As Long
```

Notice that this routine includes an additional `Optional` argument called `AllProcesses`, which we have defaulted to `True`. If you have multiple instances of an application running, specifying the `AllProcesses` argument loops through all windows until there are no matching windows.

Add the following code to find the first window:

```
' find the first window
hWin = FindWindow(ClassName, vbNullString)
```

Now, add the code to call the `KillProcessFromWindow` function that you wrote earlier. If you pass the argument to shut down all processes, you'll call the `FindWindow` function again; otherwise set the window handle to zero to exit the loop. The `FindWindow` function also returns zero if it doesn't find a matching window handle.

```
Do While (hWin <> 0)
    ' shut down the process and exit
    KillProcessFromWindow hWin, Force

    ' if we are shutting down all processes
```

Part I: Programming Access Applications

```
' get the next window, otherwise set the
' window handle to 0
If (AllProcesses) Then
    hWin = FindWindow(ClassName, vbNullString)
Else
    hWin = 0
End If
Loop
End Sub
```

Waiting for a Process to Complete

You can use the `Shell` function in VBA to launch another process, but the function returns right away. It does not wait for the application that was launched to finish running before returning to your code. In many cases this is undesirable. For example, if the application you are launching using the `Shell` function generates an output file that you need to use in your Access application, then the file may not be available to you until the other application has completed.

To determine when the process has finished, you need to get a handle to the process instead of its process identifier. Therefore, you have to use the `CreateProcess` API function to launch the process instead of the `Shell` function.

Create the Process

To create the process, use the `CreateProcess` API, defined as follows:

```
Private Declare Function CreateProcess Lib "kernel32" Alias "CreateProcessA" ( _
    ByVal lpApplicationName As Long, _
    ByVal lpCommandLine As String, _
    ByVal lpProcessAttributes As Long, _
    ByVal lpThreadAttributes As Long, _
    ByVal bInheritHandles As Long, _
    ByVal dwCreationFlags As Long, _
    ByVal lpEnvironment As Long, _
    ByVal lpCurrentDirectory As Long, _
    lpStartupInfo As STARTUPINFO, _
    lpProcessInformation As PROCESS_INFORMATION) As Long
```

As you can see, this API uses two user-defined types: `STARTUPINFO` and `PROCESS_INFORMATION`. These types are defined as follows:

```
Private Type STARTUPINFO
    cb As Long
    lpReserved As String
    lpDesktop As String
    lpTitle As String
    dwX As Long
    dwY As Long
    dwXSize As Long
    dwYSize As Long
    dwXCountChars As Long
    dwYCountChars As Long
```

```
dwFillAttribute As Long
dwFlags As Long
wShowWindow As Integer
cbReserved2 As Integer
lpReserved2 As Long
hStdInput As Long
hStdOutput As Long
hStdError As Long
End Type
Private Type PROCESS_INFORMATION
    hProcess As Long
    hThread As Long
    dwProcessID As Long
    dwThreadID As Long
End Type
```

The STARTUPINFO type provides information about how the process should be created. The PROCESS_INFORMATION type contains information about the process once it has been created. If successful, the CreateProcess function returns the process handle for the application in the hProcess member of the PROCESS_INFORMATION type. We ask the process to create with normal priority using the following constant:

```
Private Const NORMAL_PRIORITY_CLASS As Long = &H20
```

Waiting for the Process

After you start the process using the CreateProcess function, you can wait for it to complete using the WaitForSingleObject API function. This function accepts two arguments: the process handle for the process to wait for, and a timeout interval in milliseconds. The function is declared as follows:

```
Private Declare Function WaitForSingleObject Lib "kernel32" ( _
    ByVal hHandle As Long, _
    ByVal dwMilliseconds As Long) As Long
```

By specifying a timeout interval of zero, you can force the function to return a value of WAIT_TIMEOUT. When the process that you started has completed, the function returns zero. Therefore, you simply need to call this function in a loop as you see in a moment. The WAIT_TIMEOUT constant is defined as:

```
Private Const WAIT_TIMEOUT As Long = &H102
```

Closing the Process

Processes you open with either the OpenProcess or CreateProcess functions should be closed using the CloseHandle API function. This function is declared as follows:

```
Private Declare Function CloseHandle Lib "kernel32" ( _
    ByVal hObject As Long) As Long
```

Putting It Together

Here is the completed code to wait for the process. The VBA code shown here accepts a command line as an argument that is then passed to the CreateProcess function. It calls the WaitForSingleObject

Part I: Programming Access Applications

function in a loop while it returns the WAIT_TIMEOUT value. When the process has completed, the loop exits and returns control to the calling routine.

```
Public Sub RunApplication(stCmdLine As String)
    Dim pi As PROCESS_INFORMATION
    Dim si As STARTUPINFO
    Dim rc As Integer

    ' Initialize the STARTUPINFO structure
    si.cb = Len(si)

    ' run the specified command line
    rc = CreateProcess(0, stCmdLine, 0, 0, True, _
        NORMAL_PRIORITY_CLASS, 0, 0, si, pi)

    ' Wait for the application to finish
    Do While WaitForSingleObject(pi.hProcess, 0) = WAIT_TIMEOUT
        DoEvents
    Loop

    ' close the process handle
    CloseHandle pi.hProcess
End Sub
```

Testing the Routine

To test the routine you can pass the command line for an application to the RunApplication routine. For our test, we call the cmd.exe application with a command line to output the contents of the D: drive to d:\test_out.txt. When the process has completed, we import the results of this file into a table called tblHardDrive. This table has a field named FileName that receives the name of the file from the output file.

```
Sub RunApplicationTest()
    Dim stLine As String
    Dim rs As DAO.Recordset2

    ' run this command line
    RunApplication "cmd.exe /c d: & cd \ & dir /s /b > d:\test_out.txt"

    ' import..
    CurrentDb.Execute "DELETE * FROM tblHardDrive"
    Set rs = CurrentDb().OpenRecordset("tblHardDrive")

    Open "d:\test_out.txt" For Input Access Read As #1

    While (Not (EOF(1)))
        Line Input #1, stLine
        rs.AddNew
        rs!FileName = stLine
        rs.Update
    Wend

    Close #1
    rs.Close
    Set rs = Nothing
End Sub
```

Windows Vista Task Dialogs

You're probably familiar with the `MsgBox` function in VB and VBA, and chances are you've found it a bit lacking. For example, there's no easy way to change the button captions or to define your own buttons in a message box. As Access developers, we're lucky that we have the ability to create rich `Form` objects to work around this shortcoming (but wouldn't it still be nice if Windows provided something more).

Windows Vista introduces a new set of APIs to create a new type of object called a *task dialog*. A task dialog is like a message box, but so much more. In addition to simple text and a few buttons, task dialogs enable you to create additional controls, such as radio buttons or hyperlinks. Figure 2-5 shows an example task dialog prompt from Vista.

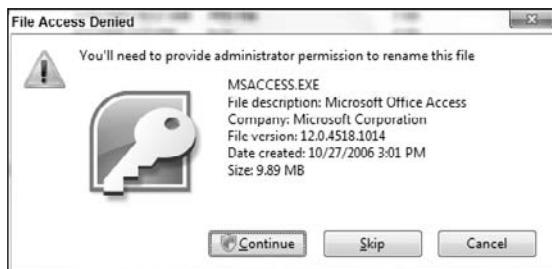


Figure 2-5

Task dialogs rock! And, you can use them in your application just by calling their APIs. Hang on to your hats — message boxes are cool again. Okay, they're not really message boxes, but they're still cool.

While task dialogs are harder to create than a message box, or even a custom Access form, we think they provide lots of opportunity to enhance Access applications that target Windows Vista. Task dialogs enable you to create user interfaces that feel as if they integrate with the operating system, and remove the need to maintain an additional form in the database.

Introduction to Task Dialogs

Task dialogs can be used for many things that message boxes would have been used for in the past. They are also used to gather information from the user, but the way in which users are presented with information has improved. For example, in addition to giving your user a button to click, you can also provide them with check boxes, radio buttons, or hyperlinks. This leads to user interfaces that tend to be easier to use and understand.

The task dialog API functions are defined in `comctl32.dll`. The functions, however, are not available to Access by default because Windows Vista actually includes two different versions of `comctl32.dll` — a version 5 library for legacy applications, and the new shiny version 6 for newer applications.

Modify the Access Manifest File

In order to use the new version of `comctl32.dll`, you must create a *manifest* file for your application. A manifest file is a special XML file that you use to tell Windows to include the newer versions of the common controls. The newer version of the common controls also includes the visual styles that provide Windows theming of form controls in Access.

Part I: Programming Access Applications

If you were writing a standalone application, you would simply tack on .manifest to the end of your application name to create the manifest file. For example, if your application was named MyAwesomeApplication.exe, you would name the manifest file MyAwesomeApplication.exe.manifest. Because your application in this case is Access, you need to modify the msaccess.exe.manifest file that is included with Access 2007.

Here's how:

1. Locate the msaccess.exe.manifest file. This is in the same directory as msaccess.exe — typically C:\Program Files\Microsoft Office\Office12.
2. Make a backup of the manifest file in case you run into problems.
3. Open the manifest file in your favorite text editor.
4. Add the following dependency entry to the manifest file, following the last dependency node in the file.

```
<dependency>
  <dependentAssembly>
    <assemblyIdentity
      type="win32"
      name="Microsoft.Windows.Common-Controls"
      version="6.0.0.0"
      processorArchitecture="X86"
      publicKeyToken="6595b64144ccf1df"
      language="*"
    />
  </dependentAssembly>
</dependency>
```

This XML tells Windows to use the newer version of comctl32.dll, which will provide the TaskDialog API function. If you do not modify the manifest file, you will see the error message displayed in Figure 2-6 when calling the API because the function is not defined in the older version of the DLL that is loaded.

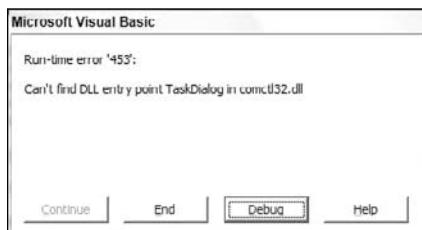


Figure 2-6

StrPtr

Before getting into the API functions, there is one more thing to be aware of. The user-defined type that stores the data associated with a task dialog uses pointers to strings, not strings directly. This is to make localization easier by way of resource DLLs. As a result, you won't be able to give the API a string value from VBA. You need to give it a pointer, but how do you get the address of a string? It so happens that

VB and VBA contain a hidden function to do this called `StrPtr`. This function returns the address of a String variable that can be used to pass to API functions.

There are two other pointer functions such as this in VB and VBA. `ObjPtr` returns the address to an Object variable and `VarPtr` returns the address of a Variant — or any other variable. We discuss the `ObjPtr` function more in Chapter 3.

A Simple Message Box Replacement

Okay, after you've modified the `msaccess.exe.manifest` file, you're ready to start calling the API. Task dialogs are created using one of two API functions: `TaskDialog`, or `TaskDialogIndirect`. The `TaskDialogIndirect` function provides extended functionality so let's start with the `TaskDialog` function.

```
Private Declare Function TaskDialog Lib "comctl32.dll" ( _  
    ByVal hwndParent As Long, _  
    ByVal hInstance As Long, _  
    ByVal pszWindowTitle As Long, _  
    ByVal pszMainInstruction As Long, _  
    ByVal pszContent As Long, _  
    ByVal dwCommonButtons As TASKDIALOG_COMMON_BUTTON_FLAGS, _  
    ByVal pszIcon As Long, _  
    pnButton As Long) As Long
```

Notice that there are no strings in the argument list because the `TaskDialog` API function expects pointers to strings as mentioned earlier. We pass pointers to strings using the `StrPtr` function.

The `dwCommonButtons` argument is an `Enum` that defines the buttons that appear in the task dialog. The `Enum` is defined as follows:

```
Public Enum TASKDIALOG_COMMON_BUTTON_FLAGS  
    TDCBF_OK_BUTTON = &H1  
    TDCBF_YES_BUTTON = &H2  
    TDCBF_NO_BUTTON = &H4  
    TDCBF_CANCEL_BUTTON = &H8  
    TDCBF_RETRY_BUTTON = &H10  
    TDCBF_CLOSE_BUTTON = &H20  
End Enum
```

Based on the values in the `Enum`, you might recognize them as being flags. As such, you can combine them using the `Or` keyword in VBA. We get to that in a minute. First, let's take a look at the VBA function to call the API. Again, start with the declaration. `MsgBox`, when used as a function, returns a variable of type `VbMsgBoxResult` that represents the button that was clicked. We do the same thing here by returning a variable of type `TASKDIALOG_COMMON_BUTTON_FLAGS` that you defined a moment ago.

```
Public Function SimpleTaskDialog( _  
    Optional Title As String, _  
    Optional MainInstruction As String, _  
    Optional Content As String, _  
    Optional Buttons As TASKDIALOG_COMMON_BUTTON_FLAGS) _  
    As TASKDIALOG_COMMON_BUTTON_FLAGS  
    Dim hr As Long  
    Dim pButton As Long
```

Part I: Programming Access Applications

We've made the arguments `Optional` so that we don't have to supply all of them.

One convenience that the `MsgBox` function includes is integration with Access. If you have defined a title for your application in the Access Options dialog box, Access will pass it to the `MsgBox` function so that the application title is used as the title of the message box. Task dialogs do not have this integration, so we add it here. For simplicity, we've wrapped it in an `On Error Resume Next` block in case the application title is not defined.

```
' Use the AppTitle if it is defined
On Error Resume Next
Title = CurrentDb().Properties("AppTitle")
On Error GoTo 0
```

Next, you need to call the API. Here's where you'll pass the string arguments in your VBA function wrapped in the `StrPtr` function to pass to the API.

```
' Call the API
hr = TaskDialog(hWndAccessApp(), 0, _
    StrPtr>Title), _
    StrPtr>MainInstruction), _
    StrPtr>Content), -
    Buttons, -
    pIcon, -
    pButton)
```

Last, verify the function was called correctly and return the `pButton` argument. This is the button that the user clicked in the task dialog.

```
' make sure we created the dialog
If (hr <> 0) Then
    MsgBox "Unable to create task dialog", vbCritical
    Exit Function
End If

SimpleTaskDialog = pButton
End Function
```

Here are some examples of the `SimpleTaskDialog` function in action. To display a task dialog with yes and no buttons, use the `Or` keyword to add multiple buttons.

```
SimpleTaskDialog "", "This is the main instruction", _
    "Your content goes here", -
    TDCBF_YES_BUTTON Or TDCBF_NO_BUTTON
```

This task dialog appears in Figure 2-7.

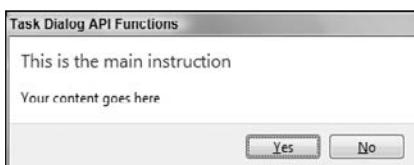


Figure 2-7

Chapter 2: Extending Applications Using the Windows API

You can create content that spans multiple lines to give your user enough information, as shown in the following code:

```
SimpleTaskDialog "", _  
    "Connection to server failed", _  
    "Click Retry to try to perform the following steps:" & _  
    vbCrLf & vbCrLf & "1. This is step 1" & _  
    vbCrLf & "2. This is step 2" & _  
    vbCrLf & "3. This is step 3", _  
    TDCBF_RETRY_BUTTON Or TDCBF_CANCEL_BUTTON
```

This task dialog is shown in Figure 2-8.



Figure 2-8

Or, if you really want to get fancy, you can span multiple lines in the main instruction and add even more buttons, as shown in the following code:

```
Sub TDAllButtons()  
    SimpleTaskDialog "", "Main" & vbCrLf & "Instruction", _  
        "Your content goes here", _  
        TDCBF_YES_BUTTON Or TDCBF_NO_BUTTON Or TDCBF_OK_BUTTON Or _  
        TDCBF_CLOSE_BUTTON Or TDCBF_NO_BUTTON Or TDCBF_RETRY_BUTTON Or _  
        TDCBF_CANCEL_BUTTON  
End Sub
```

Figure 2-9 shows this task dialog.



Figure 2-9

More Complex Task Dialogs

As you can see, task dialogs are pretty flexible, but we've only scratched the surface so far. There's still much more that you can do with them. In the next sections, you learn how to create task dialogs with these features:

- Expanded section
- Footer section
- Custom button text
- Command links
- Radio buttons
- Verification text (check box)
- Hyperlinks

Adding an Expanded Section

One distinct advantage that task dialogs have over message boxes is the ability to add an expanded section to provide more information to the user. This is done using the `TASKDIALOGCONFIG` user defined type as an argument of the `TaskDialogIndirect` API function.

The number of possibilities for using this function makes the use of optional arguments a little messy, so start a new module for these examples.

The `TaskDialogIndirect` function is defined here:

```
Private Declare Function TaskDialogIndirect Lib "comctl32.dll" ( _  
    pTaskConfig As TASKDIALOGCONFIG, _  
    pnButton As Long, _  
    pnRadioButton As Long, _  
    pfVerificationFlagChecked As Boolean) As Long
```

The `TASKDIALOGCONFIG` type enables you to supply many more configuration options for a task dialog including expanded information, and the specific text that is displayed for the expand button. The type is defined as follows:

```
Public Type TASKDIALOGCONFIG  
    cbSize                 As Long  
    hwndParent             As Long  
    hInstance              As Long  
    dwFlags                As Long  
    dwCommonButtons        As TASKDIALOG_COMMON_BUTTON_FLAGS  
    pszWindowTitle         As Long  
    pszMainIcon            As TASKDIALOG_SYSTEM_ICONS  
    pszMainInstruction     As Long  
    pszContent             As Long  
    cButtons               As Long  
    pButtons               As Long 'TASKDIALOG_BUTTON  
    nDefaultButton          As Long  
    cRadioButtons           As Long
```

Chapter 2: Extending Applications Using the Windows API

```
pRadioButtons           As Long   ' TASKDIALOG_BUTTON
nDefaultRadioButton    As Long
pszVerificationText    As Long
pszExpandedInformation As Long
pszExpandedControlText As Long
pszCollapsedControlText As Long
pszFooterIcon          As TASKDIALOG_SYSTEM_ICONS
pszFooter              As Long
pfCallback             As Long   ' PFTASKDIALOGCALLBACK
lpCallbackData          As Long   ' LONG_PTR
cxWidth                As Long
End Type
```

Before we get into the details of the arguments, you'll notice that this type includes a type called `TASKDIALOG_SYSTEM_ICONS`. This is an enumeration that defines the built-in constant values for system icons that are available with task dialogs. We discuss icons in more detail later in this chapter, but we've defined the enumeration here so the code will compile.

```
Public Enum TASKDIALOG_SYSTEM_ICONS
    IDI_APPLICATION = 32512
    TD_QUESTION_ICON = 32514
    TD_SECURITY_SHIELD_GRAY_ICON = 65527
    TD_SECURITY_SUCCESS_ICON = 65528
    TD_SECURITY_ERROR_ICON = 65529
    TD_SECURITY_WARNING_ICON = 65530
    TD_SECURITY_SHIELD_BLUE_ICON = 65531
    TD_SECURITY_SHIELD_ICON = 65532
    TD_INFORMATION_ICON = 65533
    TD_ERROR_ICON = 65534
    TD_WARNING_ICON = 65535
End Enum
```

For now, you'll focus on `pszExpandedInformation`, `pszExpandedControlText`, and `pszCollapsedControlText`, but later you learn more about other arguments in this type. Let's take a look at how you can use these. In a new module, start with the following variables in the declarations section. Go ahead and add all of these because you'll work with the others later.

```
' Data for TASKDIALOGCONFIG
Public strTitle As String
Public strMainInstruction As String
Public strContent As String
Public strExpandedInformation As String
Public strCollapsedControlText As String
Public strExpandedControlText As String
Public strFooter As String
Public strVerificationText As String
```

Next, add the VBA function that you'll use to show the expanded text.

```
Public Function TaskDialogWithExpandedSection() As TASKDIALOG_COMMON_BUTTON_FLAGS
    Dim hr As Long
    Dim pButton As Long
    Dim tdc As TASKDIALOGCONFIG
```

Part I: Programming Access Applications

Notice that we've declared a variable named `tdc` as type `TASKDIALOGCONFIG`. You'll set members of the type in just a moment. As with the `TaskDialog` example, you default to using the title of the application.

```
' Use the AppTitle if it is defined
On Error Resume Next
strTitle = CurrentDb().Properties("AppTitle")
On Error GoTo 0
```

Next, set the text for the expand button control using the variables you declared earlier. In a real application you might use a class module and expose these as properties to make it easier to work with.

```
strCollapsedControlText = "More Details"
strExpandedControlText = "Less Details"
```

The expanded information section can display much more information to the user. For this example, you simulate an error using the `Err.Raise` statement. In a real-world application, you might integrate the task dialog into your error handler to display an error to the user.

```
' simulate an error
On Error Resume Next
Err.Raise 5
strExpandedInformation = "Source: " & Err.Source & vbCrLf & _
                        "Description: " & Err.Description & vbCrLf & vbCrLf & _
                        "CurrentObjectName: " & CurrentObjectName & vbCrLf & _
                        "CurrentObjectType: " & CurrentObjectType
On Error GoTo 0
```

Great — now set the main instruction and title text to give the dialog box more meaning.

```
' Set the main instruction and content
strMainInstruction = "An error has occurred"
strContent = "Content goes here"
```

It's time to start working with the `TASKDIALOGCONFIG` type. Many of the structures that you use with API functions include a member named `cbSize`. This represents the size of the structure, which you can get using the `Len` function in VBA. Also, set the window handle that will act as the parent of the dialog box. For this, it's a good idea to pass the Access window handle using the `hWndAccessApp` function defined in Access. Setting the parent of the dialog box ensures that the dialog box is displayed in front of the specified window.

```
' Set the size of the type and the
' window handle of the parent
tdc.cbSize = Len(tdc)
tdc.hwndParent = hWndAccessApp()
```

Set the members of the type that contain the text you set earlier using the `StrPtr` function.

```
' Set the title, main instruction and content
tdc.pszWindowTitle = StrPtr(strTitle)
tdc.pszMainInstruction = StrPtr(strMainInstruction)
tdc.pszContent = StrPtr(strContent)
```

Chapter 2: Extending Applications Using the Windows API

```
' set expanded information  
tdc.pszExpandedInformation = StrPtr(strExpandedInformation)  
tdc.pszCollapsedControlText = StrPtr(strCollapsedControlText)  
tdc.pszExpandedControlText = StrPtr(strExpandedControlText)
```

Last, call the API and return the user's choice of buttons:

```
' Call the API  
hr = TaskDialogIndirect(tdc, pButton, 0, 0)  
  
' make sure we created the dialog  
If (hr <> 0) Then  
    MsgBox "Unable to create task dialog", vbCritical  
    Exit Function  
End If  
  
' return  
TaskDialogWithExpandedSection = pButton  
End Function
```

When you run this function, you should see a task dialog that is collapsed, as shown in Figure 2-10.

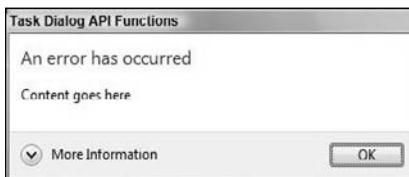


Figure 2-10

Click the expand button that says More Information. You should now see an expanded task dialog, as shown in Figure 2-11.



Figure 2-11

Adding a Footer Section

Task dialogs also enable you to add a footer section at the bottom of the dialog box. The footer section is useful for adding text such as a copyright notice or your company name. The code is similar to previous

Part I: Programming Access Applications

examples so we won't include the example here. To set the footer, use the `pszFooter` member of the `TASKDIALOGCONFIG` type. Figure 2-12 shows a task dialog with a footer section.



Figure 2-12

Setting Custom Button Text

The Windows team at Microsoft designed task dialogs for ease-of-use. Therefore, you can customize the button text to really make a dialog box understandable by your users. This is done using two members of the `TASKDIALOGCONFIG` type: `cButtons` and `pButtons` — and another user-defined type called `TASKDIALOG_BUTTON`, which is defined as follows:

```
Public Type TASKDIALOG_BUTTON
    nButtonID      As Long
    pszButtonText As Long
End Type
```

The `nButtonID` member in the type is used to give your custom button a value. This can be any `Long` integer value. You can even map this value to an `Enum` value such as `vbYes` or `TDF_YES_BUTTON` as defined in the `TASKDIALOG_COMMON_BUTTON_FLAGS` Enum that we have been working with. As you've probably guessed by now, the argument `pszButtonText` is a pointer to a string that is displayed as the button text.

There are two members of the `TASKDIALOGCONFIG` user-defined type you use to set custom text for buttons. The `pButtons` member is a pointer to an array of `TASKDIALOG_BUTTON` types. The `cButtons` member tells the API function how many elements are in the array.

For this example, let's say that you want to prompt the user to delete an employee record from a form. You can use a task dialog with custom button text to be very clear about what will happen. Let's take a look. We'll also add some expanded text to show information about the record that will be deleted. Start with the declaration of the function and the title text you used earlier. Notice that there is an array of `TASKDIALOG_BUTTON` type called `myButtons`. Arrays in VBA in Access are zero-based by default so the array contains two elements.

```
Public Function TaskDialogWithCustomButtons() As TASKDIALOG_COMMON_BUTTON_FLAGS
    Dim hr As Long
    Dim pButton As Long
    Dim tdc As TASKDIALOGCONFIG
    Dim myButtons(1) As TASKDIALOG_BUTTON

    ' Use the AppTitle if it is defined
    On Error Resume Next
    strTitle = CurrentDb().Properties("AppTitle")
    On Error GoTo 0
```

Chapter 2: Extending Applications Using the Windows API

Next, add the string arguments to display in the dialog box:

```
' Set the text arguments
strMainInstruction = "Delete record"
strContent = "Would you like to delete the current employee?"
strExpandedControlText = "Less Details"
strCollapsedControlText = "More Details"
strExpandedInformation = "Employee details:" & vbCrLf & vbCrLf & _
    "      EmployeeID: 123" & vbCrLf & _
    "      Employee Name: Cooper, Rob"
```

Create the custom buttons. Remember to use the `StrPtr` function to get the pointer to the string. The button text is very clear that a record will be deleted.

```
' add the buttons
myButtons(0).nButtonID = TDCBF_YES_BUTTON
myButtons(0).pszButtonText = StrPtr("Yes, delete")
myButtons(1).nButtonID = TDCBF_NO_BUTTON
myButtons(1).pszButtonText = StrPtr("No, don't delete")
```

Now, set the size and window handle.

```
' Set the size of the type and the
' window handle of the parent
tdc.cbSize = Len(tdc)
tdc.hwndParent = hWndAccessApp()
```

Set the text arguments of the `TASKDIALOGCONFIG` type and an icon. The icon used here will display a red background to alert the user. You can read more about using icons in task dialogs later in this chapter.

```
' Set the text arguments and icon
tdc.pszWindowTitle = StrPtr(strTitle)
tdc.pszMainInstruction = StrPtr(strMainInstruction)
tdc.pszContent = StrPtr(strContent)
tdc.pszCollapsedControlText = StrPtr(strCollapsedControlText)
tdc.pszExpandedControlText = StrPtr(strExpandedControlText)
tdc.pszExpandedInformation = StrPtr(strExpandedInformation)
tdc.pszMainIcon = TD_SECURITY_ERROR_ICON
```

To tell the API to use your custom buttons, set the `cButtons` and `pButtons` members of the `TASKDIALOGCONFIG` type. The `cButtons` member is the number of buttons in the array, or two. When you pass an array in C or C++, you are really passing a pointer to the first element in the array. To do the same in VBA, you can use the `VarPtr` function to retrieve the pointer as shown in the following code. You'll have to reference the first element of the array explicitly by index, as shown.

```
' set the custom buttons
tdc.cButtons = 2
tdc.pButtons = VarPtr(myButtons(0))
```

Great! All that's left is to call the `TaskDialogIndirect` API function and return.

```
' Call the API
hr = TaskDialogIndirect(tdc, pButton, 0, 0)
```

Part I: Programming Access Applications

```
' make sure we created the dialog
If (hr <> 0) Then
    MsgBox "Unable to create task dialog", vbCritical
    Exit Function
End If

' return
TaskDialogWithCustomButtons = pButton

End Function
```

Figure 2-13 shows the dialog box when it is expanded. Notice the custom button text and the icon used in the main instruction section of the dialog box.

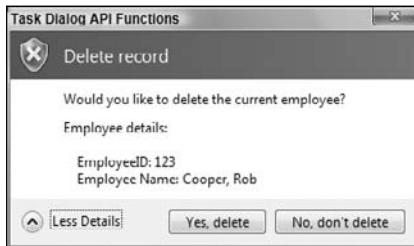


Figure 2-13

Using Command Links

You can also display the buttons in your task dialogs as *command links*. A command link is a large button in the middle of the task dialog. These buttons can display additional text to provide even more information to the user about what is happening. To tell the task dialog to use command links, set the dwFlags member of the TASKDIALOGCONFIG type to TDF_USE_COMMAND_LINKS. This constant is defined as follows:

```
Public Const TDF_USE_COMMAND_LINKS As Long = &H10
```

The following code modifies the previous example to use command links. Also note that you can display additional text in a button by adding a carriage-return.

```
Public Function TaskDialogWithCommandLinks() As TASKDIALOG_COMMON_BUTTON_FLAGS
    Dim hr As Long
    Dim pButton As Long
    Dim tdc As TASKDIALOGCONFIG
    Dim myButtons(1) As TASKDIALOG_BUTTON

    ' Use the AppTitle if it is defined
    On Error Resume Next
    strTitle = CurrentDb().Properties("AppTitle")
    On Error GoTo 0
```

Chapter 2: Extending Applications Using the Windows API

```
' Set the text arguments
strMainInstruction = "Delete record"
strContent = "Would you like to delete the current employee?"
strExpandedControlText = "Less Details"
strCollapsedControlText = "More Details"
strExpandedInformation = "Employee details:" & vbCrLf & vbCrLf & _
    " EmployeeID: 123" & vbCrLf & _
    " Employee Name: Cooper, Rob"

' add the buttons
myButtons(0).nButtonID = TDCBF_YES_BUTTON
myButtons(0).pszButtonText = StrPtr("Yes, delete." & vbCrLf & _
    "This will permanently delete the selected employee.")
myButtons(1).nButtonID = TDCBF_NO_BUTTON
myButtons(1).pszButtonText = StrPtr("No, don't delete.")

' Set the size of the type and the
' window handle of the parent
tdc.cbSize = Len(tdc)
tdc.hwndParent = hWndAccessApp()

' Set the text arguments and icon
tdc.pszWindowTitle = StrPtr(strTitle)
tdc.pszMainInstruction = StrPtr(strMainInstruction)
tdc.pszContent = StrPtr(strContent)
tdc.pszCollapsedControlText = StrPtr(strCollapsedControlText)
tdc.pszExpandedControlText = StrPtr(strExpandedControlText)
tdc.pszExpandedInformation = StrPtr(strExpandedInformation)
tdc.pszMainIcon = TD_SECURITY_ERROR_ICON

' set the custom buttons
tdc.cButtons = 2
tdc.pButtons = VarPtr(myButtons(0))

' use command links
tdc.dwFlags = TDF_USE_COMMAND_LINKS

' Call the API
hr = TaskDialogIndirect(tdc, pButton, 0, 0)

' make sure we created the dialog
If (hr <> 0) Then
    MsgBox "Unable to create task dialog", vbCritical
    Exit Function
End If

' return
TaskDialogWithCommandLinks = pButton

End Function
```

Figure 2-14 shows the modified task dialog that contains command links.

Part I: Programming Access Applications

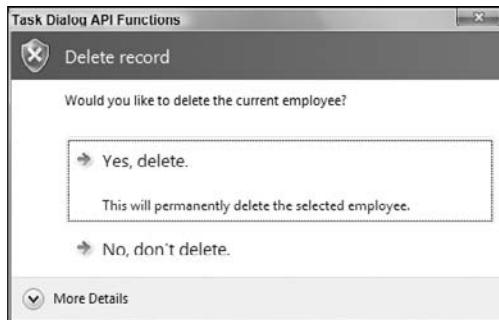


Figure 2-14

Adding Verification Text

Using verification text in a task dialog adds a check box to the bottom of the dialog box. To add verification text, set the `pszVerificationText` argument of the `TASKDIALOGCONFIG` type. Up until now, you have been calling the `TaskDialogIndirect` API function with a zero as the last argument — `pfVerificationFlagChecked`. To determine whether the check box is checked, pass a Boolean variable to the function and the API returns `True` or `False` to your variable, as shown in the following code:

```
Public Function TaskDialogWithVerificationText(fChecked As Boolean) As _
TASKDIALOG_COMMON_BUTTON_FLAGS
    Dim hr As Long
    Dim pButton As Long
    Dim tdc As TASKDIALOGCONFIG
    Dim myButtons(1) As TASKDIALOG_BUTTON

    ' Use the AppTitle if it is defined
    On Error Resume Next
    strTitle = CurrentDb().Properties("AppTitle")
    On Error GoTo 0

    ' Set the text arguments
    strMainInstruction = "Delete record"
    strContent = "Would you like to delete the current employee?"
    strExpandedControlText = "Less Details"
    strCollapsedControlText = "More Details"
    strExpandedInformation = "Employee details:" & vbCrLf & vbCrLf & _
        " EmployeeID: 123" & vbCrLf & _
        " Employee Name: Cooper, Rob"

    ' add the buttons
    myButtons(0).nButtonID = TDCBF_YES_BUTTON
    myButtons(0).pszButtonText = StrPtr("Yes, delete." & vbCrLf & _
        "This will permanently delete the selected employee.")
    myButtons(1).nButtonID = TDCBF_NO_BUTTON
    myButtons(1).pszButtonText = StrPtr("No, don't delete.")

    ' Set the size of the type and the
    ' window handle of the parent
    tdc.cbSize = Len(tdc)
```

Chapter 2: Extending Applications Using the Windows API

```
tdc.hwndParent = hWndAccessApp()

' Set the text arguments and icon
tdc.pszWindowTitle = StrPtr(strTitle)
tdc.pszMainInstruction = StrPtr(strMainInstruction)
tdc.pszContent = StrPtr(strContent)
tdc.pszCollapsedControlText = StrPtr(strCollapsedControlText)
tdc.pszExpandedControlText = StrPtr(strExpandedControlText)
tdc.pszExpandedInformation = StrPtr(strExpandedInformation)
tdc.pszMainIcon = TD_SECURITY_ERROR_ICON

' set the custom buttons
tdc.cButtons = 2
tdc.pButtons = VarPtr(myButtons(0))

' use command links
tdc.dwFlags = TDF_USE_COMMAND_LINKS

' Add the verification text
tdc.pszVerificationText = StrPtr("Don't show this dialog again")

' Call the API and pass in the boolean argument
hr = TaskDialogIndirect(tdc, pButton, 0, fChecked)

' make sure we created the dialog
If (hr <> 0) Then
    MsgBox "Unable to create task dialog", vbCritical
    Exit Function
End If

' return
TaskDialogWithVerificationText = pButton

End Function
```

This code modifies the previous example to add the verification text, as shown in Figure 2-15. The function also accepts a Boolean argument called `fChecked` that is then passed to the `TaskDialogIndirect` API function.



Figure 2-15

Part I: Programming Access Applications

You can use the following test routine to verify the value of the check box in the verification text.

```
Sub TDVerificationTextTest()
    Dim result As TASKDIALOG_COMMON_BUTTON_FLAGS
    Dim fChecked As Boolean

    result = TaskDialogWithVerificationText(fChecked)
    MsgBox fChecked
End Sub
```

Adding Radio Buttons

To give the user more choices than you might be willing to provide buttons for, you can use radio buttons. The technique for using radio buttons is a combination of the mechanism for adding custom button text and for checking the value of the verification check box.

Let's say that you have a routine in your database to export a table to a specified format. You might use a task dialog to display the choices to the user instead of a form. Start with the familiar code for setting up the task dialog. This time you define four buttons.

```
Public Function TaskDialogWithRadioButtons(lngButton As Long) As _
TASKDIALOG_COMMON_BUTTON_FLAGS
    Dim hr As Long
    Dim pButton As Long
    Dim tdc As TASKDIALOGCONFIG
    Dim myButtons(3) As TASKDIALOG_BUTTON

    ' Use the AppTitle if it is defined
    On Error Resume Next
    strTitle = CurrentDb().Properties("AppTitle")
    On Error GoTo 0

    ' Set the text arguments
    strMainInstruction = "Export table"
    strContent = "Select the export format for the selected table."
```

Next, define the radio buttons. Radio buttons are also defined using an array of type `TASKDIALOG_BUTTON`. For display purposes, we'll use some of the named format constants that are defined in Access. It so happens that these constants are defined as strings that you can use with the `DoCmd.OutputTo` method.

```
' add the radio buttons
myButtons(0).nButtonID = 0
myButtons(0).pszButtonText = StrPtr(acFormatPDF)
myButtons(1).nButtonID = 1
myButtons(1).pszButtonText = StrPtr(acFormatXPS)
myButtons(2).nButtonID = 2
myButtons(2).pszButtonText = StrPtr(acFormatTXT)
myButtons(3).nButtonID = 3
myButtons(3).pszButtonText = StrPtr(acFormatRTF)
```

Chapter 2: Extending Applications Using the Windows API

Now, set members of the `TASKDIALOGCONFIG` type:

```
' Set the size of the type and the
' window handle of the parent
tdc.cbSize = Len(tdc)
tdc.hwndParent = hWndAccessApp()

' Set the text arguments and icon
tdc.pszWindowTitle = StrPtr(strTitle)
tdc.pszMainInstruction = StrPtr(strMainInstruction)
tdc.pszContent = StrPtr(strContent)
tdc.pszMainIcon = TD_INFORMATION_ICON
```

Create the radio buttons using the `pRadioButtons` and `cRadioButtons` members of the `TASKDIALOGCONFIG` type. We use the `UBound` function to determine the number of items in the array this time.

```
' set the radio buttons
tdc.cRadioButtons = UBound(myButtons) + 1
tdc.pRadioButtons = VarPtr(myButtons(0))
```

Last, call the API and return.

```
' Call the API
hr = TaskDialogIndirect(tdc, pButton, lngButton, False)

' make sure we created the dialog
If (hr <> 0) Then
    MsgBox "Unable to create task dialog", vbCritical
    Exit Function
End If

' return
TaskDialogWithRadioButtons = pButton

End Function
```

To determine which radio button is selected, you need to pass a `Long` integer value to the API function as shown in the following test routine. The value returned from the API is the value of the radio button that you defined in the `nButtonID` member of the `TASKDIALOG_BUTTON` type.

```
Sub TDRadioButtonText()
    Dim result As TASKDIALOG_COMMON_BUTTON_FLAGS
    Dim lngButton As Long

    result = TaskDialogWithRadioButtons(lngButton)
    MsgBox lngButton
End Sub
```

The four radio buttons in the task dialog are shown in Figure 2-16.

Part I: Programming Access Applications

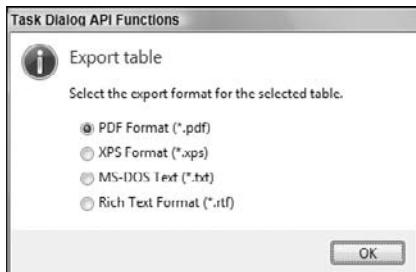


Figure 2-16

Adding Hyperlinks

The last item of functionality we'll talk about is adding hyperlinks. To tell the TaskDialogIndirect API function to use hyperlinks, set the dwFlags member of the TASKDIALOGCONFIG type to TDF_ENABLE_HYPERLINKS. This constant is defined as:

```
Public Const TDF_ENABLE_HYPERLINKS As Long = &H1
```

To define the hyperlink, use the HTML anchor tag, as shown in the following example:

```
Footer: Click <a href="http://www.wrox.com">here</a> to visit the web site.
```

You can display hyperlinks as part of the content, expanded, or footer sections of the task dialog. When you set the TDF_ENABLE_HYPERLINKS flag, hyperlink text is displayed as a hyperlink in the task dialog, as shown in Figure 2-17.

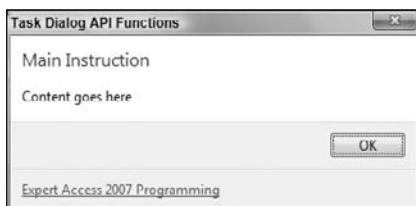


Figure 2-17

At this point, you may have tried to create a hyperlink and subsequently click it only to find that nothing happens! We should have mentioned — the task dialogs themselves do not execute hyperlinks. To enable the hyperlink, you have to write a *callback* function. A callback function is a function that you supply that Windows will call when asked. In this case, you want Windows to call your function when the user clicks on the hyperlink in the task dialog.

The callback function is not a function you define with a Declare statement. It is an actual routine that you write in VBA.

Chapter 2: Extending Applications Using the Windows API

Callback functions are often defined with a specific function signature. The function signature for task dialog callbacks is defined as follows in C++.

```
HRESULT TaskDialogCallbackProc(
    HWND hwnd,
    UINT uNotification,
    WPARAM wParam,
    LPARAM lParam,
    LONG_PTR dwRefData
);
```

The corresponding signature for this function in VBA is as follows. Of course you can name the function anything you would like.

```
Public Function TaskDialogCallback( _
    ByVal hWnd As Long, _
    ByVal uNotification As Long, _
    ByVal wParam As Long, _
    ByVal lParam As Long, _
    ByVal dwRefData As Long) As Long
End Function
```

The **TASKDIALOG_NOTIFICATIONS** Enum is defined as follows. To take an action when the hyperlink is clicked, you need to respond to the **TDN_HYPERLINK_CLICKED** notification.

```
Public Enum TASKDIALOG_NOTIFICATIONS
    TDN_CREATED = 0
    TDN_NAVIGATED = 1
    TDN_BUTTON_CLICKED = 2
    TDN_HYPERLINK_CLICKED = 3
    TDN_TIMER = 4
    TDN_DESTROYED = 5
    TDN_RADIO_BUTTON_CLICKED = 6
    TDN_DIALOG_CONSTRUCTED = 7
    TDN_VERIFICATION_CLICKED = 8
    TDN_HELP = 9
    TDN_EXPANDO_BUTTON_CLICKED = 10
End Enum
```

Before you write your callback function, you need to do one more thing. When Windows sends you the **TDN_HYPERLINK_CLICKED** notification, it gives you the hyperlink address in a pointer to a string. Thus, you have to retrieve the string from memory as you did earlier in this chapter in the section “Retrieve the Command Line.” Add the following declarations to the top of the module:

```
Private Declare Function CopyMemory Lib "kernel32.dll" Alias "RtlMoveMemory" ( _
    ByVal Destination As Any, _
    ByVal pvSource As Any, _
    ByVal Length As Long) As Long

Private Declare Function lstrlenW Lib "kernel32.dll" (lpString As Any) As Long
```

Part I: Programming Access Applications

Okay — you're ready to write the callback function. Start with the declaration:

```
Public Function TaskDialogCallback( _
    ByVal hWnd As Long, _
    ByVal uNotification As Long, _
    ByVal wParam As Long, _
    ByVal lParam As Long, _
    ByVal dwRefData As Long) As Long

    Dim stLink As String
    Dim n As Long
```

Next, filter the `uNotification` argument for the `TDN_HYPERLINK_CLICKED` notification and get the hyperlink string from memory. Remember that the task dialog API functions use Unicode strings so you need to use the `StrPtr` function with the `vbFromUnicode` argument.

```
If (uNotification = TDN_HYPERLINK_CLICKED) Then
    ' lParam contains a pointer to a string for the link
    ' find the length of the hyperlink
    n = (lstrlenW(ByVal lParam) * 2) - 1
    stLink = Space(n)

    ' copy it out of memory
    CopyMemory stLink, ByVal lParam, n
    stLink = StrConv(stLink, vbFromUnicode)
```

Last, take the link and pass it to the `FollowHyperlink` method in Access. We've wrapped it in a simple error handler in case there are any problems.

```
On Error Resume Next
FollowHyperlink stLink
If (Err) Then
    MsgBox "Cannot navigate to the link"
End If
On Error GoTo 0
End If
End Function
```

So far you've written the callback function that Windows will call when the user clicks on a hyperlink in a task dialog, but you need to tell the API how to call it. Thus, you need to give the API the address of the function, not the function itself because you don't want the function to run until needed. The address of the callback function is stored in the `pfCallback` member of the `TASKDIALOGCONFIG` structure.

To get the address of the function, you use the `AddressOf` operator in VBA, however you cannot use this operator to pass the address of a function to a user-defined type! So now what? Well, there is a workaround. The workaround is to create a wrapper routine to accept the address of the function. This routine can be used as the value of a member in a user-defined type. Define the wrapper as follows:

```
' Returns the address of a function
Public Function GetPF(lngAddress As Long) As Long
    GetPF = lngAddress
End Function
```

Chapter 2: Extending Applications Using the Windows API

Whew — that was close! Okay, now we can call the API function. Again, start with the familiar declarations.

```
Public Function TaskDialogWithHyperlink() As TASKDIALOG_COMMON_BUTTON_FLAGS
    Dim hr As Long
    Dim pButton As Long
    Dim tdc As TASKDIALOGCONFIG

    ' Use the AppTitle if it is defined
    On Error Resume Next
    strTitle = CurrentDb().Properties("AppTitle")
    On Error GoTo 0
```

Set the string arguments. Notice how the hyperlink is embedded into the footer text using the HTML anchor tag.

```
' Set the main instruction, content, and footer
strMainInstruction = "Main Instruction"
strContent = "Content goes here"
strFooter = "<a href=""http://www.wrox.com"">" & _
    "Expert Access 2007 Programming</a>"
```

Set the common members of the TASKDIALOGCONFIG type:

```
' Set the size of the type and the
' window handle of the parent
tdc.cbSize = Len(tdc)
tdc.hwndParent = hWndAccessApp()

' Set the title, main instruction, content, and footer
tdc.pszWindowTitle = StrPtr(strTitle)
tdc.pszMainInstruction = StrPtr(strMainInstruction)
tdc.pszContent = StrPtr(strContent)
tdc.pszFooter = StrPtr(strFooter)
```

Now, enable hyperlinks using the TDF_ENABLE_HYPERLINKS flag. We've also set the TDF_ALLOW_DIALOG_CANCELLATION flag that enables you to use the Escape key to cancel the dialog box, regardless of whether there is a cancel button in the dialog box. To specify the callback function, set the pfCallback member using the GetPF wrapper function. Use the AddressOf operator as the argument to the wrapper function to specify the address of the TaskDialogCallback function you wrote earlier.

```
' enable hyperlinks
tdc.dwFlags = TDF_ENABLE_HYPERLINKS Or TDF_ALLOW_DIALOG_CANCELLATION
tdc(pfCallback = GetPF(AddressOf TaskDialogCallback)
```

Nice job! You know the rest.

```
' Call the API
hr = TaskDialogIndirect(tdc, pButton, 0, 0)

' make sure we created the dialog
If (hr <> 0) Then
    MsgBox "Unable to create task dialog", vbCritical
```

Part I: Programming Access Applications

```
    Exit Function
End If

' return
TaskDialogWithHyperlink = pButton

End Function
```

When you run this code and click the hyperlink, a browser window should open to the Wrox Web site.

Icons

Earlier you used the `pszMainIcon` member of the `TASKDIALOGCONFIG` type to display a predefined icon in the task dialog. You may have also noticed that the `TaskDialog` API function defines an argument called `pszIcon`. The predefined icons were defined earlier in the `TASKDIALOG_SYSTEM_ICONS` Enum.

You can use one of these values to pass as the `pszIcon` argument to the `TaskDialog` API. Edit the `SimpleTaskDialog` function you wrote earlier to accept an icon as follows:

```
Public Function SimpleTaskDialog( _
    Optional Title As String, _
    Optional MainInstruction As String, _
    Optional Content As String, _
    Optional Buttons As TASKDIALOG_COMMON_BUTTON_FLAGS, _
    Optional Icon As TASKDIALOG_SYSTEM_ICONS _
    As TASKDIALOG_COMMON_BUTTON_FLAGS

    Dim hr As Long
    Dim pButton As Long

    ' Use the AppTitle if it is defined
    On Error Resume Next
    Title = CurrentDb().Properties("AppTitle")
    On Error GoTo 0

    ' Call the API
    hr = TaskDialog(hWndAccessApp(), 0, _
        StrPtr>Title), _
        StrPtr>MainInstruction), _
        StrPtr>Content), _
        Buttons, _
        Icon, _
        pButton)
    ' make sure we created the dialog
    If (hr <> 0) Then
        MsgBox "Unable to create task dialog", vbCritical
        Exit Function
    End If

    SimpleTaskDialog = pButton
End Function
```

Chapter 2: Extending Applications Using the Windows API

We already looked at the TD_SECURITY_ERROR_ICON value, but let's take a look at some of the icon values in action starting with TD_ERROR_ICON displayed in Figure 2-18. This uses the same red "X" icon as vbCritical for the MsgBox function.



Figure 2-18

To display a task dialog with a green background and check mark, use TD_SECURITY_SUCCESS_ICON, as shown in Figure 2-19.



Figure 2-19

To display a task dialog with a gold background and exclamation mark, use TD_SECURITY_WARNING_ICON, as shown in Figure 2-20.



Figure 2-20

These icons provided by Windows Vista are used throughout the system to provide context to task dialogs and can easily be used in your applications.

Miscellaneous API Functions

You can use the Windows API for many tasks that are not available natively in VBA or Access. Virtually everything that can be done in Windows is exposed using an API function. The following sections list a few API functions that we use regularly in Access applications.

Part I: Programming Access Applications

Creating a GUID

A GUID, or globally unique identifier, is a 128-bit number that is typically guaranteed to be unique across any number of machines. These numbers are used frequently in the Component Object Model, or COM, programming paradigm. As such, they appear in several places in Access and Windows, such as in the CLSID for a reference or as the value for the `ReplicationID` data type in a table. Because they are guaranteed to be unique, they are really handy for things such as generating file names, or for an ID value for a class instance.

GUIDs are commonly displayed in the following format:

```
{4AFFC9A0-5F99-101B-AF4E-00AA003F0F07}
```

Because GUIDs are used commonly in COM, the Windows API function used to create a GUID, `CoCreateGuid` is defined in `ole32.dll`, which is the DLL that provides the majority of the functionality for COM itself.

```
Private Declare Function CoCreateGuid Lib "ole32.dll" (pGuid As GUID) As Long
```

This function fills in a user-defined Type called `GUID`:

```
Public Type GUID
    Data1    As Long
    Data2    As Integer
    Data3    As Integer
    Data4(7) As Byte
End Type
```

In order to create a GUID in the correct format, we typically use a helper function to format the `GUID` Type as it comes back from the API. Start with the declarations:

```
Private Function FormatGuid(myGuid As GUID) As String
    Dim i As Integer
```

Next, format the `Data1` section. The number that comes back for each part of the API should be formatted in hexadecimal using the `Hex` function in VBA. Use the `Format` function in VBA to force each portion of the string to a certain length. Note that we include the curly brace as a literal character in the format:

```
FormatGuid = Format(Hex(myGuid.Data1), "{00000000-}")           ' Data1
```

Now, `Data2`, `Data3`, and the first two bytes of the `Byte` array in `Data4`:

```
FormatGuid = FormatGuid & Format(Hex(myGuid.Data2), "@@@@-")      ' Data2
FormatGuid = FormatGuid & Format(Hex(myGuid.Data3), "@@@@-")      ' Data3
FormatGuid = FormatGuid & Format(Hex(myGuid.Data4(0)), "@@")        ' Data4(0)
FormatGuid = FormatGuid & Format(Hex(myGuid.Data4(1)), "@@-")       ' Data4(1)
```

Next, bytes 2 through 6 of the `Byte` array in `Data4`:

```
For i = 2 To 6
    FormatGuid = FormatGuid & Format(Hex(myGuid.Data4(i)), "@@") ' Data4(2-6)
Next
```

Chapter 2: Extending Applications Using the Windows API

And then the last byte of Data4:

```
FormatGuid = FormatGuid & Format(Hex(myGuid.Data4(i)), "@@")      ' Data4(7)
```

Finally, you may get spaces when you use the @ symbol in the Format function. Replace these with zeroes so that it creates a valid number:

```
FormatGuid = Replace(FormatGuid, " ", "0")                      ' Replace spaces
End Function
```

The only thing to do at this point is to call the CoCreateGuid API function. For this, write the following code:

```
Public Function CreateGuid() As String
    ' generates a new GUID
    Dim pGuid As GUID
    Dim rc As Long

    ' call the API
    rc = CoCreateGuid(pGuid)
    If (rc = 0) Then
        ' return the guid
        CreateGuid = FormatGuid(pGuid)
    End If
End Function
```

ShellExecute Tricks

Earlier you used the ShellExecuteEx function to send the properties verb to the specified file to view the Properties dialog box for the file. As with ShellExecuteEx, the ShellExecute function is also used to perform an action on a file. The difference between the two functions is that the ShellExecuteEx function uses the SHELLEXECUTEINFO structure where ShellExecute does not.

Generally speaking, using the open verb with any file asks the application associated with the file to open it. The trick to these next two examples depends on the definition of what is considered a *file*.

In the example conversion found in the section “Writing Declare Statements” earlier in this chapter, we defined the ShellExecute API as:

```
Private Declare Function ShellExecute Lib "shell32.dll" Alias "ShellExecuteA" ( _
    ByVal hWnd As Long, _
    ByVal lpOperation As String, _
    ByVal lpFile As String, _
    ByVal lpParameters As String, _
    ByVal lpDirectory As String, _
    ByVal nShowCmd As Long) As Long
```

In these examples, we also use the SW_SHOW constant to show the resulting window in its default state. This constant is defined as:

```
Private Const SW_SHOW As Long = 5
```

Part I: Programming Access Applications

Sending E-mail Using the Default E-mail Client

E-mail clients typically register themselves as being able to process the `mailto` protocol. This protocol includes headers that you can use to set the address of the recipient, subject, body, cc, and bcc. By using the `mailto` protocol in the `lpFile` argument, you are asking the default e-mail client registered on the machine to generate an e-mail message. The code for this is as follows:

```
Sub SendEmail(strTo As String, strSubject As String, strBody As String)
    Dim rc As Long
    Dim strFile As String

    ' build the lpFile argument
    strFile = "mailto:" & strTo
    strFile = strFile & "?subject=" & strSubject
    strFile = strFile & "&body=" & strBody

    rc = ShellExecute(0, "open", strFile, "", "", SW_SHOW)

End Sub
```

Opening the Default Web Browser

Similar to the `mailto` protocol, you can use the `http` or `https` protocols to open the default Web browser. The following code opens the default Web browser to the specified address. This routine includes a check to ensure the address starts with either `http://` or `https://`.

```
Sub OpenWebBrowser(strAddress As String)

    Dim rc As Long
    Dim strFile As String

    If (Left(strAddress, 7) <> "http://" And Left(strAddress, 8) <> "https://") Then
        strFile = "http://" & strAddress
    Else
        strFile = strAddress
    End If

    rc = ShellExecute(0, "open", strFile, "", "", SW_SHOW)

End Sub
```

Summary

The Windows API can be your gateway to applications that have the look and feel of the operating system you are running on. It can also be used to dramatically extend functionality of an application by offering complete integration with the capabilities of the system. Windows provides many useful functions that developers can call to further extend their applications, even from languages other than C or C++.

Chapter 2: Extending Applications Using the Windows API

This chapter discussed the following:

- The new and exciting possibilities provided with task dialogs on Windows Vista
- How to use the Windows API to extend your applications beyond the capabilities of Access or VBA

We continue the building blocks of programming Access applications in the next chapter with one of our favorite topics in VBA programming — class modules.

In the next chapter, we investigate using class modules to simplify programming tasks and create reusable code for multiple applications. You'll also learn how to write class modules that wrap built-in objects, such as Access forms, so that you will ultimately have write less code to write in particular scenarios.

3

Programming Class Modules

Class modules can be a powerful tool in your programming toolbox and writing them is one of our favorite programming tasks in VBA. If you have worked with classes in object-oriented languages such as C++ or C#, you might be disappointed to learn that VBA is not a true object-oriented programming language because both VB and VBA lack implementation inheritance. However, class modules in VBA provide many other strengths that can save you time when programming Access or VBA applications. With that in mind, you learn the following about class modules by the end of this chapter:

- The components that make up a class module including properties, methods, enumerations, and events
- How to reuse code that you write in class modules
- How to subclass other classes to add your own functionality or to reduce the amount of code to write
- How to write interfaces in VBA
- How to write powerful and reusable collection classes
- How to create custom events
- Design considerations for VBA class modules

Overview of Class Modules

Object-oriented programming (OOP) has been around for a while, and the mainstay of OOP is the *class*. A class defines the blueprint for an *object* where an object is merely an instance of a class. Many classes that you design can have multiple instances, or objects, but there might be times where you want to allow only one instance of a class. You can find out more about these types of classes later in this chapter in the section “Design Considerations.”

The Class Module object in VBA is used to write a class.

Part I: Programming Access Applications

Why Use Class Modules?

Many, if not all programming tasks that you do in Access can be accomplished using standard modules. Given that, why should you use class modules at all? In order to answer that question, take a look at some very good reasons why class modules are useful.

Making Complex Tasks Easier

The TaskDialog API functions, introduced in Chapter 2, are mildly complex and provide a lot of options for you as a developer. Recall that even the simple version of the function contained many different parameters.

```
Private Declare Function TaskDialog Lib "comctl32.dll" ( _  
    ByVal hwndParent As Long, _  
    ByVal hInstance As Long, _  
    ByVal pszWindowTitle As Long, _  
    ByVal pszMainInstruction As Long, _  
    ByVal pszContent As Long, _  
    ByVal dwCommonButtons As TASKDIALOG_COMMON_BUTTON_FLAGS, _  
    ByVal pszIcon As Long, _  
    pnButton As Long) As Long
```

Now imagine that the TaskDialog functions were wrapped in a class called, appropriately enough, TaskDialog. By using a class module, you can define all of these arguments as properties of the class and get or set them as you need. Each instance of the class keeps track of the individual values so that displaying a task dialog is as easy as setting some properties and calling a method. You'll see the TaskDialog API functions implemented as a class module later in this chapter in the section "Class Module Members."

Code Reuse

Perhaps one of the best reasons for using class modules is the ability to reuse code. Writing classes enables you to create your own libraries that can be reused in multiple applications. This has several distinct benefits, as follows:

- ❑ Common code has less chance of breaking because the code is tested, well documented, and understood.
- ❑ Any changes that need to be made can be made in one place.
- ❑ By reusing code, you end up writing less code, which allows you to focus on the business problems that you are trying to solve.

Controlling Visibility

Public subroutines and functions in standard modules are available throughout your application. Private routines in standard modules are available only to other routines in the module. What if you wanted to control visibility without making a routine `Private`? The answer is to use a class module. Because members of a class module are available only when you have an instance of the class, you cannot call methods directly from other routines.

Given that calling code in classes tends to be easier, you might even write classes that are designed to be used by other developers. In such cases, you may want to explicitly hide private data or other helper members of a class. Class modules also provide this capability. In object-oriented programming (OOP) terms, this is known as *encapsulation*.

Class Module Members

The routines you write in a class module are known as members. Class modules can have several different types of members: properties, methods, enumerations, and events. The following sections examine each type of member using the TaskDialog API functions described in Chapter 2.

Enumerations

It might seem a bit odd to talk about enumerations first when discussing class modules; however we're going to use them in later discussions so we'll get them out of the way here.

An enumeration is a group of constant Long integer values. Because their values are grouped together, enumerations provide additional meaning to a set of individual constants. Unless otherwise specified, values in an `Enum` begin at zero and are numbered sequentially.

The constants listed in the `TASKDIALOG_SYSTEM_ICONS` enumeration shown here are icons provided by the system for task dialogs:

```
Public Enum TASKDIALOG_SYSTEM_ICONS
    TD_SECURITY_SHIELD_GRAY_ICON = 65527
    TD_SECURITY_SUCCESS_ICON = 65528
    TD_SECURITY_ERROR_ICON = 65529
    TD_SECURITY_WARNING_ICON = 65530
    TD_SECURITY_SHIELD_BLUE_ICON = 65531
    TD_SECURITY_SHIELD_ICON = 65532
    TD_INFORMATION_ICON = 65533
    TD_ERROR_ICON = 65534
    TD_WARNING_ICON = 65535
    TD_QUESTION_ICON = 32514
    IDI_APPLICATION = 32512
End Enum
```

Likewise, the following enumeration defines common buttons that you can use with a task dialog.

```
Public Enum TASKDIALOG_COMMON_BUTTON_FLAGS
    TDCBF_OK_BUTTON = &H1
    TDCBF_YES_BUTTON = &H2
    TDCBF_NO_BUTTON = &H4
    TDCBF_CANCEL_BUTTON = &H8
    TDCBF_RETRY_BUTTON = &H10
    TDCBF_CLOSE_BUTTON = &H20
End Enum
```

Part I: Programming Access Applications

Flags

Notice that the `TASKDIALOG_COMMON_BUTTON_FLAGS` enumeration defined a moment ago is defined as a set of *flags*. Flag values can be used in combination with each other by using the bitwise operations of `And`, `Or`, and `Not`. These operators enable you to combine enumerated values by setting individual bits.

Combining Flag Values

Use the `Or` operator to combine two or more flag values. Say you have a variable of type `TASKDIALOG_COMMON_BUTTON_FLAGS` and would like to include the Yes button, and the Cancel button. This might look something like:

```
Public Sub CombineFlagTest()
    Dim x As TASKDIALOG_COMMON_BUTTON_FLAGS

    ' combine two flags
    x = TDCBF_YES_BUTTON Or TDCBF_CANCEL_BUTTON
    Debug.Print x
End Sub
```

If you run this code, the value that should be printed is 10. Figure 3-1 shows the binary representation of the two values being combined to show how this works.

Binary Values	128	64	32	16	8	4	2	1	Or
TDCBF_YES_BUTTON = &H2	0	0	0	0	0	0	1	0	=
TDCBF_CANCEL_BUTTON = &H8	0	0	0	0	1	0	0	0	
RESULT = &HA (10)	0	0	0	0	1	0	1	0	

Figure 3-1

Removing Flag Values

Use the `And` `Not` operators to remove a flag value, as shown in the following example:

```
Public Sub RemoveFlagTest()
    Dim x As TASKDIALOG_COMMON_BUTTON_FLAGS

    ' set the flag to begin with
    x = TDCBF_YES_BUTTON Or TDCBF_CANCEL_BUTTON
    Debug.Print "Before = " & x

    ' remove the flag
    x = x And Not TDCBF_YES_BUTTON
    Debug.Print "After = " & x

End Sub
```

When you run this code, you should see the following output:

```
Before = 10
After = 8
```

Determining If a Flag Is Set

Use the `And` operator to determine if a flag value has been set. If the return value of the `And` operator returns the individual flag value you are looking for, then the flag has been set. This is demonstrated in the following example:

```
Public Sub TestFlag()
    Dim x As TASKDIALOG_COMMON_BUTTON_FLAGS

    ' set the flag to begin with
    x = TDCBF_YES_BUTTON Or TDCBF_CANCEL_BUTTON

    ' check for the Cancel button
    If ((x And TDCBF_CANCEL_BUTTON) = TDCBF_CANCEL_BUTTON) Then
        Debug.Print "Cancel button is set"
    Else
        Debug.Print "Cancel button is not set"
    End If

    ' now test for something that has not been set
    If ((x And TDCBF_NO_BUTTON) = TDCBF_NO_BUTTON) Then
        Debug.Print "No button is set"
    Else
        Debug.Print "No button is not set"
    End If

End Sub
```

This example sets the flag values to use the Yes and Cancel buttons, and then tests for the Cancel button. It then tests for the No button, which was not set. When you run this code you should see the following output:

```
Cancel button is set
No button is not set
```

Flag-Related Functions

We like to create helper functions for flags that wrap these operators for consistency and clarity. These helper functions also remove any confusion about which operator to use. The most common one we use is this next one, which wraps the `And` operator to determine if a flag has been set:

```
Public Function IsFlagSet(Flag As Long, Flags As Long) As Boolean
    IsFlagSet = ((Flags And Flag) = Flag)
End Function
```

We sometimes use helper functions to set and remove flag values as well. These routines make it very clear in the code that we're working with flag values.

```
Public Sub SetFlag(Flag As Long, Flags As Long)
    Flags = Flags Or Flag
End Sub
Public Sub RemoveFlag(Flag As Long, Flags As Long)
    Flags = Flags And Not Flag
End Sub
```

Part I: Programming Access Applications

Properties

A property is a member that describes the characteristics of the class. For example, some characteristics of a TaskDialog API are its title, main instruction, and the content. Use the `Property` procedure in VBA to create a property. There are three additional keywords that you declare with a `Property` procedure. The following table describes the keywords.

Property Procedure	Description
<code>Property Get</code>	Creates a property value that can be read.
<code>Property Let</code>	Creates a property value that can be written. Use with the scalar data types: <code>Integer</code> , <code>Long</code> , <code>String</code> , <code>Single</code> , <code>Double</code> , <code>Date</code> , <code>Currency</code> , <code>Byte</code> .
<code>Property Set</code>	Creates a property value that can be written. Use with object data types.

It helps to see these in action, so let's look at some examples. Although we implemented the characteristics of a task dialog using `Optional` arguments to our function in Chapter 2, the following code uses properties and a class module to model this type of dialog box.

For starters, create a new class module and name it `TaskDialog`. While we tend to be sticklers for using naming conventions in Access, class modules are one area where we often stray. Many people use the `cls` prefix when naming a class, but we prefer to name classes without a prefix. Objects feel more natural without the prefix, and we think they look nicer when they appear in IntelliSense. We do stick to a convention of beginning the name of a class with a capital letter.

Next, add the `Declare` statement that you added in Chapter 2. `Declare` statements in a class module must be declared as `Private`. Also include the `TASKDIALOG_COMMON_BUTTON_FLAGS` and `TASKDIALOG_SYSTEM_ICONS` enumerations from earlier:

```
' -- Declares --
Private Declare Function TaskDialog Lib "comctl32.dll" ( _
    ByVal hwndParent As Long, _
    ByVal hInstance As Long, _
    ByVal pszWindowTitle As Long, _
    ByVal pszMainInstruction As Long, _
    ByVal pszContent As Long, _
    ByVal dwCommonButtons As TASKDIALOG_COMMON_BUTTON_FLAGS, _
    ByVal pszIcon As Long, _
    pnButton As Long) As Long
```

When you write properties you must use `Private` variables in the class to store the data. Setting property values without using private data causes the `Property Let` routine to call itself repeatedly, which leads to an “out of stack space” error.

Add the `Private` variables that store the values for the class:

```
Private m_strTitle           As String
Private m_strMainInstruction As String
```

```
Private m_strContent          As String
Private m_pButtonClicked     As Long
Private m_MainIcon           As TASKDIALOG_SYSTEM_ICONS
```

Only the variables for the title, main instruction, content, and main icon are shown. The complete TaskDialog class is implemented in the corresponding download for this chapter in the file titled “TaskDialogClass.accdb.” The implementation in the download provides a single method that determines which API function to call so it differs slightly from what is shown here.

Now, add some properties. Properties are defined using `Property` procedures, as shown in the following code. You'll implement the `Title`, `MainInstruction`, `Content`, and `MainIcon` properties, as shown:

```
Public Property Get Content() As String
    Content = m_strContent
End Property
Public Property Let Content(ByVal sContent As String)
    m_strContent = sContent
End Property
Public Property Get MainIcon() As TASKDIALOG_SYSTEM_ICONS
    MainIcon = m_MainIcon
End Property
Public Property Let MainIcon(ByVal eMainIcon As TASKDIALOG_SYSTEM_ICONS)
    m_MainIcon = eMainIcon
End Property
Public Property Get MainInstruction() As String
    MainInstruction = m_strMainInstruction
End Property
Public Property Let MainInstruction(ByVal sMainInstruction As String)
    m_strMainInstruction = sMainInstruction
End Property
Public Property Get Title() As String
    Title = m_strTitle
End Property
Public Property Let Title(ByVal sTitle As String)
    m_strTitle = sTitle
End Property
```

Read-Only Properties

Because the `Property Get` routine is used to create a property that can be read, you can create a read-only property by defining a `Property Get` procedure without a corresponding `Property Let` or `Property Set`. Constant values or a calculated value, such as the autonumber field in a table or an ID value, are good examples of read-only properties.

Write-Only Properties

To create a write-only property, use a `Property Let` or `Property Set` procedure without a corresponding `Property Get`. One of the more common examples of a write-only property is a password. Password properties commonly allow you to set their value without being able to read them for security purposes.

Part I: Programming Access Applications

Metadata Properties

Certain types of read-only properties help provide additional information about the class or an instance of the class. We call these metadata properties because they provide information about the class itself.

The first property returns an ID value that represents a single instance of the class. Such ID properties are useful in collections to know which instance of a class is being used. We implement this property using the `ObjPtr` function. This function, as mentioned in Chapter 2, returns the memory address of an `Object` variable. Because each variable is stored in its own location in memory, the `ObjPtr` function should return a unique value for each instance. Use the following code to use the `ObjPtr` function to identify an instance of a class:

```
Public Property Get ID() As Long
    ID = ObjPtr(Me)
End Property
```

As with Forms and Reports, use the `Me` keyword in your classes to return the current instance of an object.

Use the following code to return the memory address formatted as a hexadecimal string:

```
Public Property Get IDString() As String
    IDString = Replace(Format(Hex$(ObjPtr(Me)), "\0x@#####@@"), " ", "0")
End Property
```

You can also track information such as when the instance was created by using the `Initialize` event of the class. The `Initialize` event is discussed in more detail later in this chapter.

The following code stores the time that an instance of the class is created in a property called `CreatedDate`. Set the created date and time when the class is initialized so it doesn't change when you call the `CreatedDate` property later in the application.

```
Private m_created As Date
Private Sub Class_Initialize()
    m_created = Now()
End Sub
Public Property Get CreatedDate() As Date
    CreatedDate = m_created
End Property
```

Methods

A method is a member of a class that performs an action. In VBA, you implement methods using `Sub` or `Function` routines. In Chapter 2, we had a simple function that called the `TaskDialog` API and accepted the arguments to pass to the function. By implementing these arguments as properties, you can simplify this function to become a method in our `TaskDialog` class. The following code implements a method called `ShowSample` that displays the dialog box.

```
Public Function ShowSample() As Long
    Dim hr As Long

    hr = TaskDialog( _
        hWndAccessApp(), _
        0, _
```

```
StrPtr(m_strTitle), _  
StrPtr(m_strMainInstruction), _  
StrPtr(m_strContent), _  
TDCBF_YES_BUTTON, _  
m_MainIcon, _  
m_pButtonClicked)  
  
' verify the API ran  
If (hr <> 0) Then  
    MsgBox "Unable to show the task dialog", vbCritical, "TaskDialog.Show()"  
End If  
  
' return  
ShowSimple = m_pButtonClicked  
End Function
```

Remember that the Show method in the download for this chapter contains logic to determine which API to call so it differs slightly from what is shown here.

To test this function, write the following code in a new standard module:

```
Sub TaskDialogTest()  
    Dim td As New TaskDialog  
    td.Title = "My Title"  
    td.MainIcon = TD_INFORMATION_ICON  
    td.MainInstruction = "Chapter 3 Test"  
    td.Content = "Content for the task dialog goes here"  
    td.ShowSample  
End Sub
```

One of the many cool things about programming with class modules is that IntelliSense is available for your classes as well, as shown in Figure 3-2. This figure shows the members for the complete implementation of the TaskDialog class.

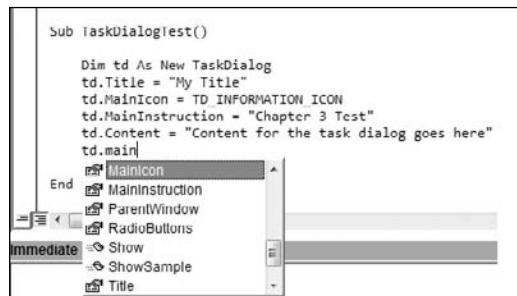


Figure 3-2

The method name and its arguments are known collectively as the method *signature*.

Part I: Programming Access Applications

Events

The Windows operating system uses events extensively to notify users and other code that something has happened. For example, when you click on a button in Windows (or on a form in Access for that matter), Windows sends the application notification that the click has occurred. This lets the application react in some manner.

As Access developers, you are probably quite familiar with events, and the `Click` event on a `CommandButton` is but one example of an event that you can handle in your application. In the Access object model, `CommandButton` is a class that defines several events, as shown in the Object Browser in Figure 3-3.

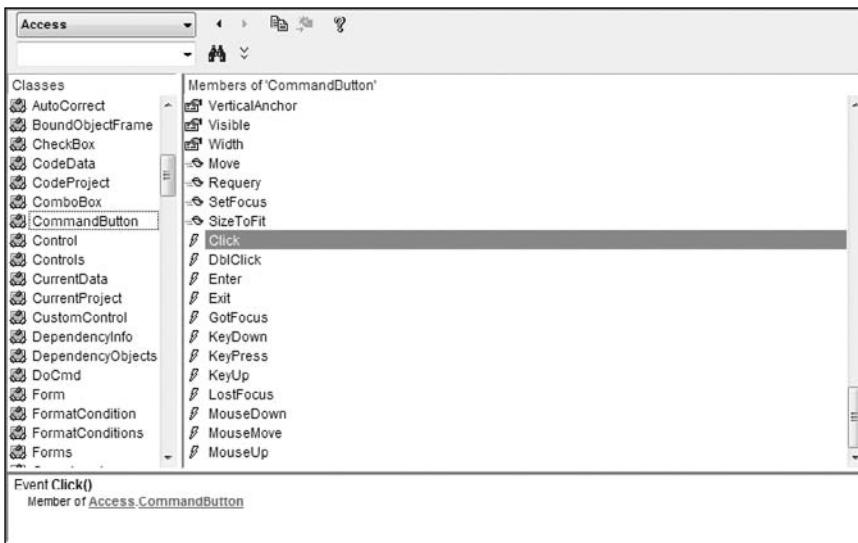


Figure 3-3

The routine you write for an event is called an event handler.

Just as classes in Access such as `CommandButton`, `Form`, and `Report` define events, you can create your own events in a class module! Events that you write are described in more detail towards the end of this chapter in the section “Events.”

Initialize Event

When creating an instance of a class, you might need to get the class into a particular state before it is used. For example, say that you are creating a class that contains methods to communicate with a SQL Server. When an instance of the class is created, it would be useful to connect to the server to maintain a single connection throughout the lifetime of the object instance. This type of code can be written using the `Initialize` event of the class, which is defined as follows:

```
Private Sub Class_Initialize()  
End Sub
```

Languages such as C# or C++ also enable you to run code when an instance of a class is created. In those languages, these methods are known as *constructors* because the code runs when an object is constructed, or created. While VBA is extremely powerful, there is one distinct limitation from these other languages. Constructors created for classes in C# or C++ enable you to pass arguments to their methods and to pass additional data to a class for initialization. Consider the example of a class that contains a SQL Server connection: You might pass the name of the server to connect to. Unfortunately, the `Initialize` event does not enable you to pass additional arguments. Using our example, as a workaround, you might write a separate initialization method that accepts arguments or maintains Boolean values or another variable to track the state of the class to optimize connections.

The `Initialize` event may be called in one of two places depending on how you create the instance. Classes can be instantiated using one of two methods:

Explicit

```
Dim obj As Class1  
Set obj = New Class1  
obj.DoSomething
```

Implicit

```
Dim obj As New Class1  
obj.DoSomething
```

In the preceding explicit example, the `Initialize` event of the class runs when the class is instantiated in the `Set` statement. In the implicit example, however, the class instance is not actually created until the first member call, in this case, on the `DoSomething` method.

Use the `Set` statement when instantiating classes for complete control over when object instances are created.

Terminate Event

Naturally, cleaning up resources is an important part of programming, and VBA is no exception. Class modules provide a `Terminate` event that enables you to cleanup any resources you used in the class. Using our SQL Server connection example, use the `Terminate` event of the class to close the connection and destroy any objects you created in the class.

The `Terminate` event fires when you destroy an object by setting it to `Nothing`, or when an object goes out of scope and is defined as follows:

```
Private Sub Class_Terminate()  
End Sub
```

Code Reuse in VBA

The ability to reuse code is a huge benefit to using class modules. You can reuse a class in one database by setting a reference to it in another database. You need, however, to do a little work beforehand in order to make this work.

Part I: Programming Access Applications

Instancing Property

The `Instancing` property of a class is used to set the level of visibility for a class. This value determines whether a class is visible externally, and how it can be used. Unlike their counterparts in Visual Basic (VB), class modules in VBA have two possible values for the `Instancing` property, as described in the following table:

Property	Value	Description
Private	1	The class is not visible outside of the VBA project where it resides.
PublicNotCreatable	2	The class is visible, but you cannot create an instance of it.

- With only these two values available, clearly there is a limitation. These properties do not enable you to create a new instance of the class, which makes reuse difficult. The following sections describe how you can work around this limitation.

Wrapper Method

The recommended approach for exposing a class outside of the database where it resides is to write a wrapper property or method in a standard module. This property or method simply exposes an instance of the class, and because it is written in a standard module, is available outside of the application.

In your database that contains the `TaskDialog` class, add a new module called `Wrappers`. In this module, add the following code to return an instance of the `TaskDialog` class.

```
Private m_dialog As TaskDialog
Public Property Get TaskDialogObject() As TaskDialog
    ' Wrapper around the TaskDialog class.
    ' This property makes the class available from outside this database.
    If (m_dialog Is Nothing) Then
        Set m_dialog = New TaskDialog
    End If

    Set TaskDialogObject = m_dialog
End Property
```

As you can see, properties can also be used in standard modules. You could also use a function here, but we tend to prefer the property semantic. This code first checks the private `m_dialog` variable. If this variable has not been set, create an instance of the class, and then return it in the property routine. This technique returns the complete instance of the `TaskDialog` class so that it can be used externally.

This compiles inside the database where the `TaskDialog` class resides, but the class is still marked `Private` through its `Instancing` property. As a result, you cannot see the `TaskDialog` class outside of this database. To make it visible, change the `Instancing` property to `PublicNotCreatable`, as shown in Figure 3-4.

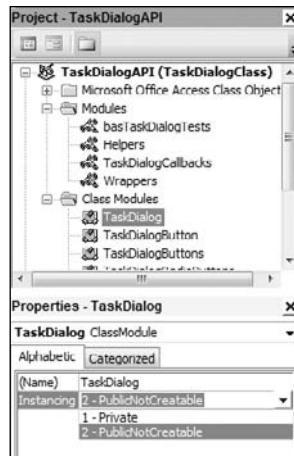


Figure 3-4

To test the class, create a new database and set a reference to the database that contains the TaskDialog class. (References are discussed in Chapter 1.) Then, create a new module and add the following test code:

```
Public Sub TaskDialogTestExternal()
    Dim td As TaskDialog
    Set td = Wrappers.TaskDialogObject

    ' set properties
    td.Title = "Dialog Title"
    td.MainInstruction = "Main Instruction"
    td.Content = "Dialog content"
    td.ShowSample
End Sub
```

When you run this code, you should have a task dialog object, as shown in Figure 3-5.

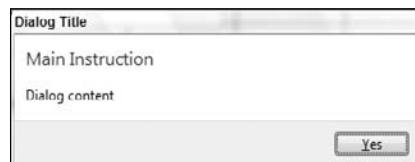


Figure 3-5

Now you can use task dialogs throughout your applications!

Instancing Property (Revisited)

Notice that we mentioned that using a wrapper method was the *recommended* approach. Here's a trick you can use to reuse classes without writing a wrapper method. VBA has its origins in VB, which has six possible values for the `Instancing` property. Of the four additional values that are available, one can be

Part I: Programming Access Applications

used with classes in VBA. By setting the `Instancing` property to 5, or `MultiUse`, you can use VBA classes in databases other than the one they were created in. To set the `Instancing` property for your class, run the following code in the `Immediate` window:

```
VBE.ActiveVBProject.VBComponents("YourClassName").Properties("Instancing") = 5
```

You can use members of the VBA Extensibility Library to set this property for all class modules in your application. The following code uses late binding so it does not require a reference to this library to run.

```
Sub SetInstancingForAll()
    Dim vbc As Object

    For Each vbc In VBE.ActiveVBProject.VBComponents
        ' set the property for all class modules
        If (vbc.Type = 2) Then
            vbc.Properties("Instancing") = 5

        ' save
        DoCmd.Save acModule, vbc.Name
    End If
    Next

    ' compile
    DoCmd.RunCommand acCmdCompileAllModules

    ' cleanup
    Set vbc = Nothing
End Sub
```

Subclassing

The real power of class modules in VBA is the ability to *subclass*. Subclassing is a process by which you encapsulate an existing class to add functionality. This is typically done to encapsulate a set of events that are exposed by the class being subclassed. Subclassing Access forms, reports, and controls enables you to write code that is common to those types of objects. As a result, you can write event handlers in the subclass and respond to them for multiple forms, reports, or controls without needing to duplicate code throughout the application.

Subclassing Access Forms and Reports

The key to subclassing a Form or Report is to use the `WithEvents` keyword in the declaration of a variable. This keyword tells VBA that you will handle any events that are exposed by the object. Types that you declare with the `WithEvents` must expose events, so by nature, you can use the `WithEvents` keyword only with classes.

The following are a few additional things to keep in mind when subclassing a form or report:

- ❑ The form or report must have a module, that is, the `HasModule` property must be set to Yes.
- ❑ Access will not fire an event unless the event property is set to [Event Procedure]

- ❑ If you are subclassing a form, you should include a `Form` property that returns the `Private` instance of the form being subclassed.
- ❑ If you are subclassing a report, you should include a `Report` property that returns the `Private` instance of the report being subclassed.

Sinking Form Events

You can respond to events for a form in your database using the `WithEvents` keyword. When you use this keyword, VBA enables you to create event handlers in your class module, just as if you were working with the object directly. In the next few sections, you create a class called `ExtendedForm` that subclasses the `Form` object to respond to events. Start by creating a new class module called `ExtendedForm`.

Before you create the subclass, however, remember that Access will not fire an event unless the event property for the event is set to `[Event Procedure]`. Add the following code to the declarations section of the class:

```
Private Const CON_EVENT_PROCEDURE As String = "[Event Procedure]"
```

Next, add the following helper routine to the class. This routine takes a `Form` object as a parameter and the name of the property to sink.

```
Private Sub AddEventHook(obj As Form, EventProperty As String)
    obj.Properties(EventProperty) = CON_EVENT_PROCEDURE
End Sub
```

With the helper method created, follow these steps to create a subclass of the Access `Form` object:

1. Add the following line of code in the declarations section of the class:

```
Private WithEvents m_Form As Access.Form
```

2. Add a `Property Get` routine called `Form` that returns the `m_Form` variable:

```
Public Property Get Form() As Form
    Set Form = m_Form
End Property
```

3. Add a `Property Set` routine called `Form` that sets the `m_Form` variable. This routine is called for each individual form that you subclass and is the property that actually performs the subclassing. If you know which events you want to sink, you can sink them when the form is subclassed. For our examples, we subclass a few events, so we call the `AddEventHook` method from this property.

```
Public Property Set Form(objForm As Form)
    Set m_Form = objForm
    AddEventHook m_Form, "OnError"
End Property
```

Part I: Programming Access Applications

Subclassing the Form_Error event

The Error event of a form is used to display data related errors for issues such as:

- Duplicated primary key
- Write conflict errors
- Validation rule errors

Oftentimes, code in an Error event is duplicated throughout an application to provide a uniform experience for handling different types of errors. By subclassing the Form object, you can handle the Error event once and respond to it for any form that throws an error in the Error event.

If you view the list of objects in the drop-down on the left in the ExtendedForm class, you should see the m_Form variable listed, as shown in Figure 3-6.

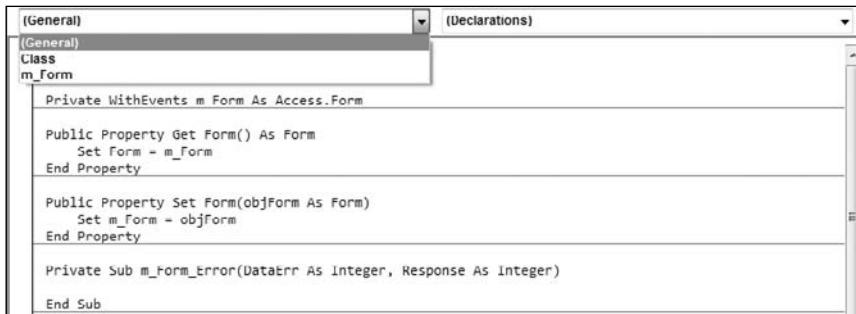


Figure 3-6

Select the m_Form variable, and then select the list of events from the drop-down on the right. You should now see the list of events that are available to use with the Access Form object. Select the Error event and add the following code to handle duplicate primary keys:

```
Private Sub m_Form_Error(DataErr As Integer, Response As Integer)
    If (DataErr = 3022) Then
        MsgBox "You have entered a duplicate key value. " & _
            "Please check the value and try again.", _
            vbExclamation, _
            "Cannot Save Record"
        ' suppress the Access error
        Response = acDataErrContinue
    End If
End Sub
```

With the Error event handled in the ExtendedForm class, you can subclass a Form object to respond to this event. Before we subclass the form, we need to create some objects. One of these tables will actually be used later.

- 1.** Create two tables based on the Contacts table template. Name the first table tblContacts and the second table tblEmployees.

2. Modify the design of the table so that the ContactID field is a Number data type to enable it to receive duplicate primary key errors that we will handle from the Error event.
3. Modify the design of the tblEmployees table to add a Date/Time field called HireDate.
4. Create a new form bound to the tblContacts table. Name the form frmContacts.

Now, follow these steps to create the subclass:

1. Set the HasModule property of the frmContacts form to Yes.
2. Add the following code to the declarations section of the form:

```
Dim objExForm As ExtendedForm
```

3. Now, add the following code to the Form_Open event. This code creates an instance of the ExtendedForm class and sets the Form property to the current form. When the form opens, it calls the Property_Set routine created earlier and sinks the Error event.

```
Private Sub Form_Open(Cancel As Integer)
    Set objExForm = New ExtendedForm
    Set objExForm.Form = Me
End Sub
```

Open the form and enter a duplicate ContactID value. You should receive the error displayed from the Error event in the ExtendedForm class, as displayed in Figure 3-7.

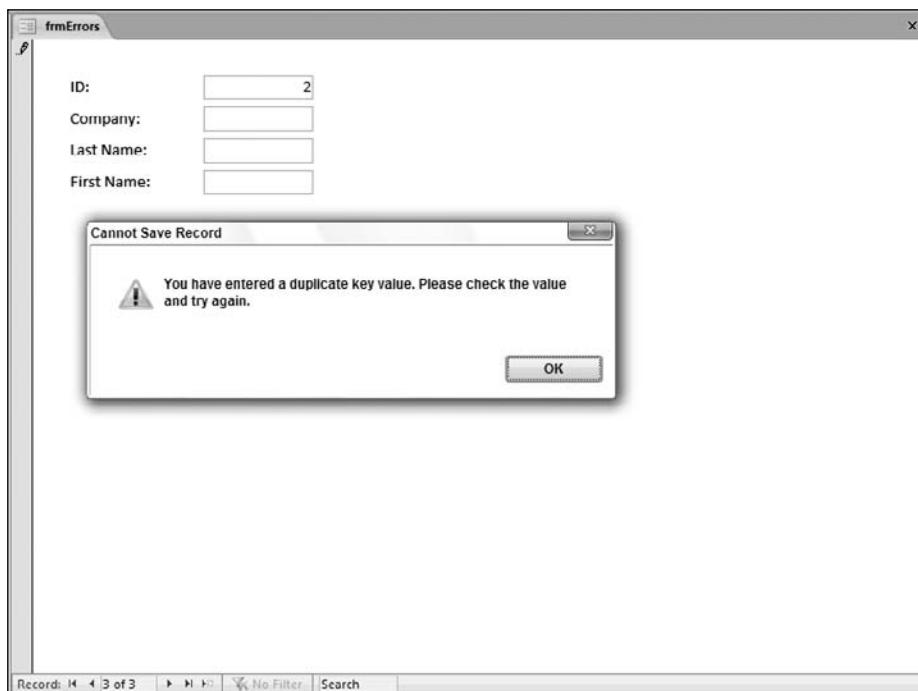


Figure 3-7

Part I: Programming Access Applications

That's it! You've just created a subclass! You can use this technique throughout your application to standardize the Error event of your forms.

Subclassing the Form_Load Event

As with the Error event, you might want to add consistency to certain forms when they are loaded. For example, you might have a group of forms that you dynamically make read-only based on some condition. To do so, you might write code that resembles the following in the Load event of the form:

```
Private Sub Form_Load()
    Me.AllowAdditions = False
    Me.AllowDeletions = False
    Me.AllowEdits = False
    Me.AllowFilters = True
End Sub
```

If you have multiple forms that use this same code, this might be a good opportunity to use a subclass. Add the following code to the ExtendedForm class you created earlier.

```
Private Sub m_Form_Load()
    m_Form.AllowAdditions = False
    m_Form.AllowDeletions = False
    m_Form.AllowEdits = False
    m_Form.AllowFilters = True
End Sub
```

Modify the Form Property Set routine from earlier to sink the Load event of the form:

```
Public Property Set Form(objForm As Form)
    Set m_Form = objForm

    ' sink events
    AddEventHook m_Form, "OnError"
    AddEventHook m_Form, "OnLoad"
End Property
```

Interfaces

In object-oriented programming, an interface is like a contract. Because interfaces are used to define members or behavior, interfaces typically do not contain any code. The actual implementation of these members is defined in separate class modules. Such classes are said to *implement* or *derive* from an interface. When multiple classes implement an interface, the interface can also be used as a common type between the classes.

Interfaces model the “is a” relationship in object-oriented programming. Because the interface is like a contract, once you define the behavior of the interface, you should not modify its members. Adding members is okay, as long as you don’t remove a member or change the method signature of a member. If a member is removed from an interface, any class that implements the member is now broken. This is bad.

By using an interface, you can change the underlying type of search by specifying a different class.

Inheritance in VBA

Interfaces provide what is known in object-oriented programming as *inheritance*. Inheritance allows you to change the functionality of a class by defining a base set of members that can be overridden or even added to.

The two types of inheritance in object-oriented programming are interface inheritance and implementation inheritance.

Interface inheritance is used to define methods or behaviors of a particular interface that a class implements. Classes can implement multiple interfaces to define their behavior. VBA supports interface inheritance.

With implementation inheritance you can add code to a base class and subsequently override it in a derived class. This is a powerful tool because you can write code once in the base class and then modify it only if needed. Unfortunately VBA does not support implementation inheritance, and as a result, you may need to write code in multiple derived classes when you implement an interface.

Writing an Interface

By convention, the name of an interface should begin with a capital letter “I.” Consider an interface named `ISearch` that is used to perform a search of different tables. In order to determine the table to search, you can specify the table name as a property. Or, you can abstract the search functionality and implement it in derived classes.

To create an interface, follow these steps:

1. Create a new class module named `ISearch`.
2. Add the following method to the class. This function defines a method called `Search` that accepts criteria and returns a DAO `Recordset2` object.

```
Public Function Search(Criteria As String) As DAO.Recordset2
    ' // Implement in derived classes
End Function
```

Implementing an Interface

To implement an interface, use the `Implements` keyword in the declarations section of a class as follows:

1. Create a new class module called `ContactSearch`.
2. Add the following code to the declarations section of the class to implement the `ISearch` interface:

```
Implements ISearch
```

3. When you implement an interface, each `Public` member of the interface must be implemented or the code does not compile. Select the `ISearch` object from the drop-down on the left. This should add the `Search` method that is defined in the `ISearch` interface. Add the following code for the `Search` method:

```
Private Function ISearch_Search(Criteria As String) As DAO.Recordset2
    Dim rs As DAO.Recordset2
```

Part I: Programming Access Applications

```
Set rs = CurrentDb.OpenRecordset("SELECT * FROM tblContacts WHERE " & Criteria)
Set ISearch_Search = rs
End Function
```

4. Create a new class module called EmployeeSearch.
5. Add the following code to the declarations section of the class to implement the ISearch interface:

Implements ISearch

6. Add the following code for the Search method:

```
Private Function ISearch_Search(Criteria As String) As DAO.Recordset2
    Dim rs As DAO.Recordset

    Set rs = CurrentDb.OpenRecordset("SELECT * FROM tblEmployees WHERE " & _
        Criteria)
    Set ISearch_Search = rs
End Function
```

Using an Interface

To use the ISearch interface, declare a variable of type ISearch and then instantiate it using either the ContactSearch or EmployeeSearch class. Add the following code to a new standard module to test the interfaces and the two classes. The tables used by these classes are available with the Interfaces.accdb database for download for this book.

```
Sub SearchTests()
    Dim objSearch As ISearch
    Dim rs As DAO.Recordset2
```

Set the objSearch variable to a new instance of the ContactSearch class:

```
Set objSearch = New ContactSearch
```

Call the Search method to return a Recordset2 object based on the tblContacts table and print the results of the recordset. The PrintRecordset helper function follows this code:

```
Set rs = objSearch.Search("[Last Name] = 'Cooper'")
PrintRecordset rs
```

Now, set the objSearch variable to a new instance of the EmployeeSearch class:

```
Set objSearch = New EmployeeSearch
Set rs = objSearch.Search("HireDate >= #1/1/2007#")
PrintRecordset rs

Set objSearch = Nothing
End Sub
```

Here is the PrintRecordset helper routine:

```
Sub PrintRecordset(rs As DAO.Recordset2)
    Dim stOut As String
    Dim fld As DAO.Field2
    ' heading
    Debug.Print String(80, "-")
    Debug.Print rs.Name
    Debug.Print String(80, "-")

    ' data
    While (Not rs.EOF)
        stOut = ""
        For Each fld In rs.Fields
            stOut = stOut & fld.Value & ","
        Next
        Debug.Print Left(stOut, Len(stOut) - 1)
        rs.MoveNext
    Wend

    rs.Close
    Set rs = Nothing
End Sub
```

When you run the `SearchTests` routine, search results from both tables should be displayed in the Immediate window.

Specialization

Adding members to a class that are not defined in an implemented interface is known as *specialization*. Using specialization you can extend a class to add additional functionality over what is defined in a particular interface. When you specialize, however, you can no longer use the interface type for the variable declaration because the interface doesn't know about the additional members.

Testing the Type of an Object

In order to prevent calling a method that doesn't exist, it is useful to test an object for its type. There are a couple different ways to do this.

TypeOf Operator

Use the `TypeOf` operator in VBA to determine if a class implements a particular interface or is an instance of another object such as a DAO Recordset. The following code determines whether an instance of the `ContactSearch` class implements the `ISearch` interface:

```
Sub TypeOfOperatorTests()
    Dim objContactSearch As ContactSearch
    Set objContactSearch = New ContactSearch

    Debug.Print TypeOf objContactSearch Is ISearch

    Set objContactSearch = Nothing
End Sub
```

Part I: Programming Access Applications

The following code determines whether the Recordset property for a form is an ADO recordset. If it is, the code calls the Save method of the ADODB Recordset object to save a recordset object as XML. Add a command button to a form and name it cmdSave to run this code. The form also needs to be bound to an ADO Recordset to actually save and it needs a reference to the ADO library to compile.

```
Private Sub cmdSave_Click()
    If (TypeOf Me.Recordset Is ADODB.Recordset) Then
        Me.Recordset.Save "d:\recordset.xml", adPersistXML
    Else
        MsgBox "Cannot save non-ADO recordset as XML"
    End If
End Sub
```

TypeName Function

The other way to determine the type of an object instance is to use the TypeName function. This function simply returns the name of the class as a String. If you are working with either an early bound or late bound Object variable, the TypeName function returns the string Nothing if the object has not been set. If you are working with a Variant, the TypeName function returns the string Empty if an object has not been set as demonstrated in the following code:

```
Sub TypeNameTest()
    Dim objSearchEarly As ISearch
    Dim objSearchLate As Object
    Dim varSearch As Variant

    ' Call the TypeName function before creating instances
    Debug.Print String(29, "-") & vbCrLf & _
                "BEFORE CREATING THE INSTANCES" & vbCrLf & _
                String(29, "-")
    Debug.Print "objSearchEarly: " & TypeName(objSearchEarly)
    Debug.Print "objSearchLate: " & TypeName(objSearchLate)
    Debug.Print "varSearch: " & TypeName(varSearch)

    Set objSearchEarly = New ContactSearch
    Set objSearchLate = New ContactSearch
    Set varSearch = New ContactSearch

    ' Call the TypeName function after creating instances
    Debug.Print String(29, "-") & vbCrLf & _
                "AFTER CREATING THE INSTANCES" & vbCrLf & _
                String(29, "-")
    Debug.Print "objSearchEarly: " & TypeName(objSearchEarly)
    Debug.Print "objSearchLate: " & TypeName(objSearchLate)
    Debug.Print "varSearch: " & TypeName(varSearch)

    ' cleanup
    Set objSearchEarly = Nothing
    Set objSearchLate = Nothing
    Set varSearch = Nothing
End Sub
```

When you run this code, you should see the following output that demonstrates the `TypeName` function used with objects and a `Variant`.

Output

```
-----  
BEFORE CREATING THE INSTANCES  
-----  
objSearchEarly: Nothing  
objSearchLate: Nothing  
varSearch: Empty  
-----  
AFTER CREATING THE INSTANCES  
-----  
objSearchEarly: ContactSearch  
objSearchLate: ContactSearch  
varSearch: ContactSearch
```

Notice that the `varSearch` variable that is declared as a `Variant` returns the string `Empty` before the object is created.

The `TypeName` function is also useful as the `Source` argument of the `Err.Raise` statement in error handlers as described in the section “Using `Err.Raise`” later in this chapter.

Determining Whether an Object Has Been Set

VBA has several functions that you can use to determine the type of data that you are working with. For example, you can use the `IsObject` function to determine whether a variable is an object, or the `IsNumeric` function to determine whether a variable is a number, or the `IsEmpty` function to determine whether a `Variant` value has been set. One thing we have often found lacking in VBA is an `IsNothing` function to determine whether an instance of an object has been set. So, here it is:

```
Public Function IsNothing(obj As Object) As Boolean  
    IsNothing = (obj Is Nothing)  
End Function
```

This function uses the `Is` operator to compare the object instance to the value `Nothing`.

Collections

Collections and arrays give you a place to put stuff. VBA even includes an object called appropriately enough `Collection` that you can use to put different types of stuff, such as strings, numbers, dates, or even objects. For flexibility, the VBA `Collection` object stores `Variant` values so that you can put anything into it. Many times, however, we want to store an object of a single type such as a custom class that we write. In these cases we don't want to put other types of objects in the collection. In other words, what we really want is a strongly typed collection. To solve this problem, you can write your own collection class.

Part I: Programming Access Applications

Writing the Standard Collection Class

By convention in VBA, collection classes have four members as listed in the following table.

Member	Description
Add	Adds an item to the collection
Item	Returns an individual item from the collection
Count	Returns the number of items stored in the collection
Remove	Removes an item from the collection

So far we've been working with `ISearch` objects that define different types of searches in our database. Let's extend this now to store multiple searches that can be searched all at once. Start by creating a new class module named `Searches`.

By convention, collection objects frequently end with an "s" to indicate that there are multiple items being stored in the class.

Storage

The underlying storage for the `ISearch` objects in this collection is a VBA `Collection` object. Add the following code to the declarations section of the `Searches` class:

```
Private m_col As VBA.Collection
```

Add Method

Next, add the `Add` method. The `Add` method in the VBA `Collection` object accepts a `Variant`. We make ours accept an `ISearch` instance so that the collection class is strongly typed.

```
Public Sub Add(objSearch As ISearch)
    ' add the object to the collection
    m_col.Add objSearch, Str(ObjPtr(objSearch))
End Sub
```

Notice that we are simply calling the `Add` method of the VBA `Collection` object. The second argument of the `Add` method accepts a value known as a `Key` that is unique in the collection. Because this should be unique, use the `ObjPtr` function to get the memory address of the `ISearch` object that was passed to the `Add` method.

Item Method

Now, add the `Item` method. The `Item` function returns a particular item in the collection. This function returns a `Variant` in the VBA `Collection` object but we make ours return an `ISearch` object so that the collection is strongly typed.

```
Public Function Item(varKey As Variant) As ISearch
    Set Item = m_col(varKey)
End Function
```

Count Property

Add the Count property to return the number of items in the collection. This simply wraps the Count function in the VBA Collection object. We prefer the syntax of a property here instead of a method.

```
Public Property Get Count() As Long  
    Count = m_col.Count()  
End Property
```

Remove Method

Next, add the Remove method. This method wraps the Remove method of the VBA Collection object:

```
Public Sub Remove(varKey As Variant)  
    m_col.Remove varKey  
End Sub
```

Initialization and Cleanup

You need to instantiate the VBA Collection object that was declared at the top of the module. To do so, add the following code in the Initialize event of the class to create the underlying collection when our Searches collection is created.

```
Private Sub Class_Initialize()  
    Set m_col = New VBA.Collection  
End Sub
```

Last, you should clean up after yourself when you're done, so add the following code to the Terminate event of the class:

```
Private Sub Class_Terminate()  
    Set m_col = Nothing  
End Sub
```

Using the Collection Class

You need to create collection classes just as you would any other class, by instantiating it with the New keyword. In many cases, you want to ensure that an instance of the collection is always available in the application. To achieve this, you can declare a new instance of the collection class in a standard module, as follows:

```
Public Searches As New Searches
```

The Searches variable always returns a new instance of the Searches collection class and subsequently guarantees that the instance never goes out of scope. If you have an object termination routine, such as the one mentioned in Appendix A, you can destroy the Searches variable when the application closes.

Once you have an instance of the collection class, you can call its methods to work with the collection. Add the following code to a new standard module to test the Searches collection class.

```
Sub CollectionTest()  
    Dim objSearch As ISearch  
    Dim i As Long
```

Part I: Programming Access Applications

```
' create a new ContactSearch and add it to the collection
Set objSearch = New ContactSearch
Searches.Add objSearch

' create a new EmployeeSearch and add it to the collection
Set objSearch = New EmployeeSearch
Searches.Add objSearch

' print the objects in the collection
For i = 1 To Searches.Count
    Debug.Print TypeName(Searches.Item(i))
Next

Set Searches = Nothing
End Sub
```

When you run this code you should see the following output in the Immediate window:

```
ContactSearch
EmployeeSearch
```

This is pretty useful. Because the collection now stores `ISearch` objects, you can call the `Search` method for each different type of search. Add the following line to the `CollectionTest` routine:

```
Dim rs As DAO.Recordset2
```

Now, add the following code inside the loop in the `CollectionTest` routine. This code requires the `PrintRecordset` helper routine that we wrote earlier in this chapter.

```
Set rs = Searches.Item(i).Search(" ")
PrintRecordset rs
```

Because the `Item` method returns an instance of the `ISearch` class, you even get IntelliSense, as shown in Figure 3-8!



The screenshot shows a Microsoft Word document containing VBA code. The code is as follows:

```
Option Compare Database
Option Explicit

Public Searches As New Searches

Sub CollectionTest()
    Dim objSearch As ISearch
    Dim i As Long
    Dim rs As DAO.Recordset2

    ' create a new ContactSearch and add it to the collection
    Set objSearch = New ContactSearch
    Searches.Add objSearch

    ' create a new EmployeeSearch and add it to the collection
    Set objSearch = New EmployeeSearch
    Searches.Add objSearch

    ' print the objects in the collection
    For i = 1 To Searches.Count
        Set rs = Searches.Item(i).Search
        PrintRecordset rs
        ' Debug.Print TypeName(Searches.Item(i)), rs.Name
    Next

    Set Searches = Nothing
End Sub
```

In the code, the line `Set rs = Searches.Item(i).Search` is highlighted, and the word "Search" is underlined with a red dotted line, indicating that it is a member of the `ISearch` interface. A tooltip or dropdown menu is visible over the word "Search", showing the available methods and properties for that object.

Figure 3-8

An Alternate Implementation for the Add Method

The method signature for the Add method in the VBA Collection object is as follows:

```
Sub Add(Item, [Key], [Before], [After])
```

We've implemented our Add method in a similar manner in the sense that we declared it as a Sub routine — it does not return a value. As an alternative, you can use the Add method to return an instance of the object being created. In this type of implementation, the object being created typically has some additional data that you want to use. So, let's modify the ISearch interface to add more data.

Add the following read-write property to the ISearch class. This defines a name for a given search. In a real implementation you might use the Name property to uniquely identify a search.

```
Public Property Get Name() As String
    ' // Implement in derived classes
End Property
Public Property Let Name(stName As String)
    ' // Implement in derived classes
End Property
```

Let's also add an Enum that indicates the type of search.

```
Public Enum SearchTypeEnum
    Contact
    Employee
End Enum
```

Last, add a property that returns the type of search.

```
Public Property Get SearchType() As SearchTypeEnum
    ' // Implement in derived classes
End Property
```

You also need to implement this property in any derived classes that you have. Here is the implementation of this property in the ContactSearch class.

```
Private m_Name As String
Private Property Let ISearch_Name(RHS As String)
    m_Name = RHS
End Property
Private Property Get ISearch_Name() As String
    ISearch_Name = m_Name
End Property
Private Property Get ISearch_SearchType() As SearchTypeEnum
    ISearch_SearchType = Contact
End Property
```

When you implement a property in an interface, VBA gives you an argument named RHS. This stands for "right-hand side" and represents the right-hand side of an assignment statement.

Part I: Programming Access Applications

Now, create a new class module called `Searches2`. This contains an implementation of the `Add` method that returns an `ISearch` instance. Because we have two new properties that help define the search, use them as the arguments for the `Add` method.

```
Public Function Add(SearchName As String, SearchType As SearchTypeEnum) As ISearch
    Dim objSearch As ISearch
```

The `SearchType` argument determines the type of search being created. Add the following code to the `Add` method to create the correct type of object and set the `Name` property.

```
' create the ISearch and set its name
If (SearchType = Contact) Then
    Set objSearch = New ContactSearch
ElseIf (SearchType = Employee) Then
    Set objSearch = New EmployeeSearch
End If
objSearch.Name = SearchName
```

Last, you need to add the item to the collection and return the `ISearch` instance for the `Add` method.

```
' add the object to the collection
m_col.Add objSearch, Str(ObjPtr(objSearch))

' return the object
Set Add = objSearch
End Function
```

After you write the `Add` method, add the remaining code from the `Searches` collection class for the `Item`, `Count`, and `Remove` members. Then, add the following code to the standard module to test the `Searches2` collection class.

```
Public Searches2 As New Searches2
Sub CollectionTest2()
    Dim objSearch As ISearch
    Dim i As Long

    ' add two searches
    Set objSearch = Searches2.Add("Test Contact Search", Contact)
    Set objSearch = Searches2.Add("Test Employee Search", Employee)

    ' enumerate the searches in the collection
    For i = 1 To Searches2.Count
        Debug.Print Searches2.Item(i).Name, Searches2.Item(i).SearchType
    Next

    Set Searches2 = Nothing
End Sub
```

Notice that you can now create an instance of the `ISearch` interface by using the `Add` method.

Setting a Default Member

When you typically work with collections, you don't have to call the Item method directly as we have here because the Item method is set as the default member of a collection class. You can do the same thing for your classes but there is some extra work involved.

Again, because VBA has its roots in VB, you can set an attribute on the Item method to tell VBA that this is the default member of the class. The VBA development environment, however, does not provide a way to do this. As a result, you have to export the class to a text file to create the attribute, and then import it into your project.

You can have only one default member of a class.

Use the following steps to set the Item method as the default member of the Searches collection class:

1. In the Visual Basic Editor (VBE), right-click on the Searches class and choose Remove Searches.
2. When prompted to export the class, click Yes and save it to the location of your choosing. This creates a .cls file named Searches.cls.
3. Open the Searches.cls file in Notepad.
4. Add the Attribute keyword to the Item method so that it reads:

```
Public Function Item(varKey As Variant) As ISearch
Attribute Value.VB_UserMemId = 0
    Set Item = m_col(varKey)
End Function
```

5. Close and save the Searches.cls file and return to the VBE.
6. On the File menu, choose Import File and open the Searches.cls file you saved.
7. Compile and save the project.

Once you've performed these steps, you can modify the CollectionTest routine as follows:

```
' print the objects in the collection
For i = 1 To Searches.Count
    Set rs = Searches(i).Search("")
    PrintRecordset rs
Next
```

Notice that we've removed the Item method because it is now the default member of the Searches class.

For Each Enumeration

The other aspect of VBA collection classes that we find annoying is that by default you cannot use the For Each statement to iterate through them. The For Each statement calls a NewEnum function that is defined for the class. Add this function to the Searches class:

```
Public Function NewEnum() As IUnknown
    Set NewEnum = m_col._NewEnum
End Property
```

Part I: Programming Access Applications

Follow the same steps as earlier for creating a default member but change the `NewEnum` method as follows:
(The `VB_UserMemId` value of `-4` tells VBA that this method will be used by the `For Each` statement.)

```
Public Function NewEnum() As IUnknown
Attribute NewEnum.VB_UserMemId = -4
    Set NewEnum = m_col.[_NewEnum]
End Function
```

Change the loop code in the `CollectionTest` routine to use the `For Each` statement:

```
' print the objects in the collection
For Each objSearch In Searches
    Set rs = objSearch.Search("")
    PrintRecordset rs
Next
```

Unfortunately VBA does not allow hidden members, so you cannot hide members of a class.

The `ICollectionEx` Interface

The .NET Framework adds some nice functionality to collections, such as being able to clear the collection or to call a method for each member of the collection. We can extend our collections to add similar functionality using an interface.

We call this interface `ICollectionEx`, which contains a few members that extend the `Collection` class found in VBA. The following table lists the members defined by the `ICollectionEx` interface:

Member Name	Description
Clear	Removes all items from the collection.
Contains	Determines whether the collection contains the specified item given some criteria.
Find	Finds an item in the collection.
ForEach	Runs a procedure for each item in a collection.

Add the following method signatures to the `ICollectionEx` interface:

```
Public Sub Clear()
    ' // Implement in derived classes
End Sub
Public Function Contains(vData As Variant) As Boolean
    ' // Implement in derived classes
End Function
Public Sub ForEach(strFunction As String)
    ' // Implement in derived classes
End Sub
Public Function Find(vData As Variant) As Variant
    ' // Implement in derived classes
End Function
```

We extend the `Searches` collection class to implement the `ICollectionEx` interface. Add the following line of code to the declarations section of the `Searches` class:

```
Implements ICollectionEx
```

Now let's take a look at each of these members in more detail by implementing each one in the `Searches` class.

Clear Method

The `Clear` method is easy. It simply creates a new instance of the underlying `Collection` object, as follows:

```
Private Sub ICollectionEx_Clear()
    Set m_col = New VBA.Collection
End Sub
```

Contains Method

Use the `Contains` method to return `True` or `False` based on some condition that is specific to the collection class being searched. For the `Searches` collection, we pass the name of the `ISearch` object to the `Contains` method so this walks each item in the collection, as follows:

```
Private Function ICollectionEx_Contains(vData As Variant) As Boolean
    Dim i As Long

    For i = 1 To m_col.Count
        If (m_col(i).Name = vData) Then
            ICollectionEx_Contains = True
            Exit For
        End If
    Next
End Function
```

By using an interface such as `ICollectionEx`, you can alter the behavior of what you are looking for in each collection.

Find Method

The `Find` method is similar to the `Contains` method with the exception that we return the matching `ISearch` object instead of `True` or `False`:

```
Private Function ICollectionEx_Find(vData As Variant) As Variant
    Dim i As Long

    For i = 1 To m_col.Count
        If (m_col(i).Name = Criteria) Then
            Set ICollectionEx_Find = m_col(i)
            Exit For
        End If
    Next
End Function
```

Part I: Programming Access Applications

ForEach Method

The `ForEach` method accepts the name of a function to run for each item in the collection. The function that is called should take an argument of the same data type as the items in the collection and is called using the `Run` method of the Access Application object.

```
Private Sub ICollectionEx_Foreach(strFunction As String)
    ' runs the specified function name for each item
    Dim i As Long

    For i = 1 To m_col.Count()
        Application.Run strFunction, m_col(i)
    Next
End Sub
```

The `ForEach` method moves the responsibility of enumerating the collection into the collection itself so that you can reduce the amount of looping code required in an application. In a debugging type of scenario, you might have a function such as this one that simply prints the name of the `ISearch` object to the `Immediate` window.

```
Public Sub PrintSearchInfo(objSearch As ISearch)
    Debug.Print objSearch.Name
End Sub
```

What's cool about this function is that you can extend it to call any function that accepts a single argument for the items in your collection. For example, if you have a collection of `Customer` objects and you want to send e-mail to each one, you might write a function that resembles the following, assuming that the `Customer` object has properties defined as `Name` and `EmailAddress`.

```
Public Sub SendCustomerEmail(objCustomer As Customer)
    ' send mail
    Dim strTo As String
    Dim strBody As String
    Dim strMail As String

    ' generate the email
    strTo = objCustomer.EmailAddress
    strBody = "Hi " & objCustomer.Name & ",%0d%0a" & _
        "Thank you for your business! We look forward to serving you " & _
        "in the coming year.%0d%0a" & _
        "- Your Company Here"

    strMail = "mailto:" & strTo & "?subject=Thanks!&body=" & strBody & ""

    ' send the email
    FollowHyperlink strMail

End Sub
```

Use the `FollowHyperlink` method as an alternative to using the `ShellExecute` API function to send e-mail, as described in Chapter 2. E-mail messages are an Internet standard so carriage returns and line-feeds should be encoded using HTML encoding, `%0d` and `%0a` respectively. Calling this in a loop such as the `ForEach` method may generate multiple email windows so keep that in mind when calling functions inside of a loop.

Testing the **ICollectionEx** Interface

Now it's time to put all of these new methods to good use. Create the following code in a standard module:

```
Sub ICollectionExTest()
    Dim objColEx As ICollectionEx
    Dim objMatch As ISearch
```

Next, set the **ICollectionEx** object. We already know that the **Searches** collection implements **ICollectionEx** but it's a good idea to verify:

```
' get the extended collection object
If (TypeOf Searches Is ICollectionEx) Then
    Set objColEx = Searches
End If
```

Call the **Clear** method to clear any existing searches:

```
' clear any existing searches
objColEx.Clear
```

Add some new searches and set their names:

```
' add some new searches
Searches.Add New ContactSearch
Searches(1).Name = "New ContactSearch object"

Searches.Add New EmployeeSearch
Searches(2).Name = "New EmployeeSearch object"
```

Now, test the **ForEach** method. This code requires the **PrintSearchInfo** routine defined earlier. Because the looping is handled in the collection itself, looping through a collection is reduced to one line!

```
' test the ForEach method
objColEx.ForEach "PrintSearchInfo"
```

Try a couple different tests for the **Contains** method:

```
' test the Contains method
Debug.Print
Debug.Print "Contains(""New ContactSearch object"": " & _
    objColEx.Contains("New ContactSearch object")
Debug.Print "Contains(""Object does not exist"": " & _
    objColEx.Contains("Object does not exist")
```

And last, verify that the **Find** method works:

```
' test the Find method
Debug.Print
Set objMatch = objColEx.Find("New EmployeeSearch object")
Debug.Print "Found: "
Debug.Print objMatch.Name, objMatch.SearchType
```

Part I: Programming Access Applications

```
Set objColEx = Nothing  
End Sub
```

You should see the following output when you run this routine:

```
New ContactSearch object  
New EmployeeSearch object  
Contains("New ContactSearch object"): True  
Contains("Object does not exist"): False  
Found:  
New EmployeeSearch object      1
```

Events

As mentioned earlier, Windows uses events extensively as a mechanism for communicating back and forth between processes and within Windows itself. Access being a Windows application also uses events extensively for forms, reports, and controls.

Events, by their nature of providing a means for communication, are always `Public`. `Private` events wouldn't make sense because there wouldn't be a way for a client application to handle the event.

Why Write Custom Events?

There are certainly other ways of providing notification to a caller. For example, you could set property values or flags within the class, or even display messages to the user directly from the class. So why would you use custom events?

Events provide a means for providing notifications to a caller that allows the caller to determine what to do with the notification. For example, say you have an application that tracks orders for a restaurant. The application contains a class module called `Order` that tracks the state of an order. Possible states of the order are when it is first placed, when it is being prepared, when preparation is complete, and last when it has been delivered to the customer. The `Order` class might fire events for all of these different states that client applications can respond to differently.

Let's take a look at how client applications in a restaurant might respond to these events as shown in the following table.

Event	Client	Response
OrderPlaced	Kitchen	Receives the order and starts preparation.
	Wait Staff	Receives notification from the kitchen with an estimated delivery time.
PreparingOrder	Kitchen	No response — kitchen is busy doing work.
	Wait Staff	Receives notification from the kitchen every two minutes that the order is being prepared.

Event	Client	Response
OrderComplete	Kitchen	Sends notification to the wait staff that an order is complete.
	Wait Staff	Receives notification from the kitchen and retrieves the order.
OrderDelivered	Kitchen	No response — the order has left the kitchen.
	Wait Staff	Receives a notification to check in with the customer.

As you can see, depending on who the client is that subscribes to an event, the response can be quite different.

Creating Custom Events

Two keywords define custom events. The first is the `Event` keyword that defines the event itself. Use the `Event` keyword in the declarations section of the class. The second is the `RaiseEvent` keyword that actually fires the event.

Event Keyword

The `Event` keyword is used in the declarations section of a class and defines an event and its arguments. Unfortunately, however, you cannot define events in an interface and implement them in a derived class. This means that you'll need to define the events for each class that should raise them.

Take a look at the `ContactSearch` class that implements the `ISearch` interface. Let's say that we want to add some events that fire before and after a search. Add the following event definitions to the class:

```
Public Event BeforeSearch()  
Public Event AfterSearch()
```

RaiseEvent Keyword

Once you've defined the events, you need to fire them using the `RaiseEvent` keyword. Events that are defined in a class are displayed with IntelliSense, as shown in Figure 3-9.

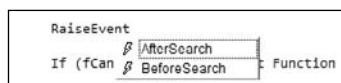


Figure 3-9

To fire the `BeforeSearch` event, add the following line of code to the beginning of the `ISearch_Search` method of the `ContactSearch` class:

```
RaiseEvent BeforeSearch
```

Part I: Programming Access Applications

To fire the `AfterSearch` event, add the following line of code to the end of the `ISearch_Search` method of the `ContactSearch` class:

```
RaiseEvent AfterSearch
```

Creating an Event That Can Be Cancelled

Access defines several events that can be cancelled such as the `BeforeUpdate` event of a control or the `BeforeInsert` event of a form. You also can create an event that can be cancelled by passing a parameter to an event. Access defines the `Cancel` argument for these events as an `Integer` but we've always felt they should be a `Boolean`. Modify the declaration of the `BeforeSearch` event in the `ContactSearch` class, as follows:

```
Public Event BeforeSearch(Cancel As Boolean)
```

To allow the caller to specify whether the event and `Search` method should be cancelled, declare a `Private` variable in the declarations section of the `ContactSearch` class.

```
Private fCancelEvent As Boolean
```

Modify the `RaiseEvent` statement that fires the `BeforeSearch` event as follows and add the following additional line:

```
RaiseEvent BeforeSearch(fCancelEvent)

' cancel the event
If (fCancelEvent) Then Exit Function
```

The `fCancelEvent` argument is set by an event handler that specifies `True` or `False` to the `Cancel` argument of the event. If the caller specifies `True`, this code exits the `Search` method.

Listening for Custom Events

You can listen for custom events by defining an instance of your class using the `WithEvents` keyword. As mentioned earlier in this chapter, the `WithEvents` keyword tells VBA that you want to handle events that are exposed by the class. The `WithEvents` keyword can be used in a class module only.

To listen for the events defined by the `ContactSearch` object, create a new form with two command buttons named `cmdCancelTrue` and `cmdCancelFalse`. Next, add the following code to the declarations section of the form.

```
Private WithEvents objSearch As ContactSearch
Private objISearch As ISearch
Private rs As DAO.Recordset2
Private fCancel As Boolean
```

Because `ISearch` defines the members of the class but `ContactSearch` defines the events, we need variables of both types. Add code to the `Load` event of the form to create the instances and set the name of the search.

```
Private Sub Form_Load()
    Set objISearch = New ContactSearch
```

```
    objISearch.Name = "Testing search events"  
  
    Set objSearch = objISearch  
End Sub
```

Add the following code for the Click events of both buttons. In this code, the button named cmdCancelFalse sets the fCancel flag to False, whereas the button named cmdCancelTrue sets the flag to True. Both buttons start a search by calling the ISearch.Search method:

```
Private Sub cmdCancelFalse_Click() : fCancel = False  
  
    ' start the search  
    Set rs = objISearch.Search("")  
End Sub  
Private Sub cmdCancelTrue_Click()  
    fCancel = True  
  
    ' start the search  
    Set rs = objISearch.Search("")  
End Sub
```

Now, add the event handler for the BeforeSearch event. The Cancel argument of the event is passed the value of the fCancel flag that was set by either button.

```
Private Sub objSearch_BeforeSearch(Cancel As Boolean)  
    Cancel = fCancel  
End Sub
```

Last, add the event handler for the AfterSearch event. This simply displays the name of the search but in a real world application you can do much more.

```
Private Sub objSearch_AfterSearch()  
    MsgBox objISearch.Name  
End Sub
```

Run the form to test the events. If you click the cmdCancelFalse button, notice that the AfterSearch event is not fired because the Cancel argument caused us to exit the Search method. Now click the cmdCancelTrue button and notice that the AfterSearch event fires.

Design Considerations

Generally speaking we design class modules with their usage in mind. In other words, think about the way you would like to use the class when designing. Chances are if you think as a user, you'll be able to spot any holes in the design.

There are some common problems that we have run across when implementing classes that bear mentioning. The examples in the following sections answer the following questions:

- ❑ How do you create a new instance of a class from an existing instance?
- ❑ How do you expose the interface members of a derived class to allow for specialization?
- ❑ What do you do with errors that occur within a class?

Copying a Class Instance

In memory, an Object variable is simply a pointer to the actual data for an individual instance of a class. As a result, the Object data type consumes only 4 bytes of memory. Let's say then that you have two instances of the ContactSearch class that implements the `ISearch` interface we have been working with throughout this chapter. You might instantiate the first instance but create a copy of the second instance as follows:

```
Sub InvalidCopy()
    Dim objSearch1 As ISearch
    Dim objSearch2 As ISearch

    ' create the first search
    Set objSearch1 = New ContactSearch

    ' set search properties on the first instance
    objSearch1.Name = "First Search"

    ' Copy the first instance into the second
    Set objSearch2 = objSearch1

    ' set the name of the second search
    objSearch2.Name = "Second Search"

    Debug.Print objSearch1.Name
    Debug.Print objSearch2.Name

    Set objSearch2 = Nothing
    Set objSearch1 = Nothing
End Sub
```

When you run this code, you notice that the `ISearch.Name` property for `objSearch1` was changed to "**Second Search**"! This change occurs because `objSearch2` was set to a copy of `objSearch1`, but what happened is that only the underlying pointer value was copied. Both variables are now pointing to the same instance in memory.

To prevent this problem, create a new method in the `ISearch` interface called `Clone`. This method is defined as follows:

```
Public Function Clone() As ISearch
    ' // Implement in derived classes
End Function
```

There are two benefits to putting this method in the interface. First, it allows you to easily create a copy from one `ISearch` instance to another. Second, by putting it in the `ISearch` interface, we can return a strongly typed instance of an object that implements `ISearch` and guarantees type compatibility.

Implement this method in the `ContactSearch` class, as follows:

```
Private Function ISearch_Clone() As ISearch
    ' create the new instance and return
    Set ISearch_Clone = New ContactSearch
    ISearch_Clone.Name = ISearch_Name
End Function
```

When the `Clone` method is called, it creates a new instance of the `ContactSearch` class and copies any properties from the current instance to the new instance. By creating a new instance you guarantee that any data remains separated from an existing instance of the class.

It might be useful, however, to expose the `Clone` method directly on the derived classes, so add a `Public` method to the `ContactSearch` class called `Clone`. This method wraps the `Clone` method that is defined in the `ISearch` interface and is provided only to make it visible to client code that declares a `ContactSearch` object directly.

```
Public Function Clone() As ISearch
    Set Clone = ISearch_Clone()
End Function
```

Once you have implemented the `Clone` method, you can now create a copy of the `ContactSearch` class by calling the `Clone` method as follows:

```
Set objSearch2 = objSearch1.Clone()
```

Notice that when you run this code the `Name` property has not been changed.

Exposing the Interface of a Derived Type

We cheated here a little by exposing the `Clone` method publicly. If you called foul on the previous example of the `Clone` method, good for you! Typically in VBA when you are working with derived classes you should encourage callers to declare variables of the interface type and not the class type. In other words, in this case you would encourage callers to declare a variable of type `ISearch`. This provides the members that you want to expose as they are defined in the interface.

In a true object-oriented design, however, you want to specialize an object. In order to retrieve members defined in the derived class, you have to declare a variable of that type. So now you're stuck, right? Well, no. It turns out there are a few ways to work around this.

Wrapping the `Private` interface members in a derived class as `Public` members as we have done here is one way, but it adds the overhead of maintaining additional code and complexity. If you have a large interface or base class that you are implementing you're unlikely to do this. You could ask client code to declare two variables, one of the interface type and one of the class type. This is inexpensive, but call us cheap — we don't want to come off of the extra 4 bytes. So, here's another way.

Create a read-only property in the derived class that returns an instance of the interface type. Because the derived type itself is an instance of the interface type, this can simply return the current instance of the class using the `Me` keyword. For example, in the `ContactSearch` class, add the following code:

```
Public Property Get Interface() As ISearch
    Set Interface = Me
End Property
```

Now, use the `Interface` property on an instance of the `ContactSearch` class to get to the `Public` members of `ISearch`. This might resemble the following:

```
Dim obj As New ContactSearch
obj.Interface.Name = "New ContactSearch Object"
```

Part I: Programming Access Applications

Raising Errors

Class modules are often written to be used by other code, either in your own applications or by other programmers. Therefore, when something unexpected happens inside of a class module, it's best to expose errors to the caller and let them handle it. Some client applications that use a class may choose to ignore errors, whereas others go so far as to log them or e-mail the developer. By exposing errors from your classes, you are giving the client application the choice.

VBA and other libraries, such as the Access object model, or data access libraries, such as DAO and ADO, define their own errors. In order to prevent conflicts with error numbers in other libraries, it is a generally accepted practice to start custom errors in class modules with the following value:

```
vbObjectError + 513
```

In many cases, this guarantees that the error numbers you expose from your class are unique. We all know that so called *magic* numbers in code are bad. A magic number is a number that appears in code that is not defined as a constant, so it doesn't have meaning. Enumerations are a nice way to define error codes while avoiding magic numbers.

We also like to use a 2- or 3-letter prefix for constants in our applications that have meaning within the application. For example, we wrote a weekly status application to track items that we worked on throughout the year. Because we called the application Weekly Status, our constants were named with the prefix WS. Therefore, the error codes were defined something like:

```
Public Enum WeeklyStatusErrorCodes
    WS_SUCCESS = 0
    WS_ERR_INVALID_DATE = vbObjectError + 513
    WS_ERR_INVALID_ITEM
    WS_ERR_ITEM_NOT_FOUND
    WS_ERR_CANNOT_CONNECT
    WS_ERR_SETTING_NOT_FOUND
    WS_ERR_OUTLOOK_NOT_INSTALLED
    WS_ERR_CANNOT_CREATE_CHART
End Enum
```

Because numbers in an `Enum` are automatically incremented you just have to define the first one.

Using Err.Raise

You expose errors to outside callers by using the `Err.Raise` statement in VBA. Inside the error handler for a class module, raise an error as shown in the following sample method. Create a new class called `RaisesErrors` for this sample.

```
Public Enum CollectionTestErrors
    C_SUCCESS = 0
    C_ERR_INVALID_FORMAT = vbObjectError + 513
    C_ERR_INVALID_SEARCH
End Enum
Public Sub ThrowAnError()
    On Error GoTo ErrorHandler

ErrorHandler:
    Err.Raise C_ERR_INVALID_SEARCH, TypeName(Me), _
```

```
"You have specified an invalid search"  
Exit Sub  
End Sub
```

Now, assume that you have a client application that calls this method.

```
Sub CatchAnError()  
    On Error GoTo ErrorHandler  
  
    Dim re As New RaisesErrors  
    re.ThrowAnError  
    Exit Sub  
  
ErrorHandler:  
    If (Err = C_ERR_INVALID_SEARCH) Then  
        MsgBox Err.Description & vbCrLf & Err.Number, , Err.Source  
    End If  
    Set re = Nothing  
End Sub
```

When the class raises the `C_ERR_INVALID_SEARCH` error, the client application is able to catch it in its error handler and take the appropriate action. In this case, we simply display the error to the user.

Defining an Error Event

If your class is going to be used from another class module, such as an Access form, you might choose to raise an event that describes the error instead. This is similar to the `Error` event for a form that was described earlier in the section “Subclassing the `Form_Error` event.” Then, forms or reports that use your class can handle errors in a single location.

Add the following `Event` declaration to the class.

```
Public Event Error(ErrorCode As CollectionTestErrors, ErrorMessage As String)
```

Now, add a new method to the `RaisesErrors` class that raises this event.

```
Public Sub RaiseAnErrorEvent()  
    RaiseEvent Error(C_ERR_INVALID_FORMAT, "Invalid format specification")  
End Sub
```

Create a new form with a command button named `cmdErrors`. Add the following code to the form.

```
Dim WithEvents objErrors As RaisesErrors  
Private Sub cmdError_Click()  
    Set objErrors = New RaisesErrors  
    objErrors.RaiseAnErrorEvent  
End Sub  
Private Sub objErrors_Error(ErrorCode As CollectionTestErrors, _  
    ErrorMessage As String)  
    MsgBox ErrorMessage  
End Sub
```

When you click the button, the class raises the `Error` event defined in the class that is subsequently handled by the event handler in the form.

Summary

Class modules add a great amount of flexibility to an application, and when designed well, a great amount of simplicity. This simplicity can sometimes come at the cost of additional work, but once the investment is made, the payoff can be felt for years to come.

Some of the key points in this chapter are:

- ❑ Class modules enable you to write code that can be shared or reused throughout multiple applications.
- ❑ Design is critical. Spend time designing for flexibility and the way you would like to use the classes. This leads to classes that are both enjoyable and easy to use.
- ❑ Less is more. You can push work into a class module to make calling code easier later on.
- ❑ Use interfaces to define a contract between classes and to provide behavioral concepts to a class.
- ❑ Subclassing enables you to add common behaviors to multiple objects.
- ❑ Collections can be much more flexible and provide a great amount of functionality beyond the Add, Item, Count, and Remove methods.

We wrap up Part I in the next chapter, “Debugging, Error Handling, and Coding Practices.” This chapter provides you with information to effectively debug your applications and to use error handling to your advantage so that you are aware of any problems that may arise from an application. You also use some of the API functions and techniques for writing class modules that have been discussed from the previous two chapters.

4

Debugging, Error Handling, and Coding Practices

We all have to deal with certain tasks of application development. While customers may never directly see these tasks — such as debugging and error handling — the applications we develop are better for it. We do these tasks in our applications all the time during development, and it's often a part of the job that developers don't enjoy.

Although the effects of these activities are not necessarily obvious to your users, improper or insufficient error handling is very obvious. Think about how many times you have tried to hide Access error messages from your users. For an end user, seeing a custom error message provides a better experience than a dialog box that asks them to debug.

In this chapter you learn:

- How to effectively debug VBA code
- How to programmatically build a call stack
- How to create debug builds for your applications that provide functionality during development
- How to use assertions in code to help with debugging
- How to write your own `Assert` method
- How to use error handling as a feature to notify you when things go wrong
- General coding practices that are helpful in day-to-day development

Debugging

Debugging is one of those things that many people seem to either love or hate. Personally, we enjoy debugging. We find that it gives us a better understanding of the code, particularly when inheriting a project from another developer. To us, debugging (and really writing code in general) is a mixture of art and science. We find the aspects of designing software as enjoyable as writing the actual code. Debugging is one small part of that.

In this section, you learn about the different tools at your disposal for debugging VBA code. Some of these tools are built in to the VBA language itself, while others are part of the VBA Integrated Development Environment (IDE), also known as the Visual Basic Editor (VBE).

Using Watches

The VBE has several different debugging windows. We'll begin with the Watches window. A watch is an expression that can be monitored in running code. An expression can be any statement in VBA that can be evaluated, such as the following:

Variables

```
Dim strTest As String
```

Assignment statements

```
strTest = "test"  
lngTest = TestRoutine()
```

Property values

```
rs.NoMatch
```

Function calls

```
TestRoutine()
```

Certain types of watches also inject breakpoints based on the value of the expression. For debugging issues that are difficult to narrow down, watches are very useful. Because functions used as expressions in the Watches are evaluated, you should be cautious when using a function call in the Watches window.

Adding a Quick Watch

One of the easiest ways to add a watch is to highlight an expression in the VBE and select Quick Watch from the Debug menu. This opens the Quick Watch dialog box, as shown in Figure 4-1.

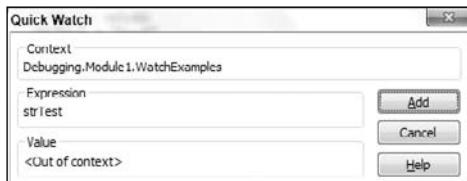


Figure 4-1

Chapter 4: Debugging, Error Handling, and Coding Practices

A quick watch enables you to watch the value of an expression in the Watches window. A quick watch creates a watch statement with the default settings.

Adding a Watch

If you need to change the type of watch that you create, highlight an expression in the VBE and select Add Watch from the Debug menu. This opens the Add Watch dialog box, as shown in Figure 4-2.

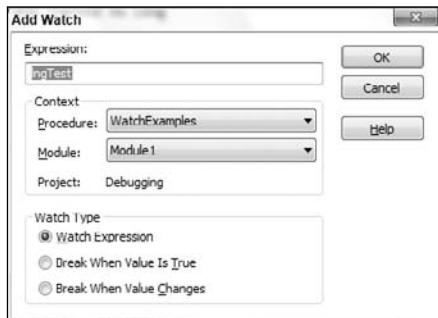


Figure 4-2

The Add Watch dialog box enables you to change both the context for the watch, as well as the type of the watch. You can read more about the different types of watches in the next section.

The Context section of the Add Watch dialog box lets you change the Procedure and Module where an expression is watched. This is useful if you are watching global variables, function calls, or properties in an application. By setting the watch context to the lowest level required, you can minimize updates to the watch window from the VBE.

Watch Types

You can create three different types of watches, depending on the type of debugging you are doing.

Watch Expression

The default type of watch is called Watch Expression, which simply watches an expression in the Watch window.

Break When Value Is True

If you are debugging a particularly difficult issue, there may be times when a value changes when you don't expect it to. Watches are very useful for these types of problems. Change the watch type to Break When Value Is True to inject a breakpoint when the statement being watched evaluates to True.

For example, if you have a form that accepts criteria from a user, then this form is used to pass criteria to the FindFirst method of a DAO Recordset object, such as:

```
Dim rs As DAO.Recordset  
Set rs = CurrentDb().OpenRecordset("SELECT * FROM Contacts")
```

Part I: Programming Access Applications

```
rs.FindFirst "[Last Name] = '" & Me.txtLastName & "'"
' unexpected value may return here
If (Not rs.NoMatch) Then
    MsgBox "Match found: " & rs("[First Name]") & " " & rs("[Last Name]")
End If
rs.Close
Set rs = Nothing
```

This code displays the `MsgBox` only if a match is found. But what if you always expect a match to be found? In the case where there is no match, something unexpected has occurred. To break when this occurs, you can add a watch on the `rs.NoMatch` statement with a watch type of `Break When Value Is True`. The VBE breaks into the code when this condition is `True`, as shown in Figure 4-3.

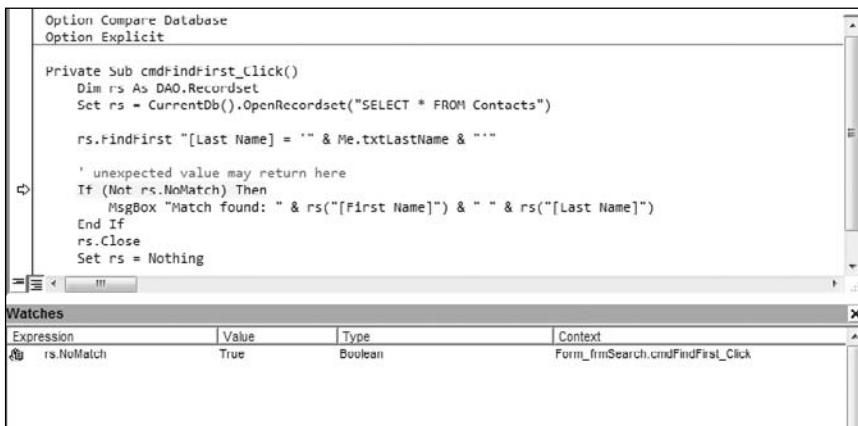


Figure 4-3

Break When Value Changes

Similar to the previous watch type, use the `Break When Value Changes` watch type to break any time a value changes. This is useful to narrow down if a global variable is being changed by another routine that you don't expect.

Using the Watches Window

The Watches window, shown in Figure 4-4, also lets you change the value of the statement being watched, enabling you to change the code path while you are debugging.

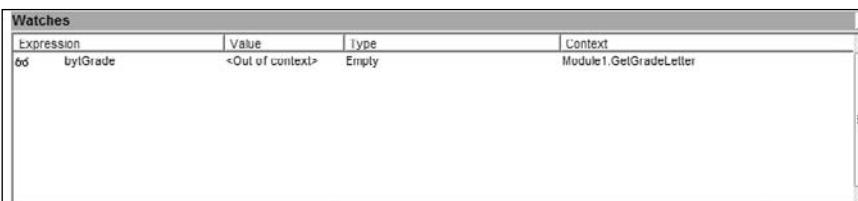


Figure 4-4

Chapter 4: Debugging, Error Handling, and Coding Practices

Let's say that you have a function called `GetGradeLetter` that calculates the grade for a student and takes an action based on that grade. The function may resemble the following code. (For the sake of this example, we're calculating the grade as a random number between 30 and 100.)

```
Public Function GetGradeLetter() As String
    ' get a random grade between 30 and 100
    Dim bytGrade As Byte
    Randomize
    bytGrade = CByte(Int((100 - 30 + 1) * Rnd + 30))

    ' assign the grade letter and take action for the grade
    Select Case bytGrade
        Case 90 To 100
            GetGradeLetter = "A"
            MsgBox "Congratulations! You got an 'A'!", , "Gold Star"
        Case 80 To 89
            GetGradeLetter = "B"
        Case 70 To 79
            GetGradeLetter = "C"
        Case 60 To 69
            GetGradeLetter = "D"
            MsgBox "Generating email to parents"      ' // Email parents
        Case Else
            GetGradeLetter = "F"
            MsgBox "Generating email to parents"      ' // Email parents
    End Select

    GetGradeLetter = GetGradeLetter & " (" & bytGrade & ")"
End Function
```

If the student receives an "A," we'll give them a gold star (a message box). If the student receives a "D" or "F," the system sends an e-mail to the student's parents. For debugging this routine, set a breakpoint on the line:

```
Select Case bytGrade
```

Next, add a watch statement on `bytGrade`, as shown in Figure 4-4. When you run the code and hit the breakpoint, you can change the value of `bytGrade`, which will determine which block of the `Select Case` statement to execute.

Using the Immediate Window

At runtime, the Immediate window is used to execute expressions in the application that are `Public` or `Private` to the module that contains running code. You can use the Immediate window to execute `Public` expressions at any time.

While debugging, you use the Immediate window for the following tasks:

- ❑ Viewing variables once — more than once, we use the Watches window or the Locals window
- ❑ Setting variable values once — more than once, we use the Watches window or the Locals window
- ❑ Running `Private` functions in the module where code is executing

Part I: Programming Access Applications

- ❑ Enumerating information such as properties or collections using `Debug.Print`
- ❑ Timing functions using `Debug.Print`
- ❑ Setting database properties

For some tips and tools designed specifically for the Immediate window, see Appendix A at the end of this book.

Using the Locals Window

When compared to the Watches window and the Immediate window, the Locals window isn't used as often. This is unfortunate because it's pretty useful.

The Locals window displays the values of all of the local variables that are declared in a given procedure. It also displays the values of variables that have module scope — that is, that are declared in the declarations section of a module. The module level variables that appear in the Locals window are only for the currently running module. It will not display variables that have module level scope in other modules.

Add the following variables to the declarations section of the module where you added the `GetGradeLetter` function earlier:

```
Private mstrPrivateString As String
Private mintPrivateInteger As Integer
Public gstrGlobalString As String
```

Display the Locals window by selecting Locals Window from the View menu of the VBE. Place a breakpoint on the last line of `GetGradeLetter` and run the `GetGradeLetter` function. You should see the Locals window, as shown in Figure 4-5.

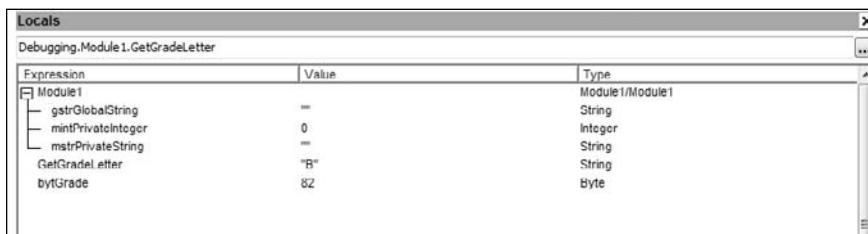


Figure 4-5

As with the Watches window, you can use the Locals window to change a value of a variable.

Viewing the Call Stack

The Call Stack window is available only in break mode and displays the list of currently running routines. Select Call Stack from the View menu to open the call stack, as shown in Figure 4-6. You can jump between routines and see where they are called by double-clicking on a given routine name.

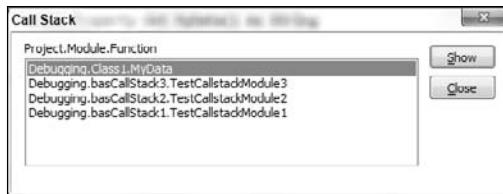


Figure 4-6

Building a Call Stack Using Code

Call stacks are interesting but there is no mechanism built into VBA to programmatically retrieve the call stack. The .NET Framework provides this functionality in the `Exception` class in a property called `StackTrace`. Wouldn't it be nice if you could do this in VBA? While there is nothing intrinsic to the language, you can build your own stack traces using a class module.

A call stack, also known as a stack trace in the .NET Framework, is based on a data structure called a *stack*. A stack is a last-in, first-out (LIFO) data structure. This means that items added to the stack last are removed first. The physical metaphor for this data structure is a stack of plates. As you add a plate to the stack, they are placed on top. These plates are subsequently the first ones to be removed.

Routines in code work the same way. When you call a function, the function is placed on the bottom of the call stack. This operation is called a *push*. When the next function is called, it is pushed to the top of the call stack. When the second function exits, it is removed from the call stack. This operation is called a *pop*. When a function exits, it is popped off of the stack and execution returns to the calling routine. This process is illustrated in Figure 4-7.

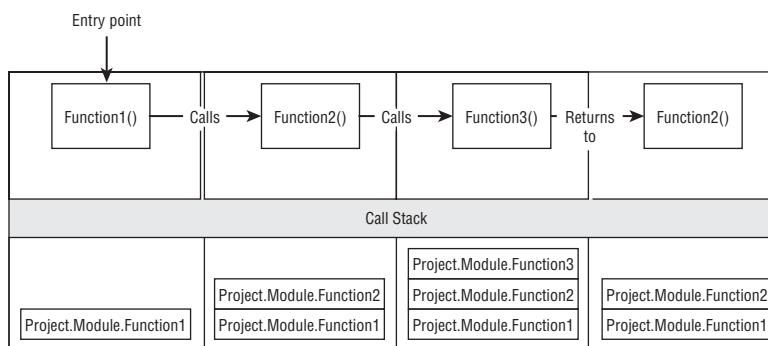


Figure 4-7

Creating the CallStack Class

You can build a stack in VBA in several ways. For simplicity, let's build the stack using a VBA `Collection` object as you did with the collection classes in Chapter 3. Follow these steps to create the stack:

1. Create a new class module called `CallStack`.
2. Add the following variable to the declarations section of the class:

```
Private m_col As VBA.Collection
```

Part I: Programming Access Applications

3. Add the following code to the Initialize event of the class:

```
Private Sub Class_Initialize()
    Set m_col = New VBA.Collection
End Sub
```

4. Stack objects should include a Push method to add items to the stack. Add the following method to the CallStack class. The Push method accepts one argument that is the name of the executing routine.

```
Public Sub Push(Statement As String)
    ' add the statement to the stack trace
    m_col.Add Statement, Str(m_col.Count)
End Sub
```

5. Stack objects should also include a Pop method to remove an item from the stack. Add the following method to the CallStack class. When you pop an item from the stack you are removing the last item. The Remove method of the VBA Collection object accepts the index of the item to remove, which is given by the Count method of the VBA Collection object.

```
Public Sub Pop()
    ' remove the top item from the stack
    If (m_col.Count > 0) Then
        m_col.Remove m_col.Count()
    End If
End Sub
```

6. You need to implement a CallStack property. This property returns the call stack to you in code. As you push an item on the stack, it is added to the end of your internal collection. Because stacks are last-in/first-out, you need to walk through the collection backwards to view the items on top.

```
Public Property Get CallStack() As String
    Dim lngCounter As Long
    Dim strCallStack As String

    ' // walk the collection and return a formatted string
    ' // since this is a stack, walk the collection backwards
    For lngCounter = m_col.Count() To 1 Step -1
        If (lngCounter > 1) Then
            strCallStack = strCallStack & m_col(lngCounter) & vbCrLf
        Else
            strCallStack = strCallStack & m_col(lngCounter)
        End If
    Next

    CallStack = strCallStack
End Property
```

Instrumenting Your Code

In order to use the CallStack object, you'll need to add calls to the Push and Pop methods throughout your code. The process of adding code that tracks the code itself is called *instrumentation*.

Chapter 4: Debugging, Error Handling, and Coding Practices

Generally speaking, you want only one call stack in your application, so add the following line of code to a standard module:

```
' global callstack
Public gobjCallStack As New CallStack
```

Next, add calls to the Push method at the top of each function you want to instrument. Also, add calls to the Pop method at the end of each function that you want to instrument. This is done in the following test code:

```
Function Function1() As Long
    ' push the current function on the stack
    gobjCallStack.Push "basCallstackTest.Function1"

    ' call another function
    Function2

    ' remove this function from the stack
    gobjCallStack.Pop
End Function

Function Function2() As Long
    ' push the current function on the stack
    gobjCallStack.Push "basCallstackTest.Function2"

    ' call another function
    Function3

    ' remove this function from the stack
    gobjCallStack.Pop
End Function

Function Function3() As Long
    ' push the current function on the stack
    gobjCallStack.Push "basCallstackTest.Function3"

    ' do some work
    MsgBox gobjCallStack.CallStack

    ' remove this function from the stack
    gobjCallStack.Pop
End Function
```

If you run the preceding Function1, you should see the following output in the Immediate window:

```
basCallstackTest.Function3
basCallstackTest.Function2
basCallstackTest.Function1
```

Notice that Function3 is on the top of the call stack and that Function1 is at the bottom of the call stack.

Programmatically Generating a Call Stack Using Error Handling

Let's say that you don't necessarily want to create another object in your code and certainly don't want to instrument your code. After all, instrumenting code is a lot of work. There is another way you can

Part I: Programming Access Applications

simulate the call stack using error handling. This requires error handling routines in the places where you are interested in receiving the call stack.

The trick is to force errors to bubble up to the highest level of the call stack. For example, if you have three routines, the lowest level function, `ThrowsAnError`, causes a runtime error. You can bubble this error up to the caller using the `Err.Raise` statement. Each time you bubble up the error, you append the name of the function that includes the error handler.

Start by adding the entry point procedure as follows. Notice that this routine calls another routine, which causes an error:

```
Sub EntryPoint()
    On Error GoTo ErrorHandler
    ' call a function
    CallsFunctionThatThrowsAnError
    On Error GoTo 0
    Exit Sub
ErrorHandler:
    ' re-raise the error so it gets bubbled up to the caller
    If (Err <> 0) Then
        Err.Raise Err, Err.Source, "Callstack" & vbCrLf & "- EntryPoint" & _
            vbCrLf & Err.Description
    End If
End Sub
```

Next, add the second routine as follows. This function actually calls the function that displays the error:

```
Function CallsFunctionThatThrowsAnError()
    On Error GoTo ErrorHandler

    ' call a function that throws an error
    ThrowsAnError

    On Error GoTo 0
    Exit Function
ErrorHandler:
    ' re-raise the error so it gets bubbled up to the caller
    If (Err <> 0) Then
        Err.Raise Err, Err.Source, "- CallsFunctionThatThrowsAnError" & _
            vbCrLf & Err.Description
    End If
End Function
```

Last, add the function that throws an error:

```
Function ThrowsAnError()
    Dim x As Integer

    ' force a div/0 error
    On Error GoTo ErrorHandler
    x = 1 / 0
    On Error GoTo 0
    Exit Function
ErrorHandler:
```

Chapter 4: Debugging, Error Handling, and Coding Practices

```
ErrorHandler:  
    ' re-raise the error so it gets bubbled up to the caller  
    If (Err <> 0) Then  
        Err.Raise Err, Err.Source, "- ThrowsAnError" & vbCrLf &  
            vbCrLf & Err.Description  
    End If  
End Function
```

Notice that in the error handler, you are raising the error that occurs using the `Err.Raise` statement. For the description of the error, you are adding additional information to indicate the routine name that caused the error. When you run the `EntryPoint` procedure, you should see the runtime error displayed in Figure 4-8.

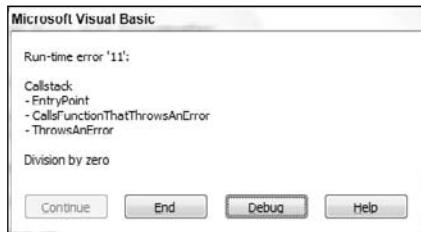


Figure 4-8

This technique requires that you have error handlers along the chain. Each time, you are pre-pending the name of the calling routine to the `Err.Description` property.

Unfortunately, there is no property built into Access or VBA that returns the name of the running routine so you have to include it in your code.

Assertions

An assertion is like a code marker for the developer to ensure that a certain condition is True. Assertions are created using `Debug.Assert` in VBA. If the statement evaluated by the `Assert` method returns `False`, the code breaks at the `Assert` method. This gives you a chance as the developer to do further investigation.

Using `Debug.Assert`

The `Assert` method of the `Debug` object takes one argument — an expression that evaluates to a Boolean result. Unlike the `Print` method of the `Debug` object, the `Assert` method is evaluated only when there is source code for a project. This statement is removed from compiled code as in the case of an ACCDE or MDE file. As such, you will not hit the `Assert` method in an ACCDE or MDE.

`Debug.Assert` was introduced in Access 2000 as a part of VBA6. Before Access 2000, you could achieve the same functionality using the `Stop` statement in a condition such as:

```
If (Expression = True) Then Stop
```

Part I: Programming Access Applications

Personally, we prefer the `Assert` method because this is the technique used in the development of Access itself. The following code shows the `Assert` method in action:

```
Sub AssertTests()
    Dim i As Integer

    ' verify that an Integer is initialized to 0
    Debug.Assert i = 0

    ' verify that the integer is greater than 0
    ' this will fail because we haven't set the
    ' value of i
    Debug.Assert i > 0

    MsgBox "AssertTests"
End Sub
```

When you run this code, the first `Assert` is not hit because VBA initializes numeric variables to 0 and the condition is `True`. The variable `i` is equal to 0. However, because we haven't changed its value, the next `Assert` expression returns `False`. This causes the code to break on the `Assert` method prior to reaching the `MsgBox`.

In an ACCDE or MDE file, the `MsgBox` runs immediately because the `Assert` expression is not evaluated.

Conditional Compilation

Using conditional compilation, you can tell the VBA compiler to ignore blocks of code. This is accomplished by defining conditional compilation constants.

Conditional compilation is useful during testing of an application, to alter the behavior of an application during development, or to conditionally include development helper routines. We use conditional compilation in the section “Writing a Better `Assert` Method.”

Private Conditional Compilation Constants

The easiest way to create a conditional compilation constant is to use the `#Const` directive at the top of a given module. These constants are then private to the module that contains them. The following is an example of a conditional compilation constant called `FDEBUG` that we have set to `True`.

```
#Const FDEBUG = True
```

Conditional compilation constants can accept a value of any type and cannot be returned. They are not recognized by the VBA compiler as a language element such as a variable.

Public Conditional Compilation Constants

Use the Properties dialog box for your project if you want a conditional compilation constant that is defined throughout your application, as shown in Figure 4-9.

Public conditional compilation constants accept only numbers and strings for their values.

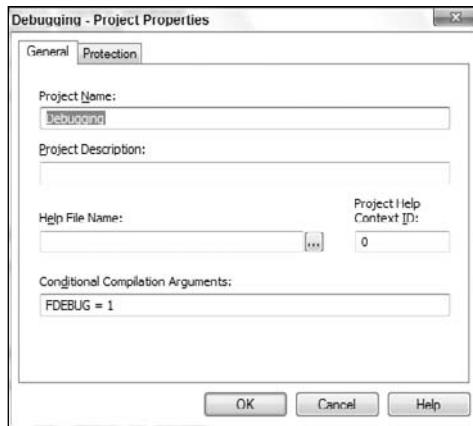


Figure 4-9

Writing a Better Assert Method

The `Assert` method is a great tool that we use frequently in our own development. However, we'd like it to have the following additional functionality:

- Custom messages for display
- The ability to display the current call stack
- A choice of whether to:
 - Ignore the assertion
 - Enter break mode to debug the problem
 - Exit the routine

To do this, we again are drawn to writing our own class module. This time, let's write our own `Debug` class called `DebugEx`. At the top of the class module, define the following conditional compilation constant:

```
#Const FDEBUG = True
```

In order to get all of the functionality we want, let's declare our own `Assert` method that will display a message using the `MsgBox` function. In order to exit the routine, this method should return a value using one of the built-in `VbMsgBoxResult` values found in VBA. Declare the method as follows:

```
Public Function Assert(stExpr As String, Optional stMessage As String) _
    As VbMsgBoxResult
    ' A more robust Assert method
    Dim fResult As Boolean
```

Because assertions typically happen only in a debug build of the application, add a conditional compilation check for the method:

```
#If FDEBUG Then
```

Part I: Programming Access Applications

Next, you need to evaluate the expression that you are given in the `stExpr` argument. To do this, use the `Eval` function in VBA:

```
fResult = CBool(Eval(stExpr))
```

If the expression evaluates to `True`, the code that called the `Assert` method keeps running. If the expression evaluates to `False`, add the rest of the code, starting with a custom message:

```
If (Not fResult) Then
    If (Len(stMessage) = 0) Then
        stMessage = "The expression [" & stExpr & "] failed."
    End If
```

Next, append the call stack using the `CallStack` class created earlier:

```
' append the callstack
stMessage = stMessage & vbCrLf & vbCrLf & "at:" & vbCrLf & _
gobjCallStack.CallStack
```

Using the message that is provided along with the call stack, display a prompt using the `MsgBox` function. The `MsgBox` uses the `Abort`, `Retry`, and `Ignore` buttons with information that tells you the behavior of the buttons.

```
' prompt
Assert = MsgBox(stMessage, _
vbAbortRetryIgnore + vbDefaultButton3 + vbCritical, _
"Assertion Failed: Abort=Quit, Retry=Debug, " & _
"Ignore=Continue")
```

Notice that the return value of `MsgBox` is given to the `Assert` method. This means that the code that calls the `Assert` method should check for `vbAbort` to exit the calling routine. If the `Assert` method returns `vbIgnore`, the calling code can continue. If, on the other hand, the return value is `vbRetry`, then you need to do something different.

The scenario you want in the case of `vbRetry` is to break on the calling code. VBA doesn't provide a direct mechanism to do this so we'll simulate it by raising an error. If the error is unhandled in the caller, then the code breaks on the call to the `Assert` method in the caller. Of course, this requires that there is no error handling in the calling code that handles the error that is raised from your assertion.

```
If (Assert = VbMsgBoxResult.vbRetry) Then
    Err.Raise -1, , "Assertion Failed: " & stMessage
End If

End If
#End If
End Function
```

The trick here is to make sure that the Error Trapping option in the VBE is set to Break on Unhandled Errors. If it is set to either Break on All Errors or Break in Class Module, the code breaks in the DebugEx class itself.

Using the DebugEx.Assert Method

To use the new `Assert` method, let's write some test code. Notice that we've declared a variable of type `VbMsgBoxResult`, which is what the `Assert` method returns. As the caller, your test code exits if you choose `Abort` in the message box.

You'll also be using a global instance of the `DebugEx` class. Add the following code to a standard module to define this instance:

```
Public gobjDebug As New DebugEx
```

Now, add the test code, as shown here:

```
Sub MyAssertTest()
    Dim ar As VbMsgBoxResult
    ' Push the function on the call stack
    gobjCallStack.Push "basAssert.MyAssertTest"

    ar = gobjDebug.Assert(1 = 0, "Verify that 1 is not equal to 0")
    If (ar = VbMsgBoxResult.vbAbort) Then
        GoTo Finally
    End If

    ResumeHere:
    MsgBox "MyAssertTest"

Finally:
    gobjCallStack.Pop
    Exit Sub
End Sub
```

When you run this routine, you should see a message box with the `MyAssertTest` routine in the call stack, as shown in Figure 4-10.

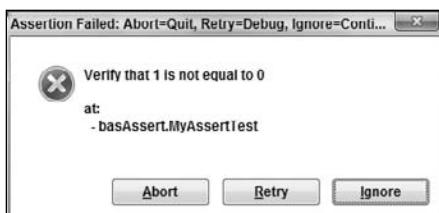


Figure 4-10

If you select `Ignore` in the `Assert` prompt, then the `MsgBox` runs and you exit the routine. If you press `Abort`, you skip the `MsgBox` and simply exit the routine. If, however, you press `Retry`, you'll see a run-time error raised, as shown in Figure 4-11.

Part I: Programming Access Applications

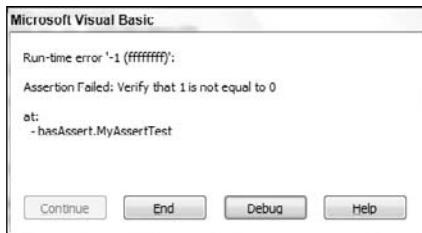


Figure 4-11

If you click Debug in the runtime error and the Error Trapping option is set to Break on Unhandled Errors, the code breaks on the call to the `Assert` method, as shown in Figure 4-12. This is similar to the way that the `Debug.Assert` method works in VBA.

```
Sub MyAssertTest()
    Dim ar As VbMsgBoxResult
    gobjCallStack.Push "basAssert.MyAssertTest"
    ar = gobjDebug.Assert(1 = 0, "Verify that 1 is not equal to 0")
    If (ar = VbMsgBoxResult.vbAbort) Then
        GoTo Finally
    End If
    ResumeHere:
    MsgBox "MyAssertTest"
Finally:
    gobjCallStack.Pop
    Exit Sub
End Sub
```

Figure 4-12

Remember that the declaration of the `Assert` method accepts an argument, `stExpr`, as a `String`, but we actually passed in the expression `1 = 0` not wrapped in quotes. In this case, VBA evaluates the expression first, and then passes in `True` or `False` to the `Assert` method. Of course, because `False` evaluates to `False`, we hit the assertion.

What this means is that you can pass functions or properties to the `Assert` method and have their return values evaluated as either `True` or `False`. Be cautious when doing this, however, because the `Assert` method evaluates expressions only when the `FDEBUG` conditional compilation constant is set.

Creating a Debug Build

When we first start testing a new version of Access, we test a debug build that includes assertions. These assertions are useful during development to track down issues. Using a debug build is a good idea for the `DebugEx` class we created earlier to ensure that certain code is available only when you are running in the special debug build.

We talk about creating builds and automating this process in more detail in Chapter 14.

There are a number of different ways to introduce the concept of a debug build of your application. Let's look at a few different techniques for creating a debug build. Of course with any of these techniques, you

Chapter 4: Debugging, Error Handling, and Coding Practices

need some mechanism for determining whether you are running the debug build. This mechanism varies depending on the technique you use.

Regardless of the technique, you might consider placing the determining function in the `DebugEx` class.

Using the Command Line to Specify a Debug Build

Perhaps the easiest way to specify that your database is a debug build is to use the command line because it doesn't call for major changes to the application. Access accepts a command-line switch called `/cmd`. This switch is used to specify command-line arguments to your application. Using the command line, you might start your application using something such as:

```
msaccess.exe "<PathTo>\YourDatabase.accdb" /cmd DEBUG
```

Then, using the `Command` function in VBA, the following code determines whether the database is a debug build.

```
Public Property Get IsDebugBuildCL() As Boolean  
    IsDebugBuildCL = (InStr(Command(), "DEBUG") > 0)  
End Property
```

Using a Table to Create a Debug Build

Another technique is to add a table that indicates whether the database is a debug build. This can be as simple as a single table with a single Yes/No field called `IsDebug` that you set to True. A user, however, may discover this. To prevent discovery, Access enables you to create your own system objects using the prefix `USys`. For example, let's say that you created a table called `USysDebugInfo` with one field: `IsDebug`. By default, this table is hidden in the Access Navigation Pane. To show the table, select Show System Objects as shown in the Navigation Options dialog box in Figure 4-13.

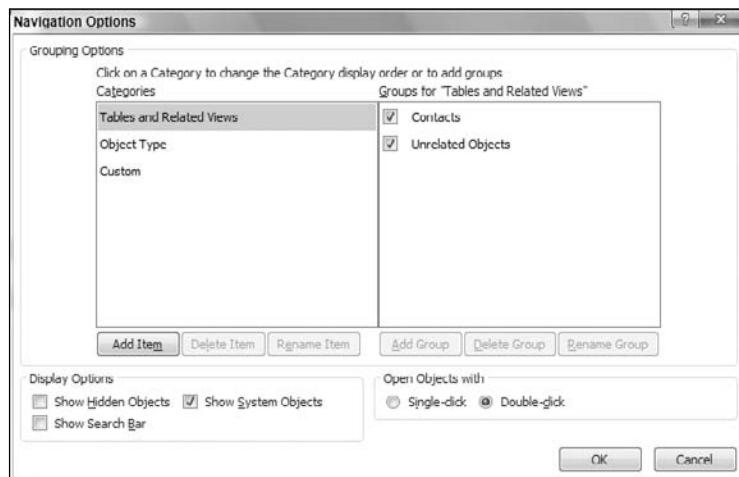


Figure 4-13

Part I: Programming Access Applications

In this case, you can easily determine whether you are running a debug build by opening a recordset object that points to the `USysDebugInfo` table, as shown in the following code:

```
Public Property Get IsDebugBuildRS() As Boolean
    ' using the recordset
    Dim rs As DAO.Recordset2

    On Error Resume Next
    Set rs = CurrentDb().OpenRecordset("USysDebugInfo")

    ' if the table does not exist, this is not debug
    If (Err = 3078) Then
        IsDebugBuildRS = False
        Set rs = Nothing
        Exit Property
    ElseIf (Err <> 0) Then
        ' unexpected error
        Stop
    End If

    ' check the field - if the field does not exist, this is not debug
    IsDebugBuildRS = (rs!IsDebug)
    If (Err = 3265) Then
        IsDebugBuildRS = False
    End If
    On Error GoTo 0

    ' cleanup
    rs.Close
    Set rs = Nothing
End Property
```

Using Conditional Compilation to Create a Debug Build

Earlier, you looked at two ways to define conditional compilation constants in your application. To create a debug build that applies to the entire application, use a public conditional compilation constant defined in the Properties dialog box for the VBA project. To determine whether the constant is set, you'll need to use the `#If` directive, as shown in the following code:

```
Public Property Get IsDebugBuildCC() As Boolean
    #If FDEBUG Then
        IsDebugBuildCC = True
    #Else
        IsDebugBuildCC = False
    #End If
End Property
```

Using a Database Property to Create a Debug Build

The technique we tend to use most often is to create a database property that indicates whether the database is a debug build. The easiest place to set this property is in the DAO `Database` object, as shown in the following code:

```
Public Sub SetDebug(fDebug As Boolean)
    On Error GoTo SetDebugErrors
```

```
' set the property
CurrentDb.Properties("DEBUG") = fDebug
Exit Sub

SetDebugErrors:
If (Err = 3270) Then
    ' property does not exist, create it
    CurrentDb.Properties.Append _
        CurrentDb.CreateProperty("DEBUG", dbBoolean, fDebug)
    Exit Sub
ElseIf (Err <> 0) Then
    ' unexpected error
    Stop
End If
End Sub
```

Use the following code to determine whether a database is a debug build using the database property.

```
Public Property Get IsDebugProp() As Boolean
On Error Resume Next
IsDebugProp = CurrentDb.Properties("DEBUG")

' property does not exist - not debug
If (Err = 3270) Then
    IsDebugProp = False
ElseIf (Err <> 0) Then
    ' unexpected error
    Stop
End If
End Property
```

Typically, you won't distribute the debug version of your application to your users. But, in the event we forget to remove a debug hook from the application, sometimes we combine the command-line approach with the database property to determine whether a database is a debug build. That way, users would have to know to set this property but also to open the database using the /cmd switch.

Error Handling

Error handling is another important part of application development. Again, we personally really like error handling because we like to add error handling features to our applications. This lets us keep in touch with the user by way of logging or sending notifications. We'll touch on those subjects shortly. First, here are some tips that we use frequently to make development easier.

Handling Errors Inline

You can do two basic types of error handling in an application. With the first technique, you use the `On Error Goto` statement and provide complete error handling for a given procedure. This is recommended in most cases. The second technique is to handle errors inline, which is how errors are handled in VBScript. This technique is useful for cases where one possible error can be thrown from a particular line of code.

Part I: Programming Access Applications

The general format for handling errors inline is as follows:

```
On Error Resume Next          ' Turn off error handling
x = 1/0                      ' Cause an error, in this case #DIV/0
If (Err = 11) Then
    MsgBox "Cannot divide by zero" ' Do something with the error
End If
On Error Goto 0               ' Resume error handling
```

Using inline error handling, you wrap a statement in an `On Error` block that consists of disabling error handling and then resuming error handling.

Categorizing Errors

If you've ever looked at a Windows header file or worked with constant values provided by an ActiveX control, you've probably noticed that constants often tend to be grouped together by their names. You can use this same technique for consistency in your applications. Consider errors that you might find in a customer and order tracking application:

```
' Error codes
Const ERR_INVALID_CUSTOMER As Long = vbObjectError + 513
Const ERR_INVALID_ORDERNUM As Long = vbObjectError + 514
Const ERR_INVALID_SHIPDATE As Long = vbObjectError + 515
Const ERR_INVALID_ITEM      As Long = vbObjectError + 516
```

By grouping constants together, you can find them easier when using IntelliSense.

System Error Codes and HRESULTS

The Windows SDK includes a header file called `winerror.h` that defines many of the error codes used in Windows itself. These constants include both regular system errors and HRESULT codes. An HRESULT is an error code typically used with COM programming that contains additional information encoded in the error number. Because these errors are intrinsic to Windows, you often see API functions that return these errors. However, you can also use these constants in your applications to provide consistency with other Windows applications.

System Error Codes

The following are some error codes defined by Windows that you might find useful in your applications:

```
Public Enum SystemErrorCodes
    NO_ERROR = 0           ' No error
    ERROR_FILE_NOT_FOUND = 2 ' Cannot find file
    ERROR_PATH_NOT_FOUND = 3 ' Cannot find path
    ERROR_ACCESS_DENIED = 5 ' Access denied
    ERROR_INVALID_DATA = 13 ' Invalid data
    ERROR_INVALID_DRIVE = 15 ' Cannot find specified drive
    ERROR_BAD_NETPATH = 53 ' Bad network path
    ERROR_BAD_NET_NAME = 67 ' Network name not found
    ERROR_FILE_EXISTS = 80 ' File already exists
    ERROR_INVALID_PARAMETER = 87 ' Invalid parameter
```

```
    ERROR_OPEN_FAILED = 110      ' Cannot open file or device
    ERROR_DIR_NOT_EMPTY = 145    ' Directory not empty
    ERROR_BAD_PATHNAME = 161     ' Path is invalid
    ERROR_INVALID_FLAGS = 1004   ' Invalid flags
End Enum
```

HRESULT Error Codes

The following are some HRESULT errors defined by Windows that you might find useful in your applications:

```
Public Enum HRESULT
    S_OK = 0                      ' No error
    E_NOTIMPL = &H80004001        ' Method not implemented
    E_UNEXPECTED = &H8000FFFF    ' Unexpected error
    E_ACCESSDENIED = &H80070005  ' Access denied
    E_INVALIDARG = &H80070057   ' Invalid argument
    E_FAIL = &H80004005         ' General failure
End Enum
```

Error codes defined by Windows do not necessarily correspond to those in VBA. For example, Windows defines error code 5 as “Access Denied,” but VBA defines error code 5 as “Invalid procedure call or argument.” Use the FormatMessage API function to return the error message for an error code defined by Windows. This function is provided in Chapter 2.

Creating an Error Handler Class

VBA does not have a global error handler or exceptions as C++ and C# do. As a result, error handlers in VBA often contain duplicated code to display error messages. In addition, as you’ve already seen, there is no way to dynamically determine the current routine that is currently running when an error occurs. To solve both of these problems and make handling errors easier, create a class module called `ErrorHandler`.

The `ErrorHandler` class will include the `ILog` interface that is defined in the section “The `ILog` Interface.” Compile the code at the end of that section to avoid compilation errors.

Enumerations

After you’ve created the `ErrorHandler` class, add the following enumerations to the class. The first enumeration, `LogTypes`, defines the types of logging that are available to the application. Logging errors are discussed in more detail in the section “Logging Errors.” The second enumeration, `ErrorLevels`, is used to indicate the level of an error. The constants found in the `ErrorLevels` enumeration correspond to icons used in the `MsgBox` function in VBA and will be used later.

```
' Enums
Public Enum LogTypes
    LogTextFile = 0          ' This will be the default if m_logtype is not set
    LogDatabase = 1
End Enum
Public Enum ErrorLevels
    Default = 0
```

Part I: Programming Access Applications

```
Critical = vbCritical      ' 0x10
High = vbExclamation     ' 0x30
Information = vbInformation ' 0x40
End Enum
```

Private Data

You'll need some private data in the class so add the following code to the declarations section of the `ErrorHandler` class:

```
' Private data and constants
Private Const SW_SHOW      As Long = 5
Private m_ProcName    As String          ' procedure name
Private m_ObjName     As String          ' object name
Private m_Message      As String          ' custom error message
Private m_AutoLog      As Boolean         ' log error message
Private m_Error        As VBA.ErrObject   ' VBA.Err
Private m_ErrLevel     As ErrorLevels     ' error level
Private m_LogType      As LogTypes        ' Log type
Private m_Log          As ILog            ' ILog object
```

Properties

Next, you'll add some properties. The `ErrorHandler` class contains the properties listed in the table that follows.

Property Name	Data Type	Description
AutoLog	Boolean	Gets or sets a flag that indicates whether errors are logged automatically when displayed.
ComputerName	String	Gets the name of the computer where the error occurred.
ErrorLevel	ErrorLevels	Gets or sets the level of the error.
ErrorObject	VBA.ErrObject	Gets or sets a VBA ErrObject that contains the error that occurred.
LogObject	ILog	Gets an ILog object that performs error logging in the <code>ErrorHandler</code> class.
LogType	LogTypes	Gets or sets the type of the log for the <code>ErrorHandler</code> class.
Message	String	Gets or sets a custom message to display.
ObjectName	String	Gets or sets the name of the object where the error occurred.
ProcName	String	Gets or sets the name of the procedure where the error occurred.
UserName	String	Gets the name of the user logged in to the computer.

Chapter 4: Debugging, Error Handling, and Coding Practices

Add the following code to implement these properties:

```
' Properties
Public Property Get ComputerName() As String
    ComputerName = Environ$("COMPUTERNAME")
End Property
Public Property Get UserName() As String
    UserName = Environ$("USERNAME")
End Property
Public Property Let ObjectName(RHS As String)
    m_ObjName = RHS
End Property
Public Property Get ObjectName() As String
    ObjectName = m_ObjName
End Property
Public Property Let ProcName(RHS As String)
    m_ProcName = RHS
End Property
Public Property Get ProcName() As String
    ProcName = m_ProcName
End Property
Public Property Get ErrorObject() As VBA.ErrObject
    Set ErrorObject = m_Error
End Property
Public Property Set ErrorObject(objError As VBA.ErrObject)
    Set m_Error = objError
End Property
Public Property Get ErrorLevel() As ErrorLevels
    ErrorLevel = m_ErrLevel
End Property
Public Property Let ErrorLevel(level As ErrorLevels)
    m_ErrLevel = level
End Property
Public Property Get Message() As String
    Message = m_Message
End Property
Public Property Let Message(stMessage As String)
    m_Message = stMessage
End Property
Public Property Get LogType() As LogTypes
    LogType = m_LogType
End Property
Public Property Let LogType(LogType As LogTypes)
    m_LogType = LogType
End Property
Public Property Get AutoLog() As Boolean
    AutoLog = m_AutoLog
End Property
Public Property Let AutoLog(fAutoLog As Boolean)
    m_AutoLog = fAutoLog
End Property
Public Property Get LogObject() As ILog
End Property
```

Part I: Programming Access Applications

Notice that we've left the `LogObject` property empty because we need a helper method to create the log object that is defined later.

Methods

The `ErrorHandler` class will contain the two methods listed in the table that follows.

Method Name	Description
Display	Displays an error message to the user.
SendError	Sends an e-mail message to the developer containing information about the error.

Add the following code to implement these methods, starting with the `Display` method:

```
' Methods
Public Sub Display()
    Dim stMsg As String

    stMsg = stMsg & "Message: " & m_Message & vbCrLf
    stMsg = stMsg & "Procedure: " & m_ProcName & vbCrLf
    stMsg = stMsg & "Object : " & m_ObjName & vbCrLf

    ' error
    If (Not (m_Error Is Nothing)) Then
        stMsg = stMsg & vbCrLf & "Error      : " & m_Error.Number & vbCrLf
        stMsg = stMsg & "Text      : " & m_Error.Description & vbCrLf

        ' log automatically
        If (m_AutoLog) Then
            LogObject.LogError m_Error
        End If
    End If

    ' show
    MsgBox stMsg, m_ErrLevel
End Sub
```

The `Display` method, implemented in the preceding code, creates a formatted message using the information stored in the class. It echoes the custom message, procedure name, and object name where the error occurred. If a valid `ErrObject` is stored in the `m_Error` variable, it also includes information about the error and logs the error if needed.

Now let's implement the `SendError` method. This method builds an e-mail message and creates the e-mail using the `ShellExecute` API function as discussed in Chapter 2. Notice that there is a helper function called `EncodedCrLf`. This function creates a repeating string of encoded carriage-returns/line-feeds.

Chapter 4: Debugging, Error Handling, and Coding Practices

Sending notification might be useful for very critical errors in your application. You might extend this method in the future to include additional logic to send only certain error messages.

```
Public Function SendError() As Long
    ' sends email about an error message
    Dim stMsg As String

    stMsg = "mailto:someone@somewhere.com"

    If (m_Error Is Nothing) Then
        stMsg = stMsg & "&subject=Application Error"
    Else
        stMsg = stMsg & "&subject=Application Error: " & m_Error.Number
    End If

    stMsg = stMsg & "&body=An application error has occurred" & EncodedCrLf(2)
    stMsg = stMsg & "Message = " & m_Message & EncodedCrLf(1)
    stMsg = stMsg & "Object Name = " & m_ObjName & EncodedCrLf(1)
    stMsg = stMsg & "Procedure Name = " & m_ProcName & EncodedCrLf(1)

    If (Not m_Error Is Nothing) Then
        stMsg = stMsg & "Error number = " & m_Error.Number & EncodedCrLf(1)
        stMsg = stMsg & "Error description = " & m_Error.Description
    End If

    ShellExecute Application.hWndAccessApp, _
        "open", stMsg, _
        vbNullString, vbNullString, _
        SW_SHOW
End Function

' Helpers
Private Function EncodedCrLf(n As Integer) As String
    Const HTML_CRLF As String = "%0D%0A"
    Dim i As Integer

    ' make sure n is greater than 0
    If (n <= 0) Then Exit Function

    For i = 1 To n
        EncodedCrLf = EncodedCrLf & HTML_CRLF
    Next
End Function
```

Logging Errors

It's a fact of life — users sometimes do things in an application that are unexpected. For example, a user might enter data on a form out of order, or select an option in a dialog box without selecting another option first. For such times, you should have good error handling in place to trap any errors that might occur. To further help you with troubleshooting, it might be helpful to log unexpected errors in the application. This section describes some techniques for logging errors for your application.

Part I: Programming Access Applications

The `ILOG` Interface

Using an interface, you can easily change the type of logging in the application without changing the code. You'll define a class module called `ILOG` that will serve as an interface for error logging, and then you'll create two classes that implement this interface.

Start by creating a new class module called `ILOG`. Remember from Chapter 3 that by convention, interface classes begin with the capital letter "I." This interface defines the behaviors that you need when error logging.

The `ILOG` interface has one property called `Size` that is defined as follows. This property returns the size of the error log.

```
Public Property Get Size() As Long  
End Property
```

The methods we define in the `ILOG` interface are shown in the table that follows.

Method Name	Description
<code>.LogError</code>	Logs the error message.
<code>Purge</code>	Clears the error log.
<code>Show</code>	Displays the error log.

Add the methods to the `ILOG` interface:

```
Public Sub LogError(objErrorHandler As ErrorHandler)  
End Sub  
Public Sub Purge()  
End Sub  
Public Sub Show()  
End Sub
```

The `.LogError` method accepts an instance of the `ErrorHandler` class created earlier because the `ErrorHandler` class contains all of the relevant data that will be logged.

Logging to a Database Table

One of the easiest ways to log is to simply log to a table in the database, but this can grow the size of the database more than you need. Thus, in the next section, you'll log errors to a text file.

To log errors to a table, create a new table in the database called `USysErrorLog`. Using the `USys` prefix, Access treats this table as a user-defined system table. Create the table to store the property values as follows. This table also stores information about the error, so add properties for the error as well.

Chapter 4: Debugging, Error Handling, and Coding Practices

Field Name	Data Type	Field Size
Id	AutoNumber	
ComputerName	Text	50
UserName	Text	50
ObjectName	Text	64
ProcName	Text	255
ErrorCode	Number	Long Integer
ErrorMessage	Memo	
ErrorSource	Text	255

Next, create a new class called `DatabaseLog` and implement the `ILog` interface as follows:

Implements `ILog`

To add the necessary private data for the class, include the following code to the declarations section:

```
' Constants  
Const TABLE_NAME As String = "USysErrorLog"  
' Private data  
Private m_rs      As DAO.Recordset2
```

Because you are logging to a table, open the recordset when an instance of the class is created using the `Initialize` event of the class:

```
Private Sub Class_Initialize()  
    Set m_rs = CurrentDb().OpenRecordset(TABLE_NAME)  
End Sub
```

When the class is destroyed, close the recordset and clean up the `m_rs` variable. Add the following code to the `Terminate` event of the class:

```
Private Sub Class_Terminate()  
    If (Not m_rs Is Nothing) Then  
        m_rs.Close  
        Set m_rs = Nothing  
    End If  
End Sub
```

Now it's time to start implementing the `ILog` interface. The `Size` property for the `DatabaseLog` class returns the number of records in the `USysErrorLog` table. As you develop your application

Part I: Programming Access Applications

you can use this property to limit the number of records in the error log, or even send the error log automatically.

```
Private Property Get ILog_Size() As Long
    ILog_Size = m_rs.RecordCount
End Property
```

Let's implement the methods, starting with the `Purge` method. This method will delete all the records from the `USysErrorLog` table:

```
Private Sub ILog_Purge()
    CurrentDb.Execute "DELETE * FROM [" & TABLE_NAME & "]"
End Sub
```

Next, implement the `Show` method. This method opens the `USysErrorLog` table to view the log.

```
Private Sub ILog_Show()
    DoCmd.OpenTable TABLE_NAME
End Sub
```

Last, implement the `.LogError` method. This is the method that does most of the work. For the `DatabaseLog` class, first check that the `ErrorObject` property of the `objErrHandler` parameter is not `Nothing`, and if it is then there is nothing to log. Then add a record to the `USysErrorLog` table using the recordset.

```
Private Sub ILog.LogError(objErrHandler As ErrorHandler)
    ' add a record : make sure there is an error
    If (objErrHandler.ErrorObject Is Nothing) Then
        MsgBox "There is no error to log", vbExclamation
        Exit Sub
    End If

    With m_rs
        .AddNew
        !ComputerName = objErrHandler.ComputerName
        !UserName = objErrHandler.UserName
        !ObjectName = objErrHandler.ObjectName
        !ProcName = objErrHandler.ProcName
        !ErrorCode = objErrHandler.ErrorObject.Number
        !ErrorMessage = objErrHandler.ErrorObject.Description
        !ErrorSource = objErrHandler.ErrorObject.Source
        .Update
    End With
End Sub
```

Logging to a Text File

Logging error messages to a table in the database is convenient, but there is a potential it will grow the database. To avoid this, let's create an implementation of the `ILog` interface that logs to a text file. Start by creating a new class module called `TextLog` and implement the `ILog` interface.

Implements `ILog`

Chapter 4: Debugging, Error Handling, and Coding Practices

Again, you need some private data in the class. The `TextLog` class logs to a file called `ApplicationErrors.log` in the same directory as the database as determined by `CurrentProject.Path` in the Access object model:

```
' Constants
Const FILE_NAME As String = "\ApplicationErrors.log"
' Private data
Private m_FilePath    As String
Private m_FileNum     As Integer
```

Open the log file in the `Initialize` event of the class using the `Open` statement in VBA. Open the file for `Append` so that you don't lose any errors along the way:

```
Private Sub Class_Initialize()
    ' calculate the path to the file
    m_FilePath = CurrentProject.Path & FILE_NAME

    ' open the file
    m_FileNum = FreeFile()
    Open m_FilePath For Append As #m_FileNum
End Sub
```

Close the file in the `Terminate` event of the class:

```
Private Sub Class_Terminate()
    Close #m_FileNum
End Sub
```

For the `TextLog` class, the `Size` property returns the size of the log file.

```
Private Property Get ILog_Size() As Long
    ILog_Size = LOF(m_FileNum)
End Property
```

Now, implement the methods of the `ILog` interface, starting with the `Purge` method. To purge the log file, you need to close it, delete it, and then re-open it:

```
Private Sub ILog_Purge()
    Close #m_FileNum
    Kill m_FilePath
    Open m_FilePath For Append As #m_FileNum
End Sub
```

Next, implement the `Show` method, which launches Notepad to open the log file:

```
Private Sub ILog_Show()
    ' close the file so the data is available
    Close #m_FileNum
    Shell "notepad.exe " & m_FilePath, vbMaximizedFocus

    ' reopen the file
    Open m_FilePath For Append As #m_FileNum
End Sub
```

Part I: Programming Access Applications

To log errors to the file using the `.LogError` method, use the `Print` statement in VBA. Then create a comma-delimited file that can easily be imported into either Access or Excel:

```
Private Sub ILog_LogError(objErrorHandler As ErrorHandler)
    Dim stErr As String

    ' add a record : make sure there is an error
    If (objErrorHandler.ErrorObject Is Nothing) Then
        MsgBox "There is no error to log", vbExclamation
        Exit Sub
    End If
    ' create the error string
    stErr = objErrorHandler.ComputerName & ","
    stErr = stErr & objErrorHandler.UserName & ","
    stErr = stErr & objErrorHandler.ObjectName & ","
    stErr = stErr & objErrorHandler.ProcName & ","
    stErr = stErr & objErrorHandler.ErrorObject.Number & ","
    stErr = stErr & objErrorHandler.ErrorObject.Description & ","
    stErr = stErr & objErrorHandler.ErrorObject.Source
    ' print to the file
    Print #m_FileNum, stErr

    ' close and re-open so the file is available right away
    Close #m_FileNum
    Open m_FilePath For Append As #m_FileNum
End Sub
```

Creating an Instance of the ILog Class

In order to use an `ILog` object throughout your application, you'll need some way to create an instance of either the `DatabaseLog` or `TextLog` class. To do this, create a new helper module called `Factory` with the following function:

```
Public Function CreateLog(LogType As LogTypes) As ILog
    Select Case LogType
        Case LogTypes.LogDatabase
            Set CreateLog = New DatabaseLog
        Case LogTypes.LogTextFile
            Set CreateLog = New TextLog
    End Select
End Function
```

The `CreateLog` function accepts an argument of type `LogTypes`, which was defined as an `Enum` earlier in the `ErrorHandling` class. The function returns an `ILog` interface but instantiates either a `DatabaseLog` or `TextLog` depending on the value of the `LogType` parameter. This lets you change the log by passing in a different parameter to this function.

Adding a new type of log is also easy. Let's say that later on you decide to log to XML or to SQL Server. You can create a new class that implements the `ILog` interface, and then change the `CreateLog` function to return the new type of log.

Chapter 4: Debugging, Error Handling, and Coding Practices

Now, let's go back to the `LogObject` property in the `ErrorHandler` class. Because you have the `CreateLog` method, you can use the following code to create the `ILog` object:

```
Public Property Get LogObject() As ILog
    ' create the log
    Set m_Log = Factory.CreateLog(m_LogType)
End Property
```

The property first checks the instance of `m_Log` that is defined in the `ErrorHandler` class. If it hasn't been set, it creates an `ILog` instance using the `CreateLog` helper function that was just created.

Remember that the `ErrorHandler` class also stores the type of the log to create in a variable called `m_LogType`. Because this variable is an `Enum` value, its underlying data type in VBA is a `Long`. VBA initializes `Long` values to 0. Therefore, if the `LogType` property in the `ErrorHandler` class has not been set, `m_LogType` defaults to 0, which corresponds to `LogTypes.LogTextFile`. As a result, the `LogObject` property returns an instance of the `TextLog` class unless you set `ErrorHandler.LogType` to `LogTypes.LogDatabase`.

Using the Error Handling Class and Logging

Now that you've written all of the logging classes and the `ErrorHandler` class, let's take a look at how to use them. To use the `ErrorHandler` class throughout an application, declare an instance of it as a `Public` variable in a standard module.

```
Public gobjErrorHandler As New ErrorHandler
```

Next, use the following test code to test the classes. Notice that you are setting properties of the `ErrorHandler` class to indicate the routine that failed and using the `LogObject` property to log an error to the database:

```
Sub DbErrorHandlerTest()
    On Error GoTo E_HANDLER

    Dim x As Integer
    ' Cause an Overflow error
    x = 100000

    Finally:
        Exit Sub

    E_HANDLER:
        Set gobjErrorHandler.ErrorObject = Err
        gobjErrorHandler.LogType = LogDatabase
        gobjErrorHandler.LogObject.LogError gobjErrorHandler
        gobjErrorHandler.Message = "Unexpected error!"
        gobjErrorHandler.ErrorLevel = Critical
        gobjErrorHandler.ObjectName = "basLogTests"
        gobjErrorHandler.ProcName = "ErrorHandlerTest"
        gobjErrorHandler.Display

        Resume Finally

End Sub
```

Coding Practices

As developers, we tend to adopt our own styles over time. These styles are often dictated by personal preference or influenced by the company we're working for. Quite often these styles adapt and change as you grow in your career or learn new skills. For example, we've been doing C# work on a routine basis for several years and this is reflected when we write VB or VBA code now. In this section, we discuss some coding practices that have worked for us over the years. We'll refrain, however, from going into the details of our own style because styles often vary.

Readability

Go ahead — admit it. Have you ever written a piece of code only to come back to it later and wonder what's going on? We have — on more than one occasion. When writing code, you should assume that someone else will read your code, even if that someone else is you two years down the road. This assumption will help you write code that is readable and easy to understand.

Comments

Using comments is a useful tool when writing code to help you understand algorithms or logic, as well as have a clear sense of what the code is doing. While it is possible to over-comment, there are several ways you can use comments effectively.

Module Headers

Assuming that you are grouping code logically into modules, use comments at the top of the module that describes the purpose of the module. Several tools are available that can insert module headers for you:

- ❑ MZ-Tools, available at www.mztools.com
- ❑ VBA Code Commenter, available with Microsoft Office 2000 Developer

The following is an example of comments at the top of a module:

```
' Module      : ErrorHandler
' DateTime   : 4/1/2007 15:45
' Author     : Rob
' Purpose    : Defines the error handling class
' History    : 4/1/2007 Created
'              : 4/2/2007 Added simple properties
'              : 4/4/2007 Refactored EncodedCrLf helper function
'              : 4/8/2007 Added the LogObject property
```

Procedure Headers

Similar to module headers, a procedure header is used to describe a procedure in a module. Because procedures can accept parameters or return values, it is also useful to describe the parameters and whether or not it returns a value.

The following is an example of a procedure header:

```
' Procedure  : ILog_LogError
' DateTime   : 4/8/2007 15:50
```

Chapter 4: Debugging, Error Handling, and Coding Practices

```
' Author      : Rob
' Arguments   : objError    - VBA.ErrObject that is the current Err
'               : ErrorLevel - error level for the specified error
' Returns     : Nothing
' Purpose     : Logs an error to a table in the current database
' -----
Private Sub ILog_LogError(objError As VBA.ErrObject, ErrorLevel As ErrorLevels)
    ' // Remaining code goes here...
End Sub
```

Other Comments

It's often more obvious *what* a piece of code is doing than *why* it's there. For example, consider the following `Enum` definitions.

Example 1

```
Public Enum LogTypes
    LogTextFile = 0           ' Initialize the enumerated values to 0
    LogDatabase = 1
End Enum
```

Example 2

```
Public Enum LogTypes
    LogTextFile = 0           ' This will be the default if m_LogType is not set
    LogDatabase = 1
End Enum
```

It's pretty obvious from the first example that the value `LogTextFile` is being set to 0, but why? The second example explains that this value is 0 because it corresponds to the default value of a variable called `m_LogType`. These types of comments are often more useful than comments that describe what is happening.

At the end of the day, you should write the code that is most readable to you and your audience if anyone else is going to read your code. Perhaps you'll have requirements to add a comment next to each variable such as:

```
Private m_ProcName As String          ' procedure name
Private m_ObjName As String            ' object name
Private m_Message As String           ' custom error message
Private m_Error As Boolean            ' log error message
Private m_Error As VBA.ErrObject     ' VBA.Err
Private m_ErrorLevel As ErrorLevels   ' error level
Private m_LogType As LogTypes         ' Log type
Private m_Log As ILog                 ' ILog object
```

Or, above each variable name like this:

```
' procedure name
Private m_ProcName As String
' object name
Private m_ObjName As String
```

Part I: Programming Access Applications

```
' custom error message
Private m_Message As String
' log error message
Private m_LogError As Boolean
' VBA.Err
Private m_Error As VBA.ErrObject
' error level
Private m_ErrLevel As ErrorLevels
' Log type
Private m_LogType As LogTypes
' ILog object
Private m_Log As ILog
```

In either case, writing code that is descriptive not only improves readability but can save you debugging time in the future!

Formatting

Code that spans over multiple lines can be either harder to read or easier to read depending on how you indent the subsequent lines. Unlike some other languages such as C++ and C#, VBA requires using a line continuation character to indicate that the code spans multiple lines. The line continuation character is a space followed by an underscore (_) at the end of a line.

Using a line-continuation character with good spacing can make code easier to read. This is particularly useful when you have a long list of arguments such as with an API declaration.

Consider the following Declare statements of the ShellExecute API function:

Example 1

```
Private Declare Function ShellExecute Lib "shell32.dll" Alias "ShellExecuteA" _
    (ByVal hWnd As Long, ByVal lpOperation As String, ByVal lpFile As String, _
    ByVal lpParameters As String, ByVal lpDirectory As String, _
    ByVal nShowCmd As Long) As Long
```

Example 2

```
Private Declare Function ShellExecute Lib "shell32.dll" Alias "ShellExecuteA" _
    (ByVal hWnd As Long, _
    ByVal lpOperation As String, _
    ByVal lpFile As String, _
    ByVal lpParameters As String, _
    ByVal lpDirectory As String, _
    ByVal nShowCmd As Long) As Long
```

In the second example, we've aligned the parameters on separate lines to make the declaration more readable but it results in larger procedures. Sometimes, however, this is acceptable.

Naming Conventions

Consistent use of naming conventions goes a long way toward making your code more readable. Developers who inherit a piece of code often use naming conventions to help understand the code.

Chapter 4: Debugging, Error Handling, and Coding Practices

This doesn't mean, however, that developers should assume that a variable declared such as `iMyValue` is an `Integer` in VBA. Perhaps the developer who originally declared this variable had a background in C or C++ where an `int` is 4 bytes, which corresponds to a `Long` in VBA.

Just as variables should follow a consistent naming convention, your objects should also follow a consistent naming convention. This makes the code look nicer when used in IntelliSense, as well as more understandable.

Version Control

Version control systems enable you to share development work with multiple developers. The Access 2007 Developer Extensions includes integration with the Microsoft Visual SourceSafe (VSS) version control system. This integration allows you to check out forms, reports, macros, and modules from a VSS database. Using version control can make development slightly more difficult as it adds some more process to the development task, but is highly beneficial.

When you check in a database to VSS, you have, in effect, a backup of your database that can be re-created. This is really helpful! In addition, most version control systems including VSS contain the ability to revert to a previous version of an object. Let's say that you make a change to a form that you aren't necessarily happy with. By using a version control system you can undo those changes by simply reverting them to a previous version of the object. Be careful that you don't lose other changes when you revert!

Refactoring

Refactoring is the process of improving code with the intent of making it more readable, maintainable, or efficient. Refactoring should not change the results of the application, only make it better. In VBA, refactoring is a manual process, but there are several areas where refactoring is helpful.

Modules

In projects that need to be completed quickly, it's sometimes easier to add a module instead of adding a procedure to an existing module. This can lead to code that is spread out over various places instead of grouped where it should be, or even worse, tightly coupled.

We once wrote an application that had a very specific task — to search a group of databases in a specified directory for specific information. As with many applications, we had to write this quickly and didn't have much time to plan it out. The application grew quickly to search for more data aside from the original data we were looking for. The original flow of the code resembled the flowchart shown in Figure 4-14.

As you can see, the functions that were called initially, `BuildList` and `Search`, were tightly coupled into the module called `basSpecificSearch1`. As new searches were added, we wrote new modules to implement the search, but we had to call them from `basSpecificSearch1.Search`, which meant that one of the main pieces of code had to change each time. This was bad. To fix this, we refactored the `BuildList` and `Search` functions into a module that we named `MAIN` because it contains the entry point routine. Next, we created an interface called `ISearch` that contained a single method called `Search`. To avoid confusion, we renamed `MAIN.Search` to `MAIN.ExecSearch` because this method calls the `ISearch.Search` method. Now, any time we want to run a search, we write a new class module that implements `ISearch` and we can pass it to the `BuildList` function. The end result resembles the flowchart in Figure 4-15.

Part I: Programming Access Applications

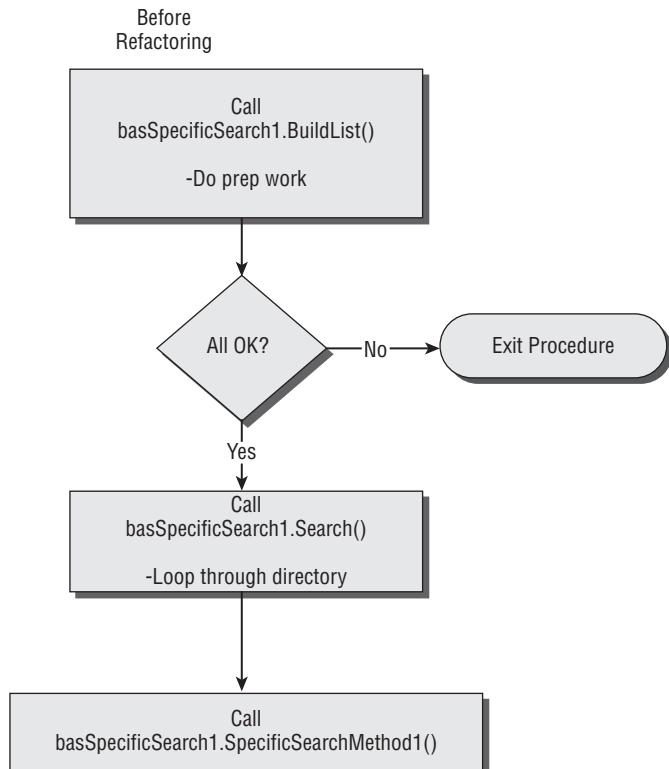


Figure 4-14

By refactoring you can create code that is more flexible, and easier to understand and maintain.

Procedures

Refactoring of procedures can also help you create code that is more readable. When looking for opportunities to refactor a procedure, look for repetition in a given routine. For example, consider the original version of this method:

```
Public Function SendError() As Long
    ' sends email about an error message
    Dim stMsg As String

    stMsg = "mailto:someone@somewhere.com"
    stMsg = stMsg & "?cc = " & Environ("USERNAME")

    stMsg = stMsg & "&body=An application error has occurred%0D%0A%0D%0A"
    stMsg = stMsg & "Message = " & m_Message & "%0D%0A"
    stMsg = stMsg & "Object Name = " & m_ObjName & "%0D%0A"
    stMsg = stMsg & "Procedure Name = " & m_ProcName & "%0D%0A"
    ' // Remaining code goes here
End Function
```

Chapter 4: Debugging, Error Handling, and Coding Practices

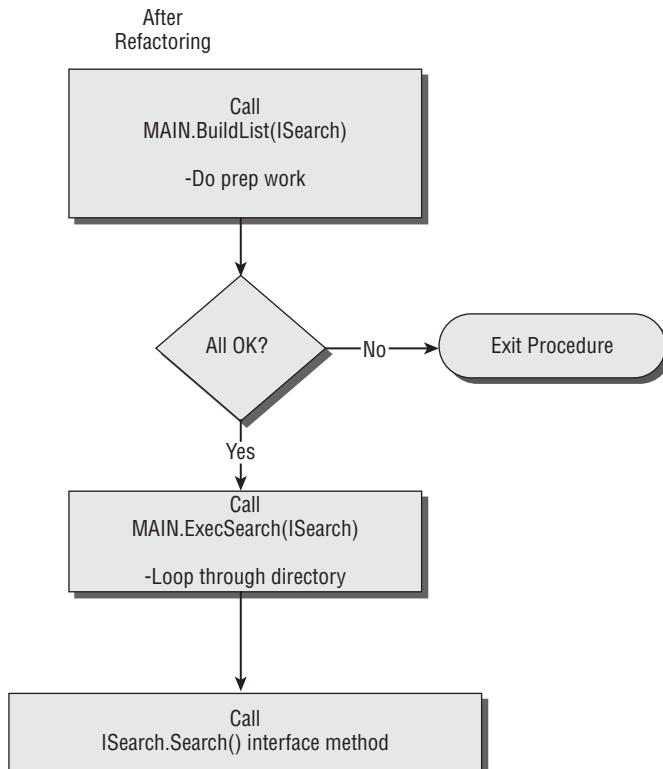


Figure 4-15

It looks like the encoded characters %0D%0A are repeated quite a bit. By refactoring these characters into a separate helper function, the code becomes flexible and more readable:

```
Public Function SendError() As Long
    ' sends email about an error message
    Dim stMsg As String

    stMsg = "mailto:someone@somewhere.com"
    stMsg = stMsg & "?cc = " & Environ("USERNAME")

    stMsg = stMsg & "&body=An application error has occurred" & EncodedCrLf(2)
    stMsg = stMsg & "Message = " & m_Message & EncodedCrLf(1)
    stMsg = stMsg & "Object Name = " & m_ObjName & EncodedCrLf(1)
    stMsg = stMsg & "Procedure Name = " & m_ProcName & EncodedCrLf(1)
    ' // Remaining code goes here
End Function

Private Function EncodedCrLf(n As Integer) As String
    Const HTML_CRLF As String = "%0D%0A"
    Dim i As Integer

    ' make sure n is greater than 0
    If (n <= 0) Then Exit Function
```

Part I: Programming Access Applications

```
For i = 1 To n
    EncodedCrLf = EncodedCrLf & HTML_CRLF
Next
End Function
```

For even more flexibility, the `EncodedCrLf` function can be refactored further to accept the pattern of characters as an argument to the function:

```
Private Function Repeat(stPattern As String, n As Integer) As String
    Dim i As Integer

    ' make sure n is greater than 0
    If (n <= 0) Then Exit Function

    For i = 1 To n
        Repeat = Repeat & stPattern
    Next
End Function
```

Code Reviews

The last coding practice we'll talk about is code reviews. If you are using a version control system such as VSS, you might ask a colleague to review your changes before you check in. For a form, report, macro, or module, you can generate your changes using the `SaveAsText` command in the Access object model. The signature for this method is as follows:

```
SaveAsText ObjectType As AcObjectType, ObjectName As String, FileName As String
```

Then, using a tool such as `windiff.exe` or VSS itself, you can view the differences between the changes you made to the object and the object that is currently checked in.

If you are not using a version control system or are an independent consultant, you might consider asking a friend or contact an online forum to review a piece of code for you. In these cases, the logic contained in a block of code might not always be obvious (unless you can also find someone with domain expertise), but you can probably find someone to help you understand a piece of code if needed, or improve its performance, or make suggestions about other ways to accomplish the same task. Many excellent online forums are available, such as www.utteraccess.com and the Microsoft Access newsgroups that begin with `microsoft.public.access`.

Summary

Love it or hate it, at some point you're probably going to have to debug an application or write some error handling code. The code in this chapter is designed to help you do so more effectively and even add features to your applications such as error logging.

Over the course of this chapter, you saw that:

- ❑ By using class modules, even for routine code such as error handling, you can greatly simplify coding throughout an application.
- ❑ The Visual Basic Editor provides several tools to help you debug your applications.

Chapter 4: Debugging, Error Handling, and Coding Practices

- ❑ Assertions are a powerful technique for finding bugs in an application.
- ❑ By employing coding practices such as using consistent style, commenting your code, and refactoring, you can create code that is readable and highly maintainable.

The next chapter begins Part II, which is all about manipulating data within an application. The chapter will show you how to effectively parse several different types of data, create HTML, and go deeper into file input/output.

Part II: Data Manipulation

Chapter 5: Parsing Data

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Chapter 7: Managing Data

5

Parsing Data

When working with data in a database, you tend to take for granted the structured nature of the data. Well-designed databases have schema that provide for granular storage of data. Well-designed relational databases are normalized so that duplicate data is minimized.

But lots of times, you need to work with less structured data that exists in external files of various formats, or that may exist in poorly designed databases that are not sufficiently granular. You often need to extract bits and pieces of data locked away in these files and scary databases. This chapter is all about how to work with this less structured data.

In this chapter, you learn how to do the following:

- Read data from and write data to external text files
- Split data apart and combine it together
- Use the VBA string handling functions to isolate data within strings
- Perform token replacements within strings
- Identify strategies for splitting up complex items into their component parts
- Create HTML files from code

File Input/Output

We've looked at countless examples of working with data contained in tables in a Jet or ACE database, but it's almost as easy to work with data contained in many text files. VBA provides many statements designed specifically to provide access to data files stored on your computer or network.

In this section, you look at a couple of different ways to access data stored in text files.

Part II: Data Manipulation

Using VBA I/O statements

VBA provides 29 different statements to help you work with data files on your disk or network. We will look at the subset of statements shown in the table that follows that provides the bulk of the functionality you will need to create new data files or open and read data stored in files already existing on your disk or network.

VBA Input/Output Statement	Purpose
FreeFile	Acquire a file handle
Open	Create or open a file
Close	Close a file
Input, Line Input	Retrieve data from a file
Print, Write	Write data to a file
EOF	Test for end of file
FileLen	Determine the size of a file
Name	Rename a file
FileCopy	Copy a file to a new location

To see a complete list of the Input/Output functions provided in the VBA library, search online help for "Input and Output Keywords."

In this section, you look at techniques for working with data stored as delimited or fixed-width text files. VBA also provides functions to work with data stored in a binary format, but because each binary file has a different format specific to the task at hand, it is beyond the scope of this book to look at how to crack these files.

The Importance of Error Handling

When working with file I/O, error handling is very important, because there are many things than can go wrong. When working with a DAO recordset, once you have debugged your code, objects rarely change names, and aside from the occasional user error, objects rarely disappear from your database. But when you start working with external files, all sorts of problems can occur.

Probably the most common error that occurs when working with the file I/O statements is good ol' number 53: File Not Found. Even if you've never previously programmed file I/O operations in one of your applications, you've almost certainly seen an Error 53 in one of the applications you've used.

Another common error is the Permission Denied error (number 70). You will find this when you have file locking problems (you need write access to a file but another process has the file locked).

Our File I/O toolbox contains several named constants for the most common file I/O errors, to make our error handlers more readable.

```
Public Const ERR_BADFILENAMEORNUMBER As Long = 52
Public Const ERR_FILENOTFOUND As Long = 53
Public Const ERR_FILEALREADYOPEN As Long = 55
Public Const ERR_BADRECORDLENGTH As Long = 59
Public Const ERR_PERMISSIONDENIED As Long = 70
Public Const ERR_PATHNOTFOUND As Long = 76
```

Each procedure we write that involves file I/O *always* has an error handler, and we urge you to follow this practice. In this way, you can handle the inevitable error with grace, and help your users figure out how to resolve the problem. No “Error 70” for your application! Your users will know that another program has the file Shippers.txt opened and locked, and that they need to close the file in that program before they can continue!

Reading an Existing Text File

The Open statement is how you instruct VBA to create a new file or open an existing file. You pass the name of the file, and an abundance of arguments specifying how you want to read or write data to the file.

One of the required arguments is a file number. This number, also referred to as a handle, is used by most of the VBA Input/Output statements to identify which file you want to work with. Although you can pass in an arbitrary number for the file number, if that number is already being used to refer to another file, bad things can happen (up to and including corrupting your data).

The FreeFile VBA function will generate a number that is guaranteed to work with any of the VBA I/O statements that require a file number. We strongly recommend that you always use the following pattern when working with VBA I/O statements:

1. Create a variable to hold the file number.
2. Call FreeFile and assign the results to your variable.
3. Call the Open statement, passing the variable for the file number argument.
4. Perform some actions on the text file (read data, write data).
5. Call the Close statement to close the file.

All of the sample code in this section will use this pattern. Lets look at a simple function that opens a text file (`Employees.csv`, a comma-delimited text file that contains all of the data from the Northwind Employees table), reads each row out of the table, and writes the data into the debug window:

```
Public Sub ReadRecordsFromFile(fileName As String)
    Dim hFileSrc As Long      '// Variable to hold the file number
    Dim item As String         '// Variable to hold the row of data from the file

    On Error GoTo Err_Handler
```

We create a variable named `hFileSrc` to hold the file number (or file handle), implementing the first step of the pattern. Here, we’re also creating a variable (named `item`) to hold the data that we are going to read in from the data file.

Part II: Data Manipulation

Also, when working with file I/O functions, it is very important to enable robust error handling. File I/O has many opportunities for errors to occur: you may refer to a filename that does not exist, or you may not have permissions to read to or write to the file, or the file may be locked by another user thus temporarily blocking you from accessing the file. Without error handling to handle these situations, you run the risk of your code failing without providing any information to your user why the code has failed.

```
'// obtain the next available file handle
hFileSrc = FreeFile

'// open the file
Open fileName For Input As #hFileSrc
```

Here, we implement steps two and three from the pattern we described previously. We call the `FreeFile` function to obtain the next available file number and cache that value in our variable. Then, we call the `Open` statement, and pass the variable name for the File Number argument. Here, we request file mode `Input`, which allows us to read data from the file. We will look at some of the other file modes shortly.

```
'// Loop through each row in the text file
Do While Not EOF(hFileSrc)
    '// read the row into the item variable
    Line Input #hFileSrc, item

    '// write the row to the debug window
    Debug.Print item
Loop
```

Next, we loop through the text file one line at a time, copying the data into the `item` variable using the `Line Input` statement. This is just one of the I/O statements that allows you to read the content of a file. `Line Input` reads one line at a time, starting from the first character of the file and continuing until it finds either a carriage return (ASCII code 13 or the `VBA.Constants.vbCr` constant) or carriage return+line feed (ASCII codes 13 and 10, or the `VBA.Constants.vbCrLf` constant). `Line Input` discards the CR or CRLF and copies the remainder of the data into the variable that you pass to the function (in this case, the `item` variable). This `DO` loop implements Step 4 of our pattern.

Notice two subtle differences when looping through a file using the VBA I/O statements compared to looping through a recordset using DAO. First, instead of using the `EOF` property on a recordset to test whether we are at the end of the recordset, we pass the file number to the `EOF` VBA function. We are performing the identical task — determining if we have reached the end of our data — but we use two different techniques depending upon the technology we are using.

Also, in DAO, it's essential to call the `MoveNext` method on our recordset; otherwise, you get stuck on the same row and never reach the end of the data. This is not necessary when working with the VBA I/O statements that perform sequential access to a file. Whenever you read data from the file, the current position pointer is automatically adjusted to point to the next chunk of data. There is no equivalent to the `MoveNext` method when working with sequential access to data files.

In our cleanup section, we close the file using the `Close` statement (Step 5 of the pattern).

```
Cleanup:
    '// Always remember to close your files!
    On Error Resume Next
    Close #hFileSrc
```

Again, we pass in the file number using the `hFileSrc` variable. The `Close` statement will perform all the necessary operations to release our hold on the file, and then will release the file number for reuse. Because we are cleaning up, we temporarily disable error handling; `Close` is a very safe call and highly unlikely to trigger an error, but in case an earlier error has done something to interfere with the shutdown process in this function, we want to hide any errors here. We use an `On Error Goto 0` statement in the `Terminate` block to restore default error handling behavior before we leave the function:

```

Terminate:
    On Error GoTo 0
    Exit Sub

Err_Handler:
    Dim msg As String
    Dim title As String

    '// Calculate the error message
    Select Case Err.Number
        Case ERR_FILENOFOUND:
            msg = "File not found: " & fileName
        Case ERR_PERMISSIONDENIED:
            msg = "Permission Denied on " & fileName
        Case Else:
            msg = Err.Description
    End Select
    '// Display the error
    MsgBox msg, vbCritical, "Error Number " & Err.Number

    '// Clear the error, then go to cleanup to close the file
    Err.Clear
    GoTo Cleanup:

End Sub

```

Following the `Cleanup` block, we have a standard `Terminate` block where default error handling behavior is restored, and the procedure is terminated.

We also include an error handler, which uses a `Select Case` block to specifically trap two common errors: File Not Found errors (which occur when the filename you specify does not exist), and Permission Denied errors (which occur when you attempt to claim a read or write lock and another process has the file open). We have included a `Case Else` block to trap any other error that may occur (in which case we simply report the error condition). After reporting the error to the user, we go to the `Cleanup` block to close the file and then terminate the procedure.

Run this procedure and pass the valid filename of a text file. Open the Immediate Window and you will see the contents of the text file appear, one line at a time, in the Immediate window.

In addition to the `Line Input` statement, you can use a couple of other statements to retrieve data in slightly different ways.

Opening a File for Random Access

If your text files contain data in a fixed width format, where each and every row is of a known size, and where each and every field always starts and ends at a known position, you may wish to open your file

Part II: Data Manipulation

in random access mode, and use a user-defined type object to load each field into the type. This eliminates the need to parse the row that you read: Each field in the text file is loaded into a field in your user defined type automatically.

The following example is based on a fixed-width file we created by exporting the Shippers table in Northwind.mdb. Select the Shippers table, and then choose to export to text file. In the wizard, select the fixed-width option. Accept all the default values in the wizard. The resulting text file should look like this:

1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931

This file is also available from this book's Web site at www.wrox.com/174029.

The ShipperID field is 11 characters long, and appears in positions 1 to 11. The ShipperName field is 40 characters long, and appears in positions 12 to 51. Finally, the Phone field is 24 characters long, and appears in positions 52 to 75. Also, there is a CRLF on each row, for a total of 77 characters per row.

In the code, we want to create a user defined type that we'll call `ShipperItem`. This type exactly mimics the layout of the data in the text file. The code to create this type looks like this:

```
Public Type ShipperItem
    Id As String * 11
    ShipperName As String * 40
    Phone As String * 24
End Type
```

We define a field for each item, and specify the length of each field. Even though the ID field in the Shippers table is defined as a long integer, because we are working with text data in a text file, we're defining it as a fixed-length string in this type object. We can coerce this value to a long integer later.

We have two options for dealing with the CRLF characters: We can add an extra field (we would probably choose to name it `CRLF` or `Padding`) to the user-defined type, but we don't like this option because the CRLF is irrelevant to the Shipper object. It's an artifact of the file system and the way the text file was created, and is completely unrelated to the Shipper information. Instead, we will do some simple math when we call the `Open` statement to indicate that our records are padded with CRLF characters.

Let's look at the function to read Shipper information from the text file. It is very similar to the function we created in the previous section.

```
Public Sub ReadShipperFromFile(fileName As String)
    Dim hFileSrc As Integer      '// Variable to hold the file number
    Dim shipper As ShipperItem   '// ShipperItem to hold data
    Dim curItem As Long          '// counter

    On Error GoTo Err_Handler

    '// obtain the next available file handle
    hFileSrc = FreeFile
```

Here we define the `hFileSrc` variable to hold our file number. Instead of a string, we create an instance of the `ShipperItem` type to hold the data we are going to read from each line of the text file. Finally, we declare a long integer that we will use as a record pointer so we know which record we are on as we traverse the text file.

```
' // open the file for random access
Open fileName For Random As #hFileSrc Len = (Len(shipper) + Len(vbCrLf))
```

The `Open` statement has changed in two ways. First, instead of opening the file `For Input`, we have requested random access mode (`For Random`). Random access mode requires that our data exist in data records of fixed sizes, so that we can always calculate where a record will be in the file. It's a pretty simple algorithm, too! If we want the *nth* record in the file, we need to start at character position

$$((n-1) * \text{RecordLength}) + 1$$

For example, we determined earlier that each record in the `shippers.txt` file is 77 characters in length. So, if we want to go to the first record, we can calculate the starting position of the record using this algorithm.

$$((1-1) * 77) + 1 = 1$$

The starting position for the first record is at position 1. If we're looking for the second record:

$$((2-1) * 77) + 1 = 78$$

The starting position for the second record is at position 78.

When we request Random Access to a file, we need to inform the `Open` statement of how long each record is, so that it can calculate which chunk of data to return when you ask for a random record. We do this with the `Len` argument to the `Open` statement. We could have passed the hard coded value 77 because we know that's how long the record is. But that is a bit inflexible. If we later change the data in the text file (and the corresponding layout of the `ShipperItem` user defined type), we'd have to remember to change the 77 to the new length. To provide flexibility (at the price of a slight performance hit), we use the `Len` VBA function to calculate the length of the variable (`shipper`) that will be used to store the record. We also need to include the length of the `CRLF` in each row. So the simple math we referred to earlier is:

$$\text{Len}(\text{shipper}) + \text{Len}(\text{vbCrLf})$$

In our example, this results in

$$75 + 2 = 77$$

so 77 is passed as the length of the record, exactly what we want. If the length of the `ShipperItem` user-defined type changes at some point in the future, the call to the `Open` statement does not need to change.

When working with random access data, you want to return a complete record rather than a "line" of data. `Line Input` can return only a line of data, so we need another method to fetch the record. That method is the `Get` statement. `Get` takes the file number, the record number we want to retrieve, and some buffer to store the data in. In this example, the buffer is the instance of the `ShipperItem` user-defined type, although

Part II: Data Manipulation

you could also pass in any variable that is large enough to contain the data. (For example, you could also pass in a standard string variable, but then you would have to parse the string to identify the individual fields within.)

To calculate the record number, we simply increment the `curItem` variable by one each time through the loop. Then, we pass this variable in the record number argument, along with the `hFileSrc` and `shipper` variables.

```
'// Loop through each row in the text file
Do While Not EOF(hFileSrc)
    '// increment the row counter
    curItem = curItem + 1
    '// read the row into the shipper variable
    Get #hFileSrc, curItem, shipper
```

After the call to `Get`, the user-defined type is filled in with the data. We can simply refer to the fields in the user-defined type to see the data parsed automatically.

```
'// write the shipper variables to the debug window
Debug.Print Trim$(shipper.Id), _
    Trim$(shipper.ShipperName), _
    Trim$(shipper.Phone)
Loop
```

The remainder of the function is identical to the one we created earlier, except that we add one additional error check, for `ERR_BADRECORDLENGTH`. This is an error you won't see when working with sequential access data, but will definitely encounter when configuring your code to work with random access data. We also change the `msg` assignments to demonstrate one use of the `StringFormat` function.

```
Cleanup:
    '// Always remember to close your files!
    On Error Resume Next
    Close #hFileSrc

Terminate:
    On Error GoTo 0
    Exit Sub

Err_Handler:
    Dim msg As String
    Dim title As String

    '// Calculate the error message
    Select Case Err.Number
        Case ERR_FILENOTFOUND:
            msg = "File not found: " & filename
        Case ERR_BADRECORDLENGTH:
            msg = "Bad record length. " & _
                "Check your calculation in the Open statement!"
        Case ERR_PERMISSIONDENIED:
            msg = "Permission Denied on '" & filename & "'"
        Case Else:
            msg = Err.Description
```

```

End Select
'// Display the error
MsgBox msg, vbCritical, "Error Number " & Err.Number

'// Clear the error, then go to cleanup to close the file
Err.Clear
GoTo Cleanup:
End Sub

```

Writing Data

When you want to write data to a text file, you use the same basic pattern as for reading data from a file, except that in the Open statement, you will specify a file mode and an access mode that allows you to write data to the file.

To demonstrate writing data to a file, we will modify the ReadShipperFromFile function to copy the shipper data from a fixed-width text file to a new comma-delimited text file. The new function will be called CopyShipperDataToFile, and it accepts two arguments: the name of the source file (shippers.txt) and the name of the file to create (shippers.csv).

```

' Procedure : CopyShipperDataToFile
' Purpose   : Read data from from one text file, write to another
' Arguments : srcFileName - source file
'             destFileName - destination file
'

Public Sub CopyShipperDataToFile(srcFileName As String, destFileName As String)
    Dim hFileSrc As Integer
    Dim hFileDest As Integer
    Dim shipper As ShipperItem '// ShipperItem to hold data
    Dim curItem As Long        '// counter
    Dim fileName As String

    On Error GoTo Err_Handler

    '// open the source file for random access
    hFileSrc = FreeFile
    fileName = srcFileName
    Open fileName For Random As #hFileSrc Len = (Len(shipper) + Len(vbCrLf))

```

Because we will be reading from one file, and writing to another file, we need two variables to hold file numbers. Here we declare hFileSrc and hFileDest. We call `FreeFile` to get the first available file handle and store that value in hFileSrc. Then we open the source file.

Note that we store the source filename into the filename variable before we call the Open statement. This is so we are able to report the correct filename in the error handler in case an error occurs while opening the file. If we didn't want to report the actual filename that generated an error, we could omit this assignment, and specify the srcFileName variable in the Open statement.

Next, we again call `FreeFile` and store this file number to the destination file handle variable.

```

'// open the destination file for sequential write
hFileDest = FreeFile
fileName = destFileName

```

Part II: Data Manipulation

```
'// if the file exists, delete it
If (Dir(fileName) = filename) Then Kill fileName
Open fileName For Output Access Write Lock Write As #hFileDest
```

We have to make the second call to `FreeFile` after we actually open the first file. If we had tried to assign `hFileDest` immediately after we assigned `hFileSrc`, we would see something odd: `hFileSrc` and `hFileDest` have the same value! Clearly this is not what we want. When we call `FreeFile`, it reports the next available file handle, but it does not reserve that value until the file is actually opened.

To make sure we always get fresh data, we call the `Kill` VBA function to delete the destination file if it exists. Then, we call the `Open` statement requesting write access and a write lock.

The `Do` loop begins identically to the code in the previous function. We loop through each row in the text file, incrementing the row counter and using the `Get` statement to load the `shipper` variable.

```
'// Loop through each row in the text file
Do While Not EOF(hFileSrc)
    '// increment the row counter
    curItem = curItem + 1
    '// read the row into the shipper variable
    Get #hFileSrc, curItem, shipper
```

Next, we use the `Write` statement to write data as comma delimited values.

```
'// write the shipper data to the destination file
Write #hFileDest, _
    Trim$(shipper.Id), Trim$(shipper.ShipperName), Trim$(shipper.Phone)
```

Our cleanup block closes both files.

```
Loop
Cleanup:
    '// Always remember to close your files!
    On Error Resume Next
    Close #hFileSrc
    Close #hFileDest
```

We have added two error conditions to our error handler. We now handle `ERR_FILEALREADYOPEN` (which is thrown in the case we attempt to open a file that was previously opened and not properly closed) and `ERR_PATHNOTFOUND` (which is returned if we attempt to create a file in a folder that does not exist).

```
Terminate:
On Error GoTo 0
Exit Sub

Err_Handler:
Dim msg As String
Dim title As String

    '// Calculate the error message
Select Case Err.Number
```

```

Case ERR_FILENOTFOUND:
    msg = "File not found: " & fileName)
Case ERR_FILEALREADYOPEN:
    msg = "File is already open: " & fileName)
Case ERR_BADRECORDLENGTH:
    msg = "Bad record length. " & _
        "Check your calculation in the Open statement!"
Case ERR_PERMISSIONDENIED:
    msg = "Permission Denied on " & fileName)
Case ERR_PATHNOTFOUND:
    msg = "Path not found for " & fileName)
Case Else:
    msg = Err.Description
End Select
'// Display the error
MsgBox msg, vbCritical, "Error Number " & Err.Number

'// Clear the error, then go to cleanup to close the file
Err.Clear
GoTo Cleanup:
End Sub

```

We can change this function to output data in a fixed-width format by making two changes to the code. First, we change the file mode in the Open statement that creates the destination file from Output to Random:

```
Open fileName For Random Access Write Lock Write As #hFileDest
```

Second, we change the Write statement to a Put statement that writes the user-defined type to the text file:

```
'// Fixed-width output
Put #hFileDest, , shipper
```

We leave the record number argument empty. When no specific record number is passed to the Put statement, the data is written to the next record position. Because we are looping through our recordset, this default behavior is desirable. Because our user-defined type does not contain a field for a CRLF character sequence, the resulting text file will *not* have carriage-return/line feed characters at the end of each line. When reading data from the resulting text file, we would not need to make the record length adjustment as we did when working with the CRLF-terminated text file.

Using FileSystemObject

The VBA I/O functions are quick and easy to use, but they show their age. They were designed when object-oriented programming techniques had not achieved broad acceptance. Programmers who are accustomed to object-oriented programming often prefer to use the Windows Script Host Object Model to perform file operations. This library exposes an object-oriented programming model for the Windows file system.

You must first add a reference to the Windows Script Host Object Model. From the Visual Basic Environment, choose \Rightarrow Tools \Rightarrow References, and search for the Windows Script Host Object Model.

Part II: Data Manipulation

The following function provides equivalent functionality to the `CopyShipperDataToFile` function created earlier:

```
Sub CopyShipperDataWithFSO(srcFileName As String, destFileName As String)
    Dim fso As New IWshRuntimeLibrary.FileSystemObject

    Dim streamIn As IWshRuntimeLibrary.TextStream
    Dim streamOut As IWshRuntimeLibrary.TextStream
    Dim srcFile As IWshRuntimeLibrary.File
    Dim item As String
```

We create an instance of the `FileSystemObject` using the new keyword. This automatically instantiates the object when it is declared.

We declare two `TextStream` variables — one each for the input stream and the output stream. A `File` variable is declared that will be an accessory to the input file.

```
On Error GoTo Err_Handler:

'// open srcFile
Set srcFile = fso.GetFile(srcFileName)
Set streamIn = srcFile.OpenAsTextStream(IOMode.ForReading, _
    Tristate.TristateUseDefault)
```

First we use the `GetFile` method to obtain a `File` object representing the source file. Then we open a text stream on the file, and explicitly request read access (using the `IOMode.ForReading` enum value).

The `Tristate` value is an odd beast. As its name implies, it is an enumeration with three values, as shown in the table that follows. What these three values represent differs depending upon the method in which it is used. In the case of the `OpenAsTextStream` method, the `Tristate` argument determines whether a file is opened as an ASCII file or a Unicode file.

Tristate Enumeration Value	Meaning
<code>Tristate.TristateUseDefault</code>	Allow the operating system to determine file encoding
<code>Tristate.TristateFalse</code>	Open the file as an ASCII file
<code>Tristate.TristateTrue</code>	Open the file as an Unicode file

Here, we delete the destination file if it already exists.

```
'// delete destFile if it already exists
If fso.FileExists(destFileName) Then fso.DeleteFile destFileName

'// Create destination file as a Unicode file
Set streamOut = fso.OpenTextFile(destFileName, _
    IOMode.ForWriting, True, Tristate.TristateTrue)
```

Compare this to the VBA code: The `FileSystemObject` has a `FileExists` method that returns true if the specified file exists. We call the `DeleteFile` method only after we have verified the file exists. In the VBA code, we blindly call `Kill` on the file.

The output file stream is created by calling the `OpenTextFile` method on the `FileSystemObject` instance. We set the `IOMode` argument to `ForWriting` because we want to write data to this file. We pass `True` to the `Create` argument, to indicate that the file should be created if it does not already exist (which it won't because we just deleted it). Finally, we pass `Tristate.TristateTrue` to explicitly request that we create a Unicode file. (We could just as easily pass either of the values — we just like Unicode files.)

```
// Loop through each row in the source file
Do While Not streamIn.AtEndOfStream
    // copy the row to item variable
    item = streamIn.ReadLine

    // write the item variable to the output stream
    streamOut.Write item & vbCrLf
Loop
```

As in the earlier function, we loop through each row in the source file. Instead of an `EOF` function, we check the `AtEndOfStream` property: While this property is false, we still have more data to process. We call the `ReadLine` method of the input stream to grab one line at a time and cache that value into the `item` string variable. Finally, we pass `item` and a `vbCrLf` to the `Write` method of the output stream to write the row to the output file.

```
Cleanup:
On Error Resume Next
// close streams, and set all objects to nothing to free memory
streamIn.Close
Set streamIn = Nothing

streamOut.Close
Set streamOut = Nothing

Set srcFile = Nothing
```

Because streams and files are objects, we need to explicitly set them to `Nothing` in order to tell the operating system to release the memory the objects are using. We do this in the cleanup block so that we will always free the memory, even if an error occurs in our function.

```
Terminate:
On Error GoTo 0
Exit Sub

Err_Handler:
MsgBox Err.Description, vbCritical, Err.Number
Err.Clear
GoTo Cleanup:
End Sub
```

Finally, the function contains our standard `Terminate` block and a simple error handler.

Part II: Data Manipulation

Determining Whether a File Is ANSI, Unicode, or UTF-8

Windows allows you to create text files in an occasionally bewildering variety of formats. Let's look at the most common formats, and discuss how you can programmatically distinguish between them.

In the earliest days of personal computing, all text files consisted of single-byte characters. There were different encodings and code pages, but each file consisted of 1-byte characters. This proved extremely limiting for many reasons, not the least of which is that with only 256 possible characters, representing international character sets and graphical symbols was difficult, and in many cases, impossible.

Unicode was developed to alleviate the limitations of the 256 item character set, but it comes in several different flavors. To identify which variant of Unicode was used to create a file, the Byte Order Mark (BOM, also known as a *preamble*) was introduced. If a file contains a BOM, it will exist in the very first few character positions of the document. The BOM is merely a flag — a set of bytes that uniquely identifies which encoding was used to create the document. You can write some very simple VBA to read the BOM and return the encoding of the document.

The following table lists some of the characteristics of the most common encodings.

Character Set/Encoding	Byte Order Mark (BOM)	Code Page	Size of character
ANSI	None	1252 ANSI — Latin 1	1 byte
UTF-8	EF BB BF	65001	1–3 bytes depending upon Unicode value
UTF-16 Little Endian			
(low byte first; default for Windows)	FF FE	1200	2 bytes
UTF-16 Big Endian			
(high byte first; default for Macintosh, PowerPC)	FE FF	1201	2 bytes
UTF-32 Little Endian	FF FE 00 00	12000	4 bytes
UTF-32 Big Endian	00 00 FE FF	12001	4 bytes

The `ByteOrderMark` enumeration provides an `enum` value for each of the common encodings listed in this table, as well as values for some of the more common error conditions that you may encounter. Positive values (generally corresponding to number of bits in a character) are used for valid encodings, whereas negative numbers indicate an error condition.

```
' Enum      : ByteOrderMark
' Purpose   : List the ByteOrderMark values detected by GetPreamble
' Returns   : Negative values indicate an error return from GetPreamble
```

```

Public Enum ByteOrderMark
    None = 0                      '// No ByteOrderMark present, May be ANSI or DOS
    UTF8 = 8
    UTF16 = 16                     '// Little-endian (Windows default)
    UTF16BE = 17                   '// Big-endian (Macintosh/PPC default)
    UTF32 = 32                     '// Little-endian
    UTF32BE = 33                   '// Big-endian
    Undetermined = -1
    EmptyFile = -2
    FileNotFound = -53
End Enum

```

If you work with documents that use encodings not listed here, that include a BOM that you can interrogate, you can extend this enum with the new encodings, and update the GetPreamble function to detect the new BOM.

```

' Procedure : GetByteOrderMark
' Purpose   : Determine the ByteOrderMark of a specified text file
' Arguments  : fileName - path to the file to interrogate
' Returns   : A ByteOrderMark enumeration value. Error conditions are signaled by
'             negative values, corresponding to the error code received.
' Remarks   : The following Byte Order Marks are recognized by this function
'             UTF32BE - 00 00 FE FF
'             UTF8     - EF BB BF
'             UTF16BE - FE FF
'             UTF16LE - FF FE
'             UTF32LE - FF FE 00 00
'             If one of the following Byte Order Marks is not detected and no
'             other error condition has been encountered (empty file, file
'             not found, etc.), ByteOrderMark.None is returned.
Public Function GetByteOrderMark(fileName As String) As ByteOrderMark
    Dim bom As ByteOrderMark

    On Error GoTo Err_Handler

    '// Set a default return value
    bom = ByteOrderMark.Undetermined

```

We declare the function, specifying a string `fileName` argument. The function will return a `ByteOrderMark` enumeration value. We set the `bom` variable to the `Undetermined` enum value; this is our default value in case we are unable to determine a more specific Byte Order Mark.

```

'// check for empty file
If FileLen(fileName) = 0 Then
    bom = ByteOrderMark.EmptyFile
    GoTo Terminate:
End If

```

An empty file doesn't have a Byte Order Mark, so we return the `EmptyFile` enumeration value.

```

Dim hFile As Long
Dim bytes(4) As Byte

```

Part II: Data Manipulation

```
'// open the file and grab the first four bytes
hFile = FreeFile
Open fileName For Binary Access Read Shared As hFile
Get #hFile, 1, bytes
```

We use the familiar Open statement to open the file. In this case, we are requesting Binary access because we want to grab the first four bytes of the file as binary data. We have declared a byte array of four characters as our data variable. When we call the Get statement, we pass the array, and specify a starting position at the first byte in the file. The Get statement will fill the array with the first four bytes from the file.

The process to identify the Byte Order Mark is a moderately complex Select Case block. We switch on the first byte. We know that all of our recognized Byte Order Marks begin with either a 00, EF, FE, or FF value (in hexadecimal), so those are the values we switch on. If the first byte is any other character, we know that this is an ANSI file (or potentially an unrecognized Byte Order Mark).

```
'// Check the first byte
Select Case bytes(0)
Case 0:      '// 00
    If bytes(1) = 0 And bytes(2) = 254 And bytes(3) = 255 Then
        '// 00 00 FE FF detected
        bom = ByteOrderMark.UTF32BE
    Else
        '// Not one of our signatures, return None
        bom = ByteOrderMark.None
End If
```

If the first byte is 00, we check to see if the second byte is also 00 (0 decimal), and the third byte is FE (254 decimal), and the fourth byte is FF (255 decimal). If so, we have recognized the UTF-32 Big Endian Byte Order Mark, and so we return that enum value. If not, this is random binary data and not a recognized BOM, so we return ByteOrderMark.None.

```
Case 239:    '// EF
    If bytes(1) = 187 And bytes(2) = 191 Then
        bom = ByteOrderMark.UTF8
    Else
        bom = ByteOrderMark.None
End If
```

If the first byte is EF (239 decimal), we check to see if the second byte is BB (187 decimal) and the third byte is BF (191 decimal). If so, we've located a UTF8 Byte Order Mark and we return that enum value. Otherwise, as in the previous case, we have random binary data and not a recognized BOM.

```
Case 254:    '// FE
    If bytes(1) = 255 Then
        '// FE FF detected - UTF16 (Big Endian)
        bom = ByteOrderMark.UTF16BE
    Else
        bom = ByteOrderMark.None
End If
```

If the first byte is FE (254 decimal), then we check the second byte for FF (255 decimal). If we have a match, we have found the UTF-16 Big Endian BOM.

```

Case 255:  '// FF
    If bytes(1) = 254 Then
        '// FF FE detected
        If bytes(2) = 0 And bytes(3) = 0 Then
            '// FF FE 00 00 - UTF32 (Little Endian)
            bom = ByteOrderMark.UTF32
        Else
            '// FF FE - UTF16 (Little Endian)
            bom = ByteOrderMark.UTF16
        End If
    Else
        bom = ByteOrderMark.None
    End If

```

If the first byte is FF (255 decimal), we have a slightly more complicated test. First, we check to see if the second byte is FE (254 decimal). If it is not, then we have random binary data, and we return ByteOrderMark.None. If the second byte is FE, then we need to check the third and fourth bytes to see if they contain 00 (0 decimal). If they do, we have the FF FE 00 00 sequence, which represents a UTF32 Little Endian Byte Order Mark. If they do not contain 00, then we have the FF FE sequence, which represents the UTF16 Byte Order Mark.

```

Case Else:
    '// Unrecognized pattern
    bom = ByteOrderMark.None
End Select

```

If the first byte is any other value, we have random binary data and no recognized Byte Order Mark.

```

Cleanup:
    '// Close the file
    On Error Resume Next
    Close hFile

Terminate:
    '// Set return value and exit
    On Error Goto 0
    GetByteOrderMark = bom
    Exit Function

Err_Handler:
    '// Return the error code
    bom = -(Err.Number)
    Select Case Err.Number
        Case 53: '// FileNotFoundException
        Case Else:
            '// Unexpected error, display message
            MsgBox Err.Description, vbCritical, Err.Number
    End Select

```

Part II: Data Manipulation

```
Err.Clear  
GoTo Cleanup  
  
End Function
```

Like all good procedures, we have a cleanup block, a terminate block, and an error handler (a very basic one in this case).

You can test this function against the SampleAnsi.txt, SampleUTF8.txt, SampleUTF16.txt, and SampleUTF32.txt files included as part of the sample code download for this chapter.

We created the DocumentIsType function as an example of a typical helper function. Whereas the GetByteOrderMark function tells you which type of ByteOrderMark is contained in the file, the DocumentIsType function enables you to quickly query whether or not a document is of a certain type. It calls GetByteOrderMark to determine the type of BOM for the document, and then compares that value to the value passed as an argument.

```
' Procedure : DocumentIsType  
' Purpose   : Returns a boolean value signaling whether the document  
'               contains the specified ByteOrderMark  
' Arguments : fileName - path to the document to check  
'               documentType - ByteOrderMark enum value to compare  
' Returns   : true if fileName contains the specified ByteOrderMark  
Public Function DocumentIsType( fileName As String, _  
    documentType As ByteOrderMark) As Boolean  
    DocumentIsType = (documentType = GetByteOrderMark(fileName))  
End Function
```

Splitting Strings

We can split a delimited string using the Split VBA function. Split accepts a string that has delimited data, and the delimiting character(s). It returns an array of items with the delimiters removed.

The SplitTest function shows how to use the Split function.

```
Public Sub SplitTest()  
    Dim vItem As Variant  
    Dim item As String  
    Dim rgItems() As String  
  
    item = "Mr,Michael,,TUCKER,Sr,PhD"  
    rgItems = Split(item, ",")  
  
    For Each vItem In rgItems  
        Debug.Print Trim$(vItem)  
    Next vItem  
End Sub
```

This function takes the item string with delimited name elements, and returns an array with each name element stored as its own string.

Mr
Michael
TUCKER
Sr
PhD

Notice that we get an empty array element where there was no data between the delimiters.

You can also split a string into individual words. This technique can be helpful in certain cases, such as when you are parsing a name or address and want to check each word against a dictionary of known values (for example, honorifics, titles, suffixes, street types, and directionals). To split on words, you can call the `Split` function using a space character as the delimiter. Because the space character is the default delimiter, you can omit the delimiter argument altogether, although we recommend always including the argument to improve the readability of your code. Both of the following statements return an array of five strings:

```
rgItems = Split("Mr. Michael Tucker Sr PhD", " ")
rgItems = Split("Mr. Michael Tucker Sr PhD")
```

The VBA String Handling Functions

The `Split` function works well when you have items that are consistently formatted. Often, you will have data with a known delimiter, but you don't have a known or consistent number of data elements. You merely want to extract data up to (or beginning with) the delimiting character. In this case, the `Split` function probably does not provide the functionality you need. You can use the `Left`, `Right`, `Mid`, and `InStr` VBA functions to remove the data you want. The `Left` function returns the left-most *n* characters of a string. The `Right` function returns the right-most *n* characters of a string. The `Mid` function returns a specified number of characters (by default, all characters in the string) beginning at a specific position. Finally, the `InStr` function returns the character position (counting from the first character in the string) of a specified substring. If the substring is not located, `InStr` returns 0. We're assuming you are familiar with these string-handling functions, and so the descriptions here are very brief. If you need more information, these functions are clearly documented in the help file.

Our string parsing toolbox defines the constant `INSTR_NOTFOUND`, which we use to test for an `InStr` call that does not find the substring. It also defines a constant `TOKEN_NOTFOUND`, which is one less than `INSTR_NOTFOUND`. We'll explain this oddity in a moment.

```
Public Const INSTR_NOTFOUND As Long = 0
Public Const TOKEN_NOTFOUND As Long = -1
```

Part II: Data Manipulation

We also define constants for each of the delimiters that we use frequently.

```
Public Const DELIM_SPACE As String = " "
Public Const DELIM_COMMA As String = ","
Public Const DELIM_TAB as string = vbTab
```

Note that you cannot use the `Chr$` function to assign a value to a constant because, by its very nature, the return value of a function is variable data. `vbTab` is an intrinsic constant, a member of the `VBA.Constants` class built into the VBA library, and so it can be assigned to a constant string. Several useful constants are defined in that class, and we could also just as easily use the intrinsic constant directly (and in the process save the memory space). We like the consistent naming convention, and the single-character constant is small enough that we're not too concerned about allocating extra memory to enable the use of our naming convention.

Let's look at two of the workhorse functions in our parsing toolbox, `PeekToken` and `PeelToken`.

PeekToken

`PeekToken` is used to find a token at the beginning of a string. For example, if we have the following string containing products and the quantity ordered, separated by commas,

```
Chai,3,Alice Mutton,2,Doily,14,NuNuCa Nufß-Nougat-Crème,3
```

we can use the `PeekToken` function to return the first token from the string (in this case the product name Chai). We have written `PeekToken` to accept the string that you want to locate the token in, and an optional argument to specify which delimiter we use for the string. We have defined the default delimiter to be the `DELIM_COMMA` constant; this can be easily modified if your data generally uses a different delimiting character.

```
' Procedure : PeekToken
' Purpose   :
' Arguments  :
'
'
Public Function PeekToken(ByVal item As String, _
    Optional delimiter As String = DELIM_COMMA) As String
    Dim pch As Long
    pch = InStr(Nz(item), delimiter) - 1

    If pch = TOKEN_NOTFOUND Then pch = Len(Nz(item))

    PeekToken = Trim$(Left$(Nz(item), pch))
End Function
```

Inside the function, we use the `InStr` function to locate the first instance of the delimiter in the item string:

```
pch = InStr(Nz(item), delimiter) - 1
```

In this example, the first comma occurs at position 5 in the string, so `InStr` will return 5. Notice that we are decrementing the value returned to `pch` by one. We do this because the call to `InStr` returns the position of

the first character of the delimiting character. Because we do not want to see the delimiter, we back up one character. This is why the TOKEN_NOTFOUND value is one less than the value of INSTR_NOTFOUND: We use TOKEN_NOTFOUND whenever we're testing against an adjusted character position such as this. So in our example, pch contains 4, which is the length of our first token, Chai.

The token is returned by this line in the code:

```
PeekToken = Trim$(Left$(Nz(item), pch))
```

We use the `Left$` function to return the token. The `pch` variable points to the character immediately preceding the first delimiter, so the `Left$` function returns everything from the first character to this point. In our example, we are asking for the left-most four characters of the item string, which returns `Chai`. We run the results through the `Trim$` function to remove any leading or trailing whitespace; it is not necessary in this case, but if your data isn't clean, this step can significantly reduce your work when working with the returned tokens. If you are certain your data is clean, you can remove the call to `Trim$` to improve performance.

For robustness, whenever we reference the `item` variable, we wrap it in the `Nz` VBA function. If the value of the `item` variable is Null, `InStr` will return Null instead of 0, and many of the other string handling functions throw an error if you attempt to pass Null into them. If the value of the argument to the `Nz` function is Null, `Nz` returns an alternate value (by default an empty string). This guarantees that `InStr` will always see a valid string (even if it is empty, it's still a string!) and so the return value from `InStr` will always be a numeric value. Like the call to `Trim$`, if you are certain your data is clean and contains no Null values (or that your code will never pass a Null value), you can remove the `Nz` function calls for a slight performance improvement.

PeelToken

While `PeekToken` works well for its purpose (identifying the first token in a string), it doesn't allow you to find subsequent tokens in a string. `PeelToken` performs the same task as `PeekToken`, except that when it returns the token to you, it removes the token (peels it off) from the input string. The next time you call `PeelToken` with the same input string, you'll get the next token from the string.

Using our sample string, if you call `PeekToken` five times, you'll receive the same string, `Chai`, each time because `PeekToken` does not modify the input string. If you use `PeelToken` instead, you'll receive the following output the first five times you call it:

```
Chai  
3  
Alice Mutton  
14  
Doily
```

Let's look at how this works:

```
Public Function PeelToken(ByRef item As String, _  
    Optional delimiter As String = DELIM_COMMA) As String
```

Part II: Data Manipulation

PeelToken adds the `ByRef` keyword to the `item` argument. `ByRef` is the default, and it is not required to add it here, but when we are manipulating the contents of a string inside a function, we like to specifically call this out by adding the `ByRef` keyword to make the code more self-documenting.

```
Dim pch As Long
pch = InStr(Nz(item), delimiter) - 1

If pch = TOKEN_NOTFOUND Then pch = Len(Nz(item))

PeelToken = Trim$(Left$(Nz(item), pch))
```

The body of the function is identical to the `PeekToken` function.

```
item = Mid$(Nz(item), pch + Len(delimiter) + 1)
End Function
```

The last line of this function is where we remove the first token from the input string. We use the `Mid$` function to extract everything after the first delimiter. Recall that `pch` points to the end of the first token. In our example, on the first call, it contains 4, pointing to the last character (`i`) in the first token (`Chai`). We add the length of the delimiter to `pch` (in this example, our delimiter is a comma, one character in length) so `pch` now contains 5, which points to the comma itself. Finally we add 1 to `pch` (so now `pch` contains 6), to point to the character immediately following the delimiter. In this example, the `Mid$` call essentially looks like this:

```
Mid$(Nz(item), 6)
```

Because we do not specify the optional third argument to `Mid$`, we are instructing VBA to select all characters from `item` starting with the sixth character.

We can create a simple procedure to test these two functions: The function contains a variable named `item` that shows three products from the Northwind Products table, and a quantity of the product that was ordered. We will add a space before and after one of the items to demonstrate that these functions return trimmed strings.

```
Public Sub PeelTokenTest()
    Dim item As String
    item = "Chai,3, Alice Mutton ,14,Doily,14"

    Debug.Print "Item before PeelToken", _
        "PeekToken", "PeelToken", "Item after PeelToken"
    Do While Len(item) > 0
        Debug.Print item;
        Debug.Print , PeekToken(item), PeelToken(item), _
            IIf(vbNullString = Nz(item), "{empty}", item)
    Loop
End Sub
```

The following table displays the output that will appear in the Immediate window after running this procedure.

Item before PeelToken	PeekToken	PeelToken	Item after PeelToken
Chai,3, Alice Mutton ,14,Doily,14	Chai	Chai	3, Alice Mutton ,14,Doily,14
3, Alice Mutton ,14,Doily,14	3	3	Alice Mutton ,14,Doily,14
Alice Mutton ,14,Doily,14	Alice Mutton	Alice Mutton	14,Doily,14
14,Doily,14	14	14	Doily,14
Doily,14	Doily	Doily	14
14	14	14	{empty}

Notice that we can call `PeekToken` without affecting the value of `item`, but when we call `PeelToken`, the value of `item` is changed, and the token is removed from the string. Also notice that both `PeekToken` and `PeelToken` remove leading and trailing spaces from their output (the “Alice Mutton” values are trimmed).

Replacing Tokens in Strings

The `Replace` VBA function provides everything you need to implement simple token replacement. Once you have determined your token value, you call the `Replace` function against a string, passing in the data string, the token you wish to replace, and the value you wish to substitute.

```
Replace(item, token, substitution)
```

It is important to remember to assign the results of the `Replace` function to the string you want to hold the detokenized string. We’ve wasted lots of time over the years trying to figure out why tokens weren’t being replaced only to realize we had forgotten to assign the results to the string. It is a common pattern to assign the results back to the same string that had previously contained the tokens.

```
item = Replace(item, token, substitution)
```

When you use the pattern, it is important to remember that the content of the `item` variable is actually being changed. If you use this pattern inside of a function that is passed `item` in an argument, it is important to remember that by default, all strings are passed `ByRef` in the VBA environment. This means that unless you explicitly declare the `item` argument `ByVal` in your function, when you change the value of `item` in your function, the value of `item` will also be changed in the calling code.

What Is the Difference Between Mid and Mid\$?

Most of the VBA functions come in both variant and string versions. The variant version returns a variant, which can be passed into any variable or argument that accepts a variant. This gives you an abundance of flexibility, at the cost of the overhead of working with variant data.

When you know that you are working with string data, the string version of the function provides a potentially significant performance boost. The string version returns a string object instead of a variant.

To call the string version of a VBA function, append a dollar sign character (\$) following the function name.

Variant Version	String Version
Left	Left\$
Right	Right\$
Mid	Mid\$
Trim	Trim\$
LTrim	LTrim\$
RTrim	RTrim\$

Formatted Token Replacement

We have created a `StringFormat` function to perform token replacement when working with templated data that we have control over. This function mimics much of the behavior of the `String.Format` method in the Microsoft.NET Framework. It accepts a string with tokens in the form `{0}` and a list of arguments to replace the tokens. If there are three arguments in the list, it will replace token `{0}` with the first argument, token `{1}` with the second argument, and token `{2}` with the third argument.

The `StringFormat` function uses the `ParamArray` keyword to pass a variable number of arguments in a Variant array. An argument may be declared a `ParamArray` argument only if it is the last argument in an argument list. You can work with the arguments in your code just as you would with any array.

```
Public Function StringFormat(ByVal item As String, _
    ParamArray args() As Variant) As String
    Dim i As Long
```

We determine the upper and lower bounds of the array (these will be set for you automatically and will accurately represent the contents of the array). We loop through each argument in the `ParamArray` (here named `args`), and pass the argument value to the `Replace` function. The result from the

Replace function is stored back into the item variable before we repeat the loop. Once we have replaced each argument in the ParamArray, we return the value of the item variable.

```

For i = LBound(args) To UBound(args)
    item = Replace(item, "{" & i & "}", args(i))
Next i
StringFormat = item
End Function

```

Notice that we have explicitly declared the item argument `ByVal`, to ensure that changes we make to the string in this function don't have unintended consequences in the calling code.

We use the `StringFormat` function all the time when we're generating templated data. We also use this function a lot when generating strings for message boxes, status bars, and labels on forms. We will use `StringFormat` later in this chapter to generate HTML files, and Chapter 15 uses `StringFormat` with a template to generate Help files and user documentation.

Parsing into Data Types

When you parse text out of a file, you are generally working with string data. Often, you will want to convert the text you parse into a more specific data type. For example, a text file may contain order information that includes a customer ID number (integer), an order date (date), and a shipping charge (currency). You will often need to convert these strings back into the specific data types. VBA provides several functions that allow you to perform these conversions.

VBA Type Conversion Function	Purpose
<code>CBool(expression)</code>	Converts a string or number to a Boolean. Numbers: 0 returns False, everything else returns True.
<code>CCur(expression)</code>	Converts a number or string to a currency value. String may include the currency symbol defined in the regional settings.
<code>CDate(expression)</code>	Converts a number or string to a datetime value. String may include Month names as defined in the regional settings.
<code>CDbl(expression)</code>	Converts a number or string to a double.
<code>CByte(expression)</code>	Converts a number or string to a byte value. Number must be in the range 0–255.

Continued on next page

Part II: Data Manipulation

VBA Type Conversion Function	Purpose
<code>CDec(expression)</code>	Converts a number or string to a decimal.
<code>CInt(expression)</code>	Converts a number or string to an integer. Must be in the range -32768 to +32767.
<code>CLng(expression)</code>	Converts a number or string to a long integer. Must be in the range -2,147,483,648 to 2,147,483,647.
<code>CSng(expression)</code>	Converts a number or string to a single.
<code>CStr(expression)</code>	Converts any value to a string.
<code>CVar(expression)</code>	Converts any value to a variant.

You can pass any string or numeric expression into one of the type conversion functions, and the function will pass back the expression coerced into the appropriate data type. For example, if you pass the string 1.23 into the CCur function, you will receive back a Currency data type object with the value 1.23.

The type conversion functions respect the users' regional settings as defined by the operating system. This means that decimal points, thousands separator characters, and currency symbols are recognized as they are defined on the system. By default, in the United States, the \$ symbol is defined as the currency symbol, a dot (.) is the decimal, and a comma character (,) is the thousands separator.

The following table displays several Input/Output pairs that result when the operating system regional settings are defined with the US default values.

Expression	Resulting Data type	Resulting Value
<code>CCur(1.23)</code>	Currency	1.23
<code>CCur(\$"1.23")</code>	Compile Error Illegal Character (Cannot have \$ symbol outside a string)	
<code>CCur("\$1.23")</code>	Currency	1.23
<code>CCur("¤.23")</code>	Error 13 Type Mismatch (¤ is not the defined currency symbol)	
<code>CCur("1,23")</code>	Currency	123 (comma is not a valid decimal separator)
<code>CDate(#4/26/2008#)</code>	DateTime	April 26 2008
<code>CDate(#4/5/2008#)</code>	DateTime	April 5 2008

Expression	Resulting Data type	Resulting Value
CDate(#26/4/2008#)	DateTime	April 26 2008
CDate("4/26/2008")	DateTime	April 26 2008
CDate(39564.4798611111)	DateTime	April 26 2008 11:31:00 AM

A few of the results for the CDate conversions may be surprising. In the United States, Windows defaults to m/d/yy for the short date format, and so the conversion of 4/26/2008 to April 26 2008 is expected. It may be surprising at first that VBA would correctly interpret #26/4/2008# as April 26 even when the short date format is set to m/d/yy. It makes sense because 26/4/2008 is unambiguous: There is no 26th month, and so the only logical interpretation is April 26. On the other hand, the expression #4/5/2008# is ambiguous, depending upon how you set the regional settings: It could be April 5 or it could be May 4. Whenever there is ambiguity, VBA will use the values set in the regional settings.

The conversion of 39564.479861111 to 11:31 a.m. on April 26, 2008 might seem odd at first, too, but there is a logical (and extremely useful) explanation. DateTime values in Access are internally stored as a double numeric value. The integer portion of the double (the digits to the left of the decimal) indicate the number of days from December 30, 1899. The mantissa portion (the digits to the right of the decimal) indicate the time of day as a relative percentage. Let's look at an example.

There are 39,564 days between December 30, 1899 and April 26, 2008. For a quick and dirty approximation, divide 39,564 by 365.24 (the average number of days in a year, including leap year) and you get approximately 108.32. 2008 is 108 years after December 30, 1899, and April is approximately one-third of the way through the year, so this number is clearly in the ballpark. If you must, whip out a calendar and check off the days — you'll soon trust the value!

If we take the mantissa of the value (.479861111) and multiply by 24, we get 11.516666664. If we take the mantissa of that number and multiply by 60 (the number of minutes in an hour), we get 30.99999984, which when rounded to timekeeping precision, results in 31 minutes. So we see that the date representation is correct.

Access allows Boolean fields to contain three sets of values: True/False, Yes/No, and On/Off. Unfortunately, the CBool function recognizes only the "True" and "False" strings; if you execute the following code, you will receive a Type Mismatch runtime error:

```
CBool("on")  // Error 13 Type Mismatch
```

We have created a ConvertBoolean function in our toolkit to handle all three of the Boolean string sets:

```
Public Function ConvertBoolean(item As Variant) As Boolean

    ' Trim leading/trailing space
    item = Trim(item)
    ' convert yes/no and on/off strings
    Select Case item
        Case "Yes", "On"
            item = True
        Case "No", "Off"
            item = False
    End Select
End Function
```

Part II: Data Manipulation

```
Case "No", "Off"
    item = False
End Select

// If we are unable to convert a string, we return false
On Error Resume Next
ConvertBoolean = CBool(item)
On Error GoTo 0

End Function
```

This function tests the value passed for one of the alternate string sets and explicitly converts the variant to True or False. For example, if we pass the string “on” to the ConvertBoolean function, the Select Case statement will change the value of item to True. This value is then passed to the CBool function, and the result is returned. If we pass a string that cannot be evaluated by the CBool function, a Type Mismatch error will be generated. Because we add a Resume Next directive before we attempt to convert the value, the Type Mismatch error will be ignored, and the ConvertBoolean function will return False.

An End-to-End Example

Let's look at one scenario and see how we can use VBA to parse the contents of a moderately complex text file into a table.

In this scenario, every month, our customer is presented with a text file that contains an extract of order data from the previous month. Unfortunately, in a misguided attempt to be more efficient, the extract is created with ASCII text formatting, which makes it impossible to simply import the file! The file looks like this:

Order ID	Customer	Order Date	Shipped Date	Freight	Ontime
10400	Eastern Connection	01-Jan-1997	16-Jan-1997	\$83.93	Yes
10423	Gourmet Lanchonetes	23-Jan-1997	24-Feb-1997	\$24.50	No
10424	Mère Paillarde	23-Jan-1997	27-Jan-1997	\$370.61	Yes
10426	Galería del gastrónomo	27-Jan-1997	06-Feb-1997	\$18.69	Yes
10427	Piccolo und mehr	27-Jan-1997	03-Mar-1997	\$31.29	No

We need to create a function that will import this file into a table in our database. Just to make this example more realistic, we'll say that although the data types of the fields are the same between the extract and the table, the names and order of the fields are different!

The following table shows how the fields in the text file map to the fields in the data table.

Field in Text File	Field in Table	Data type
Order	OrderID	Long Integer
Customer	Customer Name	Text
Order Date	DateOrdered	DateTime

Field in Text File	Field in Table	Data type
Shipped Date	DateShipped	DateTime
Freight	Freight Charges	Currency
Ontime	Was the order shipped on time?	Boolean (and of course you will never name a field like this!)

First, paste the following Data Definition Language (DDL) query into a new query window to create a table to store your records. (DDL queries are described in detail in Chapter 6.)

```
CREATE TABLE DatatypeParseExampleDestinationTable
(
    CustomerName Text(40),
    OrderID Long PRIMARY KEY,
    DateOrdered DateTime,
    DateShipped DateTime,
    [Was the order shipped on time?] bit,
    [Freight Charges] Currency
)
```

After you have successfully created the table, create the following function:

```
' Procedure : ParseOrdersExtractFile
' Purpose   : Import formatted text file into a recordset
' Arguments  : fileName - name of the text file to import
' Remarks   : This procedure has specific knowledge of the layout of the file it is
'              : going to parse, specifically:
'              : 1) There are three header rows that can be skipped
'              : 2) Each data row is delimited by a pipe symbol '|'
'              : 3) Because of the pipe sign in the first position, the first item
'              :     parsed is always empty and can be skipped
'              : 4) The field names and datatypes
'

Public Sub ParseOrdersExtractFile(fileName As String)
    Const Delimiter As String = "|"
    Const DividerCharacter As String = "-"
    Const DividerLength As Long = 128
```

First, we set up some basic constants that define characteristics of the text file. We identify the field delimiter as the pipe symbol, and identify the row dividers as a string of hyphens 128 characters in length.

Next, we set up variables to support the VBA file-handling functions as well as the recordset we will use to add rows to the table. We also define a string array to hold the elements we will split out of each line of data.

```
Dim hFile As Long
Dim lineItem As String
Dim db As DAO.Database
Dim rs As DAO.Recordset
Dim rgItems() As String
```

Part II: Data Manipulation

We create the recordset we will use to add rows to the table and then we get the next available file handle and open the text file for input. Because we know the first three lines are header data that we don't need to add to our table, we get those three lines and immediately discard them.

```
On Error GoTo Err_Handler
Set db = CurrentDb
Set rs = db.OpenRecordset("DatatypeParseExampleDestinationTable")
hFile = FreeFile
Open fileName For Input As hFile
    // Skip the first three rows (header data)
Line Input #hFile, lineItem
Line Input #hFile, lineItem
Line Input #hFile, lineItem
```

At this point, we should be sitting on the first line of data in our text file.

We will loop through each remaining line in our text file until we hit the end of the file. We check each line to see if it is a divider line. If the line is not a divider line and does contain data, we split the line into the rgItems array. We should now have each field in a separate element of the array.

```
'// Loop through the remainder of the data file
Do While Not EOF(hFile)
    Line Input #hFile, lineItem

    // if this is a divider line, skip it
    If String(DividerLength, DividerCharacter) <> lineItem Then
        rgItems = Split(lineItem, Delimiter)
```

We prepare a new row for the recordset, and then start loading the fields. We need to skip the first element in the rgItems array because each row begins with the delimiter character, and the split function creates an empty element in the first position to mark the "missing" data that would have been before this delimiter. Because the file is consistently formatted, it is safe to ignore this first array element (rgItems(0)).

```
'// prepare recordset for new row
rs.AddNew
With rs
    // skip the first item, since it's empty
```

The second array element contains the OrderID. This needs to be stored as a Long Integer data type, so we pass it to the CLng conversion function, and store the result in the OrderID field in the recordset.

```
'// Parse OrderID into a Long
rs("OrderID") = CLng(rgItems(1))
```

The third array element contains the Customer's name. This is a string field, so we don't need to perform any conversion on the data, but we should trim the string so that there is no extraneous leading or trailing whitespace.

```
'// Parse Customer into a string
// (no conversion, but trim the contents)
rs("CustomerName") = Trim$(rgItems(2))
```

Repeat the process for the `OrderDate`, `ShippedDate`, and `Freight` fields. Purely for demonstration purposes, we have trimmed the value of the `OrderDate` field before passing it to the `CDate` function, and we have not trimmed the value of the `ShippedDate` field. This demonstrates that the `CDate` function is very flexible about handling whitespace.

```
'// Parse OrderDate into a date field
rs("DateOrdered") = CDate(Trim$(rgItems(3)))

'// Parse ShippedDate into a date field (don't bother to trim
'// it first; notice that conversion functions handle whitespace
'// gracefully)
rs("DateShipped") = CDate(rgItems(4))

'// Parse Freight into a currency field
rs("Freight Charges") = CCur(rgItems(5))
```

Because the `Ontime` field contains Yes and No string values, we cannot use the `CBool` function directly (because `CBool` does not recognize Yes and No). We pass the string into the `ConvertBoolean` helper function that we created earlier (which does recognize the strings “Yes” and “No”), and store that result in the recordset.

```
'// Parse Ontime into a boolean field
rs("Was the order shipped on time?") = ConvertBoolean(rgItems(6))
End With
```

Finally, we call `Update` on the recordset to commit the row, and loop back to the next row in the text file. Our standard `Cleanup` and `Terminate` blocks take care of closing the file and recordset, and releasing the database objects.

```
'// Commit the record
rs.Update
End If
Loop
Cleanup:
On Error Resume Next
Close #hFile
rs.Close
Set rs = Nothing
Set db = Nothing

Terminate:
On Error GoTo 0
Exit Sub
```

Finally, we have a simple error handler. Because `OrderID` is a primary key, if we attempt to load an order into the table twice, we will receive a 3022 runtime error, reporting an attempt to add a duplicate record. We catch this, and offer the user a chance to break out of the loop and stop processing the file.

```
Err_Handler:
Select Case Err.Number
Case 3022: '// Duplicate record in table
            '// cancel the addnew
```

Part II: Data Manipulation

```
    rs.CancelUpdate
    If MsgBox("An entry already exists for this record.", _
              vbOKCancel, Err.Number & ": " & Err.Description) = vbCancel Then
        GoTo Cleanup
    End If
    Resume Next
Case Else
    MsgBox Err.Description, vbCritical, Err.Number
    Resume: GoTo Cleanup
End Select

Resume Next
End Sub
```

Now that we have the function created, run the function passing the name of the `OrderExtract` table. Step through the code and watch as we open the file, read each row in the text file, parse the row, and fill our recordset.

Splitting Names

Splitting names isn't too difficult if data is consistently formatted and uses delimiters. If all of your names are formatted like this, we have a very easy task:

- TUCKER, Michael
- COOPER, Robert
- CARTER, Christopher

We can split on the delimiting character (the comma), assigning everything to the left of the comma to the `LastName` field, and everything to the right of the space to the `FirstName` field.

Without delimiters, it's still fairly straightforward if the data is clean and consistently formatted. However, if the data isn't clean and isn't consistently formatted, there are distinct limits on how much data you can split.

Let's look at some names and consider the challenges we face when attempting to parse names:

- Michael Tucker
- Rob Cooper
- Carter Christopher

These names don't look too bad. We can split on the words, placing the first word in the `FirstName` field, and the second word in the `LastName` field. For the third name, we'll have to hope that the user entered the data consistently and didn't reverse the first and last names.

Things get a little more interesting when names have more than two words in them.

- Scott Matthew Hoisington
- Mary Jo Jones

These names looks pretty easy, too. Especially if we have a `MiddleName` or `MiddleInitial` field, we can again break on the word boundaries. Scott goes in the `FirstName` field, Matthew goes into the `MiddleName` field, and Hoisington goes in `LastName`. Same thing for Mary Jo Jones. Or is it? Mary and Jo together are often times considered a single first name. Do we need Mary Jo in the `FirstName` field, or is it okay to put Mary in the `FirstName` field and Jo in the `MiddleName` field? What if we know that her full name is Mary Jo Roberta Jones? Our brain probably concludes that Mary Jo is the first name, and Roberta is the middle name. That is, until we look at the data further, and see that her father's name is William Roberta and her mother's name is Sara Jones. Now, we may conclude that Mary Jo is the first name, and Roberta Jones is a hyphenated last name sans hyphen.

Many last names have prefixes. Sometimes these prefixes are an integral part of the name, but other times, they stand alone as separate words.

- Stephen MacDonald
- Lucile di Borgia
- Mark van der Bogert

When we look at the last two names, our brains recognize first names and multi-word last names. Experience has shown us that "van der Bogert" is probably a composite last name. If Mark has a middle name, it's probably neither "van" nor is it "der." Unfortunately, short of a lookup into a master list of names, you are going to have a hard time writing code that can recognize this name correctly.

There are several additional categories of name data that can pose significant challenges to parsing code.

- Cher
- Dr. Robert Philip Coggins Sr PhD

Oh yeah! How do you handle people with one name? Is it a last name? Is it a first name? How do you handle a database that requires data in both the `FirstName` and `LastName` fields? How about honorifics, generational disambiguators, and titles and degrees?

Throughout this chapter and this book, you see over and over again the importance of two cardinal rules of database design:

- Consistent data entry is essential in order to get meaningful results out.
- It is easy to combine data; it is always more work (and often impossible) to accurately split data apart.

Consistency is crucial. Even if the data is in a sub-optimal format, so long as it is *consistently* sub-optimal, you can write code to let you work with it. But if you have entered data in different ways, with inconsistent delimiters, it is almost impossible to generate consistently accurate results.

Granularity is your friend. It is almost trivial to combine two pieces of data to form a greater whole. For instance, if you have a first name and a last named stored in a database, you can use the expression `[First Name] & " " & [Last Name]` to create a `[Full Name]` expression. However, as you have seen, if you have the data stored in a `[Full Name]` field and you wish to extract the first name out of the expression, you have some nasty work ahead of you, and you won't be able to consistently generate 100 percent accurate results.

Part II: Data Manipulation

Depending upon the needs of your application, we recommend storing name data in at least separate first and last name fields. It is not uncommon to see name data split into four or more fields:

- LastName (Family Name, or surname)
- FirstName
- MiddleName (or MiddleInitial)
- NameSuffix (Jr., Sr., III, and so on)
- Honorific (Mr., Mrs., Dr., Honorable, and so on)
- Title/Degree (MD, PhD, JD, and so on)

Suffixes, Honorifics, and Titles/Degrees are good candidates for lookup tables to assist with consistent data entry (and potentially, to reduce the size of your data tables). If you choose to use lookup tables, you will most likely want to enable users to add items to these lookup tables on-the-fly.

This advice applies equally to addresses and phone numbers.

Address Element Granularity

If you have a single address field in your database, mailing labels will be easy, until you try to sort your labels in an address-presort order. We recommend, at a minimum, having separate fields for the street address, city, region, country, and postal code. Depending upon your application, you may wish to further granulate the street address field. In applications where duplicate addresses must be identified with a high degree of certainty, it's not unusual to split addresses into eight or more fields, in addition to the city, region, country, and postal code fields:

- AddressNumber
- AddressNumberSuffix
- DirectionalPrefix
- StreetName
- StreetType
- DirectionalSuffix
- UnitType
- UnitNumber

This layout allows you to separate most American and Canadian street addresses into each of their constituent parts. Consider how the following example addresses fit into this schema.

1452-1/2 E 18th Ct Apt B-101, Great Falls MT 59401 USA

215 Glacier Ave NW Suite 5, Toronto ON M5R 2H7 CANADA

1 Microsoft Way, Redmond WA 98052-6399 USA

22bis rue Vieille du Temple, 75004 Paris FRANCE

Address Field	Address 1	Address 2	Address 3	Address 4
AddressNumber	1452	215	1	22
AddressNumberSuffix	½			bis
DirectionalPrefix	E			
StreetName	18 th	Glacier	Microsoft	Vieille du Temple
StreetType	Ct	Ave	Way	rue
DirectionalSuffix		NW		
UnitType	Apt	Ste		
UnitNumber	B-101	5		
City	Great Falls	Toronto	Redmond	Paris
Region	MT	ON	WA	
Country	US	CA	US	FR
PostalCode	59401	M5R 2H7	98052-6399	75004

Addresses from most foreign countries generally fit into this structure; however, the order of the fields may change dramatically from country to country, and you will need to account for this if your database has significant geographical diversity.

Remember that postal codes should be stored in a text field. In the United States and many other countries, postal codes may contain leading zeros, which will disappear if stored in a numeric field. Many countries including Canada and Great Britain have letters in their ZIP codes (and US ZIP+4 postal codes contain the non-numeric hyphen character).

Directional fields, Street Type, UnitType, Region, and Country fields are good candidate for lookup tables, as are the City and/or Postal Code fields if you have access to a comprehensive list of postal codes (such as the postal code database available for purchase from most national postal authorities). Most national postal authorities have published standards for the size of address elements, as well as a list of authorized abbreviations. Internet RFC document 1366 defines standardized 2- and 3-character country codes for every country on the planet. Consider these standards when creating your addressing schema.

Run an Internet search for RFC1366 to locate a list of the standardized country codes.

Visit the United States Postal Service Web site at www.usps.com and search for “Address Automation Standards” for documentation on postal automation standards in the United States.

Phone Number Granularity

It is not uncommon to see phone numbers treated as a single field in a database, and in practice, this often works well, as long as you have consistent data entry. Even phone numbers can benefit from increased granularity. During the mid-1990s, America experienced a severe shortage of phone numbers, resulting in many new area codes. Literally millions of phone numbers needed to be updated with new area codes. Databases that had phone numbers split into multiple fields were trivial to update; those that stored phone numbers in a single field were a nightmare to correct.

When a table needs to hold phone number data, it should contain the following fields:

- CountryCode
- AreaCode
- Phone Number
- Extension

If your database consists entirely of phone numbers within the North American Numbering Plan (essentially any phone number in the United States, Canada, Mexico, and the Caribbean), you may consider omitting the CountryCode field, and splitting the phone number field into a central office number (3 digits) and a line number (4 digits). This is particularly useful when updating phone numbers for changed area codes because area code changes occur for entire central offices at a time. However, if your database contains phone numbers for other geographic areas that do not conform to the XXX-YYY-ZZZZ number pattern in North America, this may be less useful.

Phone number fields should be defined as text fields. In some countries, phone numbers contain required leading zeros, which will disappear if you store the number in a numeric field.

Creating HTML

Hypertext Markup Language, or HTML, is the *lingua franca* of the World Wide Web. Almost all Web pages are created using HTML. When working with a Web site, you often need to generate Web pages based on data in a database.

In this section, we discuss a couple of different techniques you can use to generate HTML documents from data contained in an Access database.

We assume you have some experience working with HTML files in this section. The syntax for creating HTML documents and CSS style sheet specifications is beyond the scope of this book, but there are many excellent reference books and abundant documentation on the Web.

Exporting HTML Files

You can export data or forms and reports using the HTML export feature in Access. This is a satisfactory solution if you need to create quick and dirty HTML pages, but you will be quite limited in the control you'll have over formatting and layout.

To export a table, query, form, or report, select the object you want to export, and choose the More button in the Export chunk of the External Data Ribbon. From the drop-down, choose the HTML Document item, as shown in Figure 5-1.

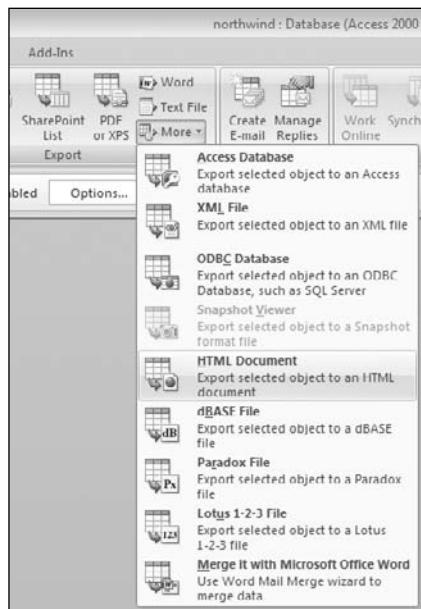


Figure 5-1

A dialog box is displayed that allows you to specify the filename of your new HTML document, and some options, including whether or not to include formatting in your file. The HTML Output Options dialog box is shown in Figure 5-2.

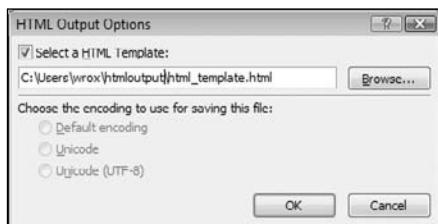


Figure 5-2

If you choose not to include formatting, Access will create a basic HTML page, and include a simple HTML table with data from your table or query, or a simplified representation of your form or report. If you are exporting a report, one or more pages are created — one for each page in your report — with navigation links to the first, last, previous, and next pages. Figure 5-3 shows a no-frills HTML page created by Access.

Part II: Data Manipulation

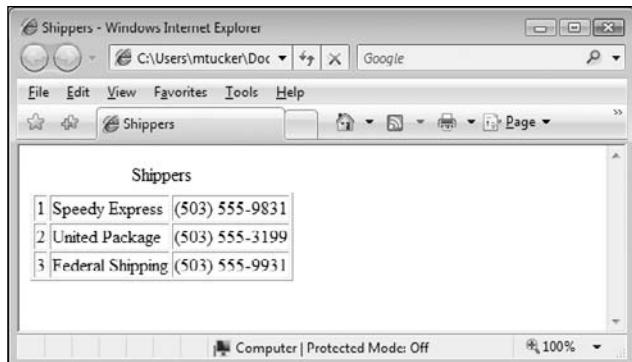


Figure 5-3

If you choose to include formatting when exporting a table or query, you will be presented with the option to use an HTML template file. This file allows you a slightly greater control over the presentation of your data. You can specify styles in the template file that will override the default formatting tag attributes that Access uses to generate formatting. This means you can control the font, color, spacing, and similar attributes of your output.

An HTML template file is a standard HTML file, but with placeholder tokens in the form of an HTML comment with the text “AccessTemplate” followed by a token identifier. The tokens listed in the table that follows are recognized by the Access HTML exporter.

HTML Template Token	Purpose
<!--AccessTemplate_Title-->	Inserts the name of the object being exported
<!--AccessTemplate_Body-->	Inserts a data table for tables and queries; or inserts formatted form or report data
<!--AccessTemplate_Page-->	Inserts the current page number (forms, reports only)
<!--AccessTemplate_PreviousPage-->	Creates a link to the HTML document representing the previous page in the report (reports only)
<!--AccessTemplate_NextPage-->	Creates a link to the HTML document representing the next page in the report (reports only)
<!--AccessTemplate_FirstPage-->	Creates a link to the HTML document representing the first page of the report (reports only)
<!--AccessTemplate_LastPage-->	Creates a link to the HTML document representing the last page of the report (reports only)

Within the template file, you can place any additional objects you want, including graphics, supplemental text, tables, and so on. You can add a CSS style sheet block to override the default formatting that Access

provides. You can even place the AccessTemplate tokens inside frames or tables. One restriction on the use of these HTML template tokens: Access will replace only the first occurrence of a given token. If you use the <!--AccessTemplate_Title--> token inside the <TITLE> tag, you cannot reuse the token in the body (you can put it there without error, but Access will not replace the token with the document title).

The following HTML file is a simple HTML template that demonstrates the use of the AccessTemplate tokens:

```
<html>
<head>
    <title><!--AccessTemplate_Title--></title>
</head>
<body>
    <h1>Web Content Generated from an HTML Template File</h1>
    <!--AccessTemplate_Body-->
    <br />
    <a href="<!--AccessTemplate_FirstPage-->">First</a>
    <a href="<!--AccessTemplate_PreviousPage-->">Previous</a>
    <a href="<!--AccessTemplate_NextPage-->">Next</a>
    <a href="<!--AccessTemplate_LastPage-->">Last</a>
</body>
</html>
```

Save this HTML as a text file, and then play around with HTML export. Try exporting a table as HTML specifying the HTML template. Notice how the title bar of the page shows the name of your table, and a table of data appears following the heading. View the source of the page, and you will see how Access explicitly generates a set of formats for your data (it chooses a Calibri font, and specifies colors for the table header cells, table borders, and so on). An HTML page created by exporting the Shippers table using our HTML template is shown in Figure 5-4.

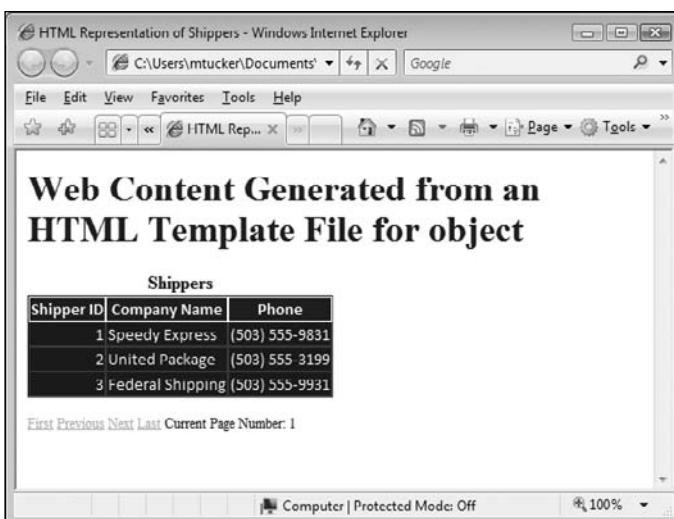


Figure 5-2

Part II: Data Manipulation

If we add a CSS Style block inside the head tag in our template file, we can override many of the default values that Access specifies.

```
<html>
<head>
    <title><!--AccessTemplate_Title--></title>
    <style>
        body
        {
            background-color: black;
            color: papayawhip;
            font-family: 'Times New Roman', sans-serif;
        }
        th
        {
            background-color: papayawhip;
            color: blue;
            font-family: 'comic sans', sans-serif;
        }
        td
        {
            background-color: yellow;
            color: red;
            font-family: 'courier new', monospace;
        }
    </style>
</head>
<body>
    <h1>Web Content Generated from an HTML Template File</h1>
    <!--AccessTemplate_Body-->
    <br />
    <a href="<!--AccessTemplate_FirstPage-->">First</a>
    <a href="<!--AccessTemplate_PreviousPage-->">Previous</a>
    <a href="<!--AccessTemplate_NextPage-->">Next</a>
    <a href="<!--AccessTemplate_LastPage-->">Last</a>
</body>
</html>
```

The body block in the style tag sets values that will be used in the body of the document. Here we set the document background color to black and use my favorite CSS color specification, papayawhip, for our text. The th block sets our style for the table header cells, and finally, the td block specifies formatting for the individual (non-header) table data cells.

Rerun your HTML export specifying this new template file with the CSS style block, and see the difference. The page now appears radically different, but there is a problem! The fonts and colors we asked for on the table elements aren't being respected: We still see the default font and text colors that Access applied to the earlier example. An HTML page created by exporting the Shippers table using our HTML template with CSS is shown in Figure 5-5.

Open the source and you'll see that Access still specifies its default colors and font information in a FONT tag. Most Web browsers (including Microsoft Internet Explorer and Firefox) allow FONT tags to override the formatting specified by CSS. Because Access introduces FONT tags everywhere, we are unable to control the font face, size, and color used for the data in the table grid. Obviously, this is a major limitation of the HTML export feature.

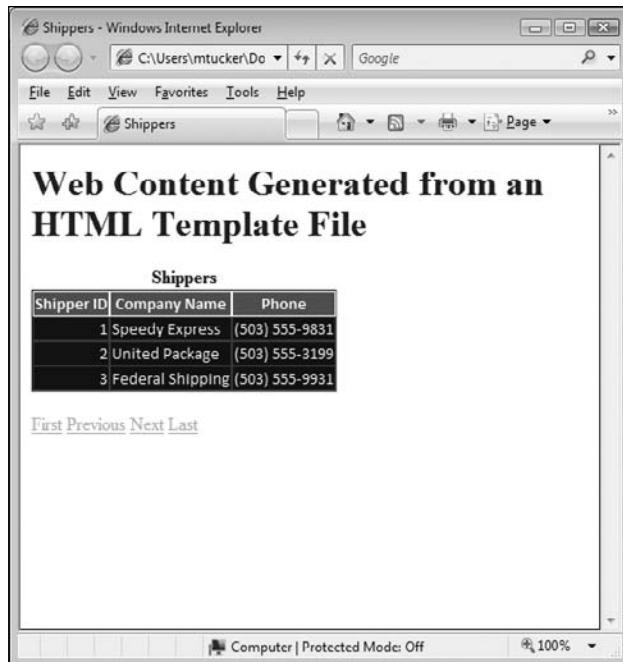


Figure 5-5

You may also have noticed that there are some useful tokens that don't appear in the preceding list, namely, tokens to reference the individual fields in the table, query, form, or report that you are exporting. And that is the drawback to using this template system: There is no way to control the layout of the body block. For tables and queries especially, you are unable to control the sizing of data columns, and you cannot specify conditional formatting for individual fields. Every field in the table will always appear in the same font, with a column that uses a default value you cannot control.

If you can live with the limited options for formatting the contents of the table, HTML export provides a very fast and efficient way to generate static HTML pages from your database. If, however, you need more control with your HTML output, you can maintain complete control by generating the HTML using VBA code. We look at one way to do this later in this chapter.

Why Create HTML Using Code?

Access provides built-in functionality to export data from your application into HTML formats, so why would you want to go to the bother of writing code to perform this task? Moreso than with other formats, HTML provides a much greater level of flexibility and customization for your data. There really aren't many things to change in a delimited data file (Do you want your strings quoted? Which delimiter do you want to use?)

HTML files generally provide some level of formatting to go with your data. With style sheets, you literally have an infinite number of ways to generate an HTML file to display your content. The HTML export feature in Microsoft Access does a good job of getting your data to HTML with a standardized format, but there is zero flexibility for formatting the data in the grid. If you want to have thorough control of your data formatting, you have to export your own HTML.

Part II: Data Manipulation

Creating HTML Files

For simple files, consider storing an HTML template that contains replacement tokens. You can run the template through the `StringFormat` function to replace the tokens with data from a table. Chapter 15 contains an example of generating HTML files for user documentation based on a template and data in a table.

Let's look at how you can re-create the HTML file we've been exploring with total control over the presentation of the data grid. Using code, you will be able to place tokens in the file multiple times (for example, a `TITLE` token in both the page title and in the body of the page), and exert complete control over the formatting of content in the data grid.

Our system involves three elements.

First, let's create a table to hold the HTML templates. Run the following DDL query to create the table:

```
CREATE TABLE [HtmlTemplates]
(
    ID autoincrement PRIMARY KEY,
    TemplateName text(50),
    HtmlTemplate memo,
    RowTemplate memo
)
```

Add a template to the table. Name the template Shippers Custom Template. Add the following text to the `HTMLTemplate` field:

```
<html>
<head>
    <title>Custom Formatted Export of <!--{TITLE}--></title>
    <style>
        body
        {
            background-color: black;
            color: papayawhip;
            font-family: 'Times New Roman', sans-serif;
        }
        th
        {
            background-color: papayawhip;
            color: blue;
            font-family: 'comic sans MS', sans-serif;
        }
        td
        {
            background-color: yellow;
            color: red;
            font-family: 'courier new', monospace;
        } hr { color: green; }
    </style>
</head>
<body>
    <h1>Web Content Generated from a home-grown HTML
    Template solution for object <!--{TITLE}--></h1>
    <h1>A table with headers</h1>
```

```

<table border=1>
<thead>
    <th><!--{Field(0).Name}--></th>
    <th><!--{Field(1).Name}--></th>
    <th><!--{Field(2).Name}--></th>
</thead><!--{BODY}-->
</table>
<hr/><h1>A table without headers</h1>
<table border=5><!--{BODY}--></table><br />
</body>
</html>

```

Add the following text to the RowTemplate field:

```

<tr>
    <td><!--{Field(0).Value}--></td>
    <td><!--{Field(1).Value}--></td>
    <td><!--{Field(2).Value}--></td>
</tr>

```

Create the following procedures. The BuildDocument function accepts the data recordset and the document template. It locates and replaces the field name tokens (<!--{Field(1).Name}-->) with the actual field names, and replaces the title token (<!--{TITLE}-->) with the name of the data recordset:

```

Public Function BuildDocument(ByRef rs As DAO.Recordset, template As String)
    Dim sReturn As String
    Dim cLoop As Long
    ' Default the return variable to the template we passed it
    sReturn = template

    ' If there is no data, there is nothing to do
    If rs Is Nothing Then GoTo Terminate
    ' Loop through the recordset
    For cLoop = 0 To rs.Fields.Count - 1
        sReturn = Replace(sReturn, _
            StringFormat("<!--{Field({0}).Name}-->", cLoop), rs.Fields(cLoop).Name)
    Next cLoop
    ' Replace the title token, if any
    sReturn = Replace(sReturn, "<!--{TITLE}-->", rs.Name)

    Terminate:
    BuildDocument = sReturn
    Exit Function
End Function

```

You will use the BuildBlock function to generate the rows of data that go into the table. The function is called with the recordset containing the data to place into the grid, the template used to build each row, and a block tag to indicate what kind of block you want to build (in our example, we want a TBODY block that will appear inside a table).

```

PPublic Function BuildBlock(ByRef rs As DAO.Recordset, _
    blockTemplate As String, blockTag As String) As String

```

Part II: Data Manipulation

```
Dim sReturn As String
Dim sRow As String
Dim cLoop As Long

If rs Is Nothing Then GoTo Terminate

Do While Not rs.EOF
    sRow = blockTemplate

    For cLoop = 0 To rs.Fields.Count - 1
        sRow = Replace(sRow, StringFormat("<!--{Field({0}).Name}-->", _
            cLoop), rs.Fields(cLoop).Name)
        sRow = Replace(sRow, StringFormat("<!--{Field({0}).Value}-->", _
            cLoop), rs.Fields(cLoop).Value)
    Next cLoop

    sReturn = sReturn & sRow
    rs.MoveNext
Loop

Terminate:
BuildBlock = StringFormat("<{0}>" & sReturn & "</{0}>", blockTag)
Exit Function

End Function
```

Finally, the CreatePageFromTemplate function accepts a TemplateID and a filename. It calls the previous functions and generates a complete HTML page that is saved to the specified filename.

```
Public Sub CreatePageFromTemplate(templateId As Long, fileName As String)
    Dim db As DAO.Database
    Dim rsData As DAO.Recordset
    Dim rsTemplates As DAO.Recordset

    Dim sDocument As String
    Dim sRow As String

    Dim hFile As Long

    // Build recordset with template information
    Set db = CurrentDb
    Set rsTemplates = db.OpenRecordset( _
        "SELECT * FROM [HtmlTemplates] WHERE [ID] = " & templateId, dbOpenDynaset)

    // Load templates from table
    sDocument = Nz(rsTemplates("HtmlTemplate"), vbNullString)
    sRow = Nz(rsTemplates("RowTemplate"), vbNullString)

    // Create recordset of data to add to template
    Set rsData = db.OpenRecordset("Shippers", dbOpenDynaset)

    // Replace tokens in document
    sDocument = BuildDocument(rsData, sDocument)
```

```
'// Build the data body
sRow = BuildBlock(rsData, sRow, "tbody")

'// Insert the Body into the document
sDocument = Replace(sDocument, "<!--{BODY}-->", sRow)

'// write the document to file
hFile = FreeFile
Open fileName For Output As #hFile
Write #hFile, sDocument

Cleanup:
On Error Resume Next
Close #hFile
rsData.Close
rsTemplates.Close
Set rsData = Nothing
Set rsTemplates = Nothing
Set db = Nothing

Terminate:
On Error GoTo 0
Exit Sub

End Sub
```

Figure 5-6 shows the custom-generated Shippers Web page that results from running this function.

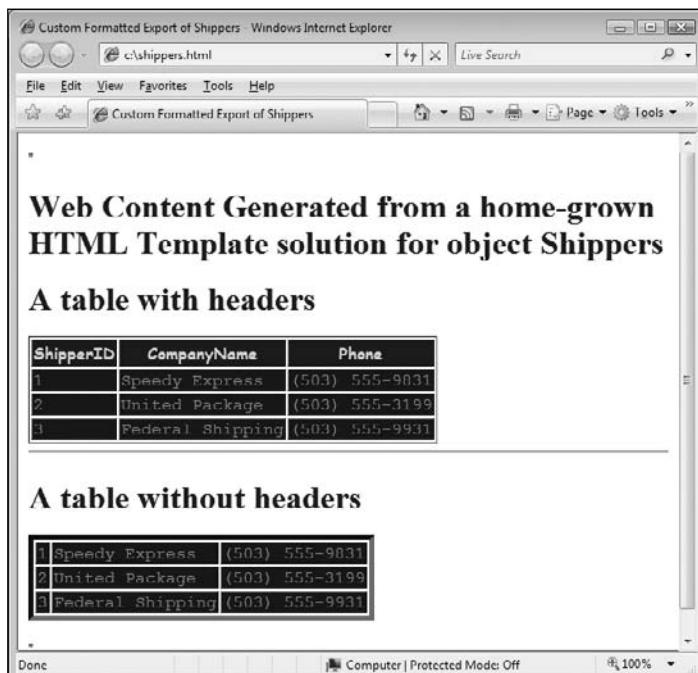


Figure 5-6

Summary

In this chapter, you looked at techniques for splitting data apart, strategies for designing your schema for granular data storage, and strategies for interacting with external text files. We showed you how to use VBA code to:

- Read and write external text files
- Split data apart and combine it together
- Isolate data within strings using VBA intrinsic string handling functions
- Create a library of token handling functions
- Create HTML files from code

The most important concept in this chapter is the importance of granularity in your data. This chapter presented several examples of how granularity can help you answer complex questions with ease, with specific suggestions on how to approach schema design.

We hope you are able to learn from our examples and design your schema with sufficient granularity from the start, so that you can avoid the most troublesome parsing problems. But when confronted with data that needs to be separated, we hope this chapter gets you going!

6

Using SQL to Retrieve Data and Manipulate Objects

Structured Query Language — more commonly referred to as SQL — is the language used by relational databases to define queries against a database. The database engine accepts requests created using the SQL language and returns a recordset or executes actions based on the values in the SQL request.

In this lengthy chapter, we examine the following:

- How to create select queries to return data from a database
- How to create action queries to add, modify, and remove data from a database
- How to create and work with subqueries
- How to create and work with crosstab and union queries
- How to use Data Definition Language queries to create and modify schema
- Some of the subtleties of working with ANSI-92 compatibility mode

The Microsoft Access database engine (commonly referred to as ACE) was introduced with Microsoft Access 2007. ACE is an update to the venerable Microsoft Jet database engine, which has powered all previous versions of Access, and which is shipped with every version of Microsoft Windows since Windows 98. ACE is completely backward compatible with Jet, and adds support for the multi-valued lookup fields (often referred to as complex data) introduced in Access 2007. With the exception of syntax added to interact with multi-valued lookup fields, the SQL syntax is identical for both Jet and ACE, and all of the sample queries presented in this chapter will work on either Jet or ACE unless we mention otherwise.

By default, both the Jet and ACE database engines use a variant of SQL based on the ANSI-89 standard, which varies slightly from the later ANSI-92 standards. Unless you are an experienced SQL Server developer accustomed to ANSI syntax, you will be more than satisfied with the

Part II: Data Manipulation

default syntax used by Access, and this is the language we will use in the examples throughout this book. The section “SQL Pass-Through Queries” describes the differences between standard Jet behaviors and ANSI mode behavior and how to tell the database engine you want to use ANSI-92 syntax.

Most of the examples in this chapter will be based on the Northwind Traders sample database that has shipped with every version of Access since version 1.0. Unfortunately, the Northwind 2007.accdb that ships with Access 2007 was completely revised and in the process, most of the useful samples — and more importantly, the useful data — were removed or made unrecognizable. We will be using the old Northwind.mdb file that ships with Access 2000, Access 2002, and Access 2003. This file is available for download from www.microsoft.com/downloads (search for northwind.mdb).

Where SQL Can Be Used in Access

SQL is used under the hood throughout Microsoft Access. Queries and views are based on SQL, as are forms and reports. Many properties on Access forms and reports also accept certain SQL elements.

Every query in an Access database resolves to a SQL string. Access provides arguably the best interactive graphical query designer on the market today, but in the end, the result is a SQL string. Many users are completely satisfied using the graphical designers to create their queries. However, advanced users will encounter certain queries that can only be accomplished by creating a SQL request manually. After working directly with the SQL syntax for a while, many users find they can accomplish their tasks more quickly and efficiently by skipping the graphical designer altogether.

Form and Report Recordsets

Most forms and reports in Access are based on some set of data. How does that data get to the form or report? A query, of course, and as you know, all queries come down to SQL.

Each form and report has a `RecordSource` property, which defines where the data for the form or report originates. This property can contain one of three things: the name of a table object in the database, the name of a saved query object in the database, or a SQL string.

By saving the record source as a SQL string, you can reduce the number of stored queries in the database (reducing clutter), as well as ensure that changes to stored queries don’t inadvertently break forms and reports.

Partial SQL Properties

Many properties throughout Access accept a partial SQL string. Table and query objects each have a `Filter` property that accepts a SQL `WHERE` clause. These properties do not take a complete SQL statement; rather, they take just a `WHERE` clause that defines which rows are to be included in the query.

Likewise, an `OrderBy` property accepts a SQL `OrderBy` clause that specifies how data in the object should be sorted when it is displayed.

We will look at these properties in further detail when we start discussing these clauses.

The Various Types of SQL Queries

Several different types of queries can be created using SQL. SQL language features are often divided into two categories: Data Definition Language (DDL) and Data Manipulation Language (DML). In practice, the distinction is not important.

Suffice it to say that DDL is used to create schema objects: tables, indexes, relations and more. In most cases, you will probably find it faster and more convenient to create your schema objects using the Access user interface. However, if you need to create schema objects from code, DDL sometimes provides a more concise way to perform simple tasks than the corresponding DAO or ADO code.

DML is the syntax used to write standard queries that report on the data contained within a database, and is what is generally meant when someone refers to SQL. Most of this chapter is devoted to DML syntax for querying and interacting with data stored in Access databases.

The Access Query Designer can design six different types of DML queries. Access provides a friendly name to describe each type of query in the Query Designer. Each type of query has a related SQL statement that is used when a query is created using SQL.

Access Query Type	SQL Statement	Description
Select query	SELECT	Display existing records from an existing table.
MakeTable query	SELECT INTO	Create a new table from data selected from existing tables.
Append query	INSERT INTO	Add new records to an existing table.
Update query	UPDATE	Update existing records in an existing table.
Delete query	DELETE	Remove existing records from an existing table.
Crosstab query	TRANSFORM	Display existing records from an existing table.

The Access Query Designer can be used to create three additional types of queries that require the use of SQL. *Union queries* allow you to combine multiple Select queries into a single recordset. *SQL Pass-Through queries* are queries that can be run remotely on a Microsoft SQL Server. Finally, *data definition queries* enable you to create database objects using SQL's Data Definition Language (DDL).

Whenever a query is created graphically in the Access Query Designer, Access is building a SQL query under the hood. You can always see exactly what SQL was generated for a specific query by switching to the SQL view in the Query Designer. This is one of the best ways to learn how to create well-formed SQL: Use the graphical Query Designer in Access to create a query, and then switch to SQL view to see the SQL string that was created. The Access Query Designer always generates fully qualified, well-formed SQL.

To switch to From the Home ribbon, choose SQL View from the View drop-down in the Views group, as shown in Figure 6-1.

Part II: Data Manipulation

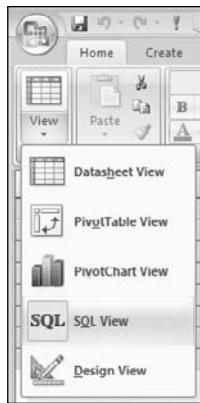


Figure 6-1

Or you can press the SQL View button in the Views selector (as shown in Figure 6-2), located in the lower-right corner of the Access application window.

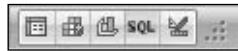


Figure 6-2

The **SELECT** Query

The **SELECT** query is the workhorse of any relational database system. A select query is used to return a set of rows from one or more tables in your database. A **SELECT** query can also use other select queries as the source of data in a query, just like a table.

A **SELECT** query can return all rows in a table, or a filtered set of rows based on criteria you specify. They can aggregate and sort your data (or not). They are able to return the results of complex expressions, and can call user-defined functions. They are virtually infinitely flexible, and you will use them constantly in your applications.

SELECT queries return their results in a recordset. If you are running the query from code, you can use the resulting recordset directly from your code to walk through the results. When the query is run from inside Access, the results are displayed in a grid, where you can scroll through the data visually.

The “Hello World!” of the SQL realm looks something like this:

```
SELECT * FROM [MyTable]
```

This instructs the database engine to return every field of every row contained in the database object named **MyTable**. This query contains all of the required elements of a **SELECT** query: the **SELECT** SQL keyword, a field list, and a **FROM** clause.

A **SELECT** query consists of one or more of the following elements. Every **SELECT** query must have a field list and most have a **FROM** clause; you can add any of the other clauses to add additional features to your query.

Query Element	Description
SELECT [Field List]	SQL Keyword and Field List (required)
FROM [Table List Join Specification]	From Clause (Data Source; REQUIREDOptional)
WHERE [WHERE Clause]	WHERE Clause (Filtering; optional)
GROUP BY [GroupBy Clause]	GroupBy Clause (Aggregation; optional)
HAVING [Having Clause]	Having Clause (Filtering on aggregated data; optional)
ORDER BY [OrderBy Clause]	Order By Clause (Sorting; optional)

We will look at each of these elements in detail. Let's start with the field list.

Project Your Columns: the Field List

The *field list* is where you specify the *columns* you want to display in the resulting recordset. The field list may contain fields from tables and other select queries, as well as expressions that are evaluated and displayed in a column. Each item included in the field list will be displayed as a column in the recordset. If you want a field to appear in the resultset, the field must appear in the field list. The field list must have at least one item in the list, and may include a maximum of 255 items.

Individual fields from any source table may be added to the field list. The asterisk (*) wildcard character is used to include all of the fields from a source table in the field list. Expressions can be included in the field list. Expressions are evaluated and the results are displayed in the resultset.

If a field or expression doesn't appear in the field list, it won't be displayed in your query results. However, you can use fields and expressions that do not appear in the field list elsewhere in your query to restrict the rows that you want to return, or to sort or group your data.

The list of field names is often called the projection, and you will often hear people talk of projecting a column, field, or expression; this just means that the column, field, or expression is included in the field list and will appear in the results set.

Let's look at each of the things that can go in a field list.

Add a Field Using Field Names

Individual fields are added to a query by simply typing the name of the field in the field list. To specify more than one field, separate each field with a comma. For example, to add the FirstName, LastName, and BirthDate fields from the Employees table, you would write:

```
SELECT [FirstName], [LastName], [BirthDate]
FROM [Employees]
```

Part II: Data Manipulation

Fully Qualified Field Names

Field names can be specified in several different ways. In the previous examples, we used the name of the field without any adornments because there was no ambiguity as to which field we wanted in our query. There was only one table, and field names within a table must be unique. Consider, however, what happens if there are two tables in a query, each of which contains a field named `ID`, and the `ID` field from Table 1 in your query needs to be displayed:

```
SELECT [ID]
FROM [Table1], [Table2]
```

Because both tables have a field named `ID`, the database engine will not know which `ID` field to display: the one from `Table1` or the one from `Table2`. When this query is run, Access displays the following error message:

The specified field 'ID' could refer to more than one table listed in the FROM clause of your SQL statement.

To disambiguate this query, you must specify which table the field belongs to, like this:

```
SELECT [Table1].[ID]
FROM [Table1], [Table2]
```

This is known as a *fully qualified field name* because you have specifically indicated which table the field belongs to.

You do not need to fully qualify your object names unless there is some ambiguity. The Access Query Designer always generates fully qualified field names to ensure that the queries it generates will always be able to compile and run regardless of any schema changes that might occur. A fully qualified name can always be resolved.

Whenever a field name belongs to more than one table in a query, the field name must be fully qualified.

To ensure your queries will run regardless of any schema modifications that may occur in the future, always fully qualify your field names.

Add All Fields Using the Wildcard Character

The asterisk (*) wildcard character is used to automatically include every field in a table in the resultset. When you use the wildcard character, the field list will be generated dynamically each time the query is compiled, and will include every field in the source table(s). If the schema of the table is later altered to add or remove fields, the query will be recompiled, so that the query will continue to display every field in the table as the table exists when the query is run.

The following query will display every field in the `Customers` table, for every row in the table.

```
SELECT *
FROM [Customers]
```

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

The asterisk wildcard can also be fully qualified to a specific table. When you add the asterisk wildcard to a field list, by default, every available field from every table or query in the query table list is included in the resultset. If you only want to display every field from a particular table, you can fully qualify the wildcard to specify the table name. For example, the following SQL returns every field from the `Orders` table, and only the `Employee ID` from the `Employees` table:

```
SELECT [Orders].*, [Employees].[Employee ID]
FROM [Orders] INNER JOIN [Employees] ON
[Orders].[Employee ID] = [Employees].[Employee ID]
```

Using the asterisk wildcard is a quick and easy way to create a query, and it can be tempting to use it in all of your queries. There is a good reason why you should think twice before using this wildcard in your production queries: performance.

When you run a query that includes the asterisk wildcard in the field list, you are returning every possible column from the source table. It is a rare query that needs to return every column. If you return columns you don't need, you increase the amount of time it takes to compile and execute your query (generally this performance penalty is negligible), but more important, if you are running over a network, you increase the amount of traffic on the wire. If you return data you won't use, not only are you slowing down the response of the query, but you are also increasing the load on the network. Depending upon the number of unused columns and number of rows returned in your query, this additional data load can be substantial, and result in a noticeable decrease in performance.

Appendix B discusses this and other query performance optimizations.

Add Expressions

Finally, expressions can be included in the field list. The expression may be a constant ("Hello! ") or an expression that displays the results of a function (`Time()`). The expression may even reference the value of one or more fields in your data sources. Let's look at an example in the Northwind sample database.

For example, in the Northwind database, we need to create a query against the `Employees` table. We want to show the full last name, but only the first initial of every employee, as well as the year they were hired. The following query satisfies these requirements:

```
SELECT "Hire Year", [LastName], Left$([FirstName], 1), Year([HireDate])
FROM [Employees]
```

The following table shows how the resulting recordset would appear.

Expr1000	LastName	Expr1002	Expr1003
Hire Year	Tucker	M	1991
Hire Year	Cooper	R	1996
Hire Year	Walther	M	2003

Part II: Data Manipulation

In this example, the constant expression "Hire Date" will appear in the first column of each row in the resulting recordset. Because it is a constant expression, it is not dependent upon any data in the row, and will always appear the same on each row.

The field name [Last Name] specifies that the contents of the Last Name field of each row will appear in the second column.

The expression `Left$([First Name], 1)` uses the `Left$` VBA function to return the first character from the First Name field of each row in the third column. Because this expression is dependent upon the First Name field, the value of the expression is different on each row. When each output row is created, the value of the First Name field for that row is passed into the `Left$` function, and the output from the function is displayed in the cell.

The expression `Year([Hire Date])` uses the `Year` VBA function to return the four-digit year from the Hire Date of each row in the fourth field. Again, this expression is dependent upon a field (the Hire Date field), so each output row passes the value of this field into the `Year` function, and the result of the function is placed into the cell.

Notice that a column name was automatically generated for each column in the resultset. Access uses the name of the field that is displayed in a column as the column name. A random name is generated and used for columns that result from an expression. We specify an alias to customize the name that is generated. We will explore aliases shortly.

Let's look at another example using slightly more complex expressions. Let's parse the `ContactName` field of the `Customers` table into separate first and last name fields. You use the `Left$`, `Mid$`, `InStr$`, and `Trim$` VBA functions to split this field on the space between the first and last names.

```
SELECT Trim$(Left$([ContactName], InStr([ContactName], " "))),  
       Mid$([ContactName], InStr([ContactName], " ") + 1)  
FROM [Customers]
```

The `First Name` column is constructed by locating the space in the `ContactName` field using the `InStr` function. This function returns the position of the first instance of a space in the field. This value is passed into the `Left$` function, and the resulting string is passed into the `Trim$` to remove any leading or trailing spaces.

The `Last Name` column is constructed by again locating the space using the `InStr` function, and passing that value (incremented by one so we don't get a leading space) into the `Mid$` to return all characters that appear after the space.

Square Bracketing field names is particularly important when string math is used to create SQL strings. Consider the following VBA code, which creates a SQL string:

```
sqlString = "SELECT * FROM " & tableName
```

If the table name does not contain any embedded spaces (as with the `Customers` or `Employees` table), this code will generate a valid SQL string. However, if the table name contains an embedded space (as with the `Order Details` table), a runtime error will occur when the SQL is executed.

SQUARE BRACKETS and SQL Object Names

Square brackets are used to delineate object names in SQL code. If an object name contains an embedded space or punctuation mark (for example, a field named "Last Name"), the field name must be surrounded with square brackets. If you forget the square brackets, one of several error messages will result when the query is executed.

Brackets may always be used, whether or not an object name contains embedded spaces or punctuation. We strongly recommend always using square brackets when creating object names in code on-the-fly, or when object names are passed as parameters. If you use string math to generate SQL strings, using square brackets will ensure that you always generate a well-formed SQL statement.

LastName	This field name contains no spaces or special characters and so brackets are not required. However, square brackets are always valid and may be used even when not required.
[LastName]	Square brackets are always valid and may be used even when not required.
[Last Name]	Because there is a space in this object name, square brackets are required.

The following VBA code resolves the problem by building SQL with square brackets around the object name:

```
sqlString = "SELECT * FROM [ " & tableName & " ]"
```

Always use square brackets when using string math to generate object names.

Why Spaces Are Evil

Although spaces are perfectly valid in your database object names, they result in a lot of extra work for the developer and power user. As we have just discussed, when object names contain spaces, the use of square brackets becomes compulsory. This doesn't sound like a big deal, until you start having to type them — lots of them, throughout your code, in every query. Everywhere! Power users and developers will curse your name each and every time they have to reach for those inconveniently located keys on the keyboard.

One of the main reasons for using spaces is readability of your object names, but there are ways to preserve readability while still avoiding spaces. Avoid cryptic or vowel-phobic abbreviations (lstm, dtrcvd, posslq). Use full word names with Initial Capitals (for example, LastName, DateReceived), or use an underscore instead of the space (first_name, Company_Name).

Spaces can be used in the Caption property to provide the friendly name — with spaces — in the column headers of datagrids, and as the default label for associated controls on forms and reports. This gives you the best of both worlds: friendly, spaced out labels for users, and friendly, no-bracket object names in queries and code!

Avoid using spaces in your object names or alias names.

Part II: Data Manipulation

Field Name Aliases

Often, you may want to create an alternate name for a field in the field list. This might be necessary to display a meaningful title for an expression, or when two fields from two different tables have the same name.

An alias is simply an alternative name for the field. If an alias is defined for a field list item, it will be displayed as the name of that column in the resultset. An alias may be defined for any field in the field list (except for the * wildcard character). Aliases may contain up to 64 characters.

To create a field name alias in the Access Query Designer, enter the alias name followed by a colon (:) into the field cell immediately before the field name or expression. To add an alias in SQL code, add the reserved word AS following the expression or field name you want to alias, and then add the alias name. If the alias name contains any spaces, the alias name definition must be enclosed in square brackets. Adding an alias to a field in the Access Query Designer is shown in Figure 6-3.

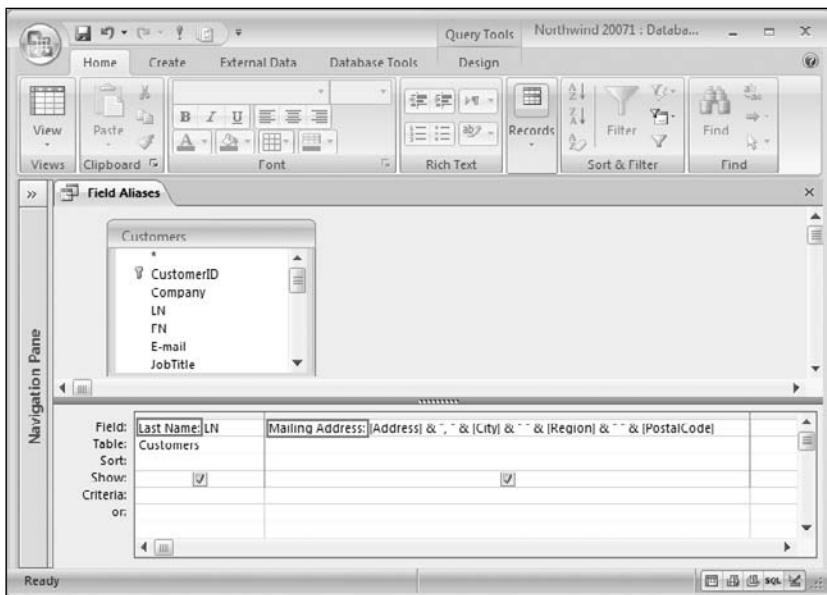


Figure 6-3

The following query defines an alias for each of the fields in the field list:

```
SELECT [Last Name] AS [LN], [First Name] AS [Customer's First Name]
FROM [Customers]
```

Just like field names themselves, you must enclose an alias definition in square brackets if it contains any spaces. It also must be unique among all the items in the field list.

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Let's add aliases to each of the expressions in the query we built earlier, so that we can display meaningful column names instead of the randomly generated names you saw earlier.

```
SELECT "Hire Date" AS [Hire Date],  
       [LastName] As [Last Name],  
       Left$([FirstName], 1) AS [First Initial],  
       Year([HireDate]) AS [Year Hired]  
  FROM [Employees]
```

The following table shows how the resulting recordset would appear.

Hire Date	Last Name	First Initial	Year Hired
Hire Date	Davolio	N	1992
Hire Date	Fuller	A	1992
Hire Date	Peacock	M	1993
Hire Date	King	R	1994

Choose Your Tables: The *FROM* Clause

The `FROM` clause is used to specify which tables or queries contain the data for a query. Data may be selected from any table or query inside an ACE or Jet database, as well as any table, stored procedure, or view in any supported external database format, such as an Excel spreadsheet, SQL Server or Oracle database.

In its simplest form, you can choose the name of a single table, like this:

```
SELECT *  
FROM [Customers]
```

This instructs the query engine to base the query on the data in the `Customers` table.

Frequently, a query will need to reference data in more than one table. This can be done by using a table join specification. The *join specification* instructs the query engine how to combine the tables together when selecting data. It is the SQL representation of the join lines in the Query Designer. There are several types of joins, and we discuss each of them in detail later in this chapter. For now, let's take a look at a simple example using the default join type, the inner join.

The following query will display the company name, order number, and order date for every order in your orders table. Because the name of the company that placed the order is not contained in the orders table, you join to the `Customers` table, using the common join field `CustomerID`:

```
SELECT [Customers].[CompanyName], [Orders].[OrderID], [Orders].[OrderDate]  
FROM [Customers] INNER JOIN [Orders] ON [Customers].[CustomerID] =  
[Orders].[CustomerID]
```

This query produces a recordset similar to the following table.

Part II: Data Manipulation

Company Name	Order ID	Order Date
Alfreds Futterkiste	10643	25-Aug-1997
Alfreds Futterkiste	10692	03-Oct-1997
Ana Trujillo Emparedados y helados	10308	18-Sep-1996
Antonio Moreno Taquería	10365	27-Nov-1996
Around the Horn	10355	15-Nov-1996

Table Name Aliases

As we mentioned earlier, an alias can be supplied for any table in the `FROM` clause by adding the `AS` reserved word and the name of the alias. This allows you to refer to the table using the alias name. As with aliased fields, this feature can be handy either to shorten references to tables (in the case of obnoxiously long table names) or to add clarity to the query (as in the case of obnoxiously obtuse table names).

```
SELECT O.ID, B.ID  
FROM [Office] AS O, [Branch] AS B  
SELECT *  
FROM [CusXtr200804] AS [April2008CustomerExtract]
```

When an aliased table name is referenced in the field list or another SQL element, the table alias must be used instead of the base table name. The following query results in a parameter value prompt for `[Customers].[CustomerID]` because the `Customers` table is aliased, but the alias was not used in the field list.

```
SELECT [Customers].[CustomerID]  
FROM [Customers] AS [C]
```

Both of the following queries will work, the first one because it uses the table alias, and the second because it does not directly reference a table name (and because there is no ambiguity as to where the field `Customer ID` must come from, it does not need a table name reference).

```
SELECT [C].[CustomerID]  
  
FROM [Customers] AS [C]                      SELECT [CustomerID]  
  
FROM [Customers] AS [C]
```

If you have long table names and need to fully qualify your field names, adding an alias for a table can significantly shorten the SQL string and make it easier to read and debug. The following SQL string is difficult to read:

```
SELECT [Very Long Table Name].[Long Field Name 1],  
      [Very Long Table Name].[Long Field Name 2]  
FROM [Very Long Table Name]
```

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

You can add an alias to the table name to shorten the SQL string.

```
SELECT [V].[Long Field Name 1],  
       [V].[Long Field Name 2]  
  FROM [Very Long Table Name] AS [V]
```

To add an alias to a table name from the Access Query Designer, display the property sheet for the table in the Query Designer, and enter the alias name in the Alias property for the table. Adding an alias to a table in the Access Query Designer is shown in Figure 6-4.

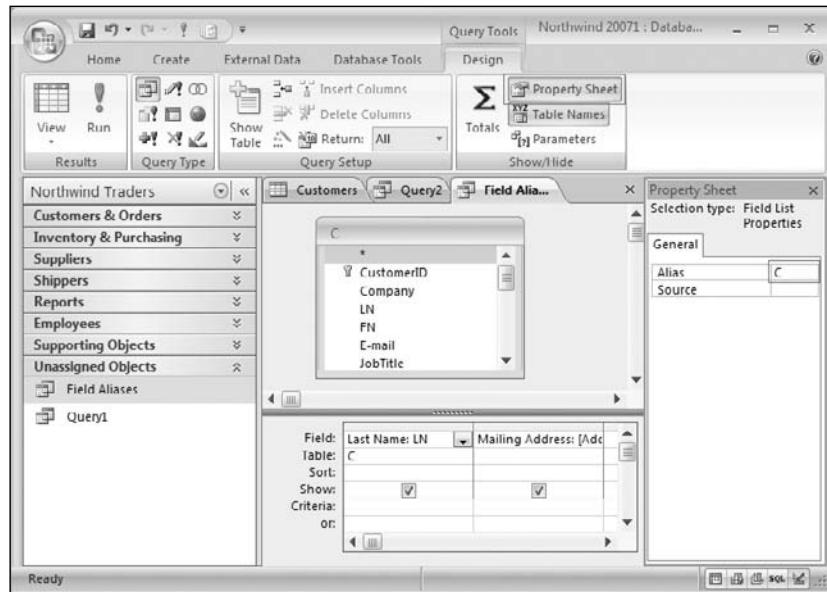


Figure 6-4

Table aliases are very useful when working with table joins, which we will discuss shortly.

Filter Your Data: The WHERE Clause

The `WHERE` clause is used to filter or restrict the rows that are displayed in your query. Many (if not most) of your queries will need to show only a subset of the data in your tables. For example, you may wish to show only the orders that need to be fulfilled, or list only your unpaid invoices. You use a `WHERE` clause to specify which rows you want to view in your query. If you do not include a `WHERE` clause in your query, no filtering will occur, and all rows in the source table(s) will be displayed. If you need to restrict the rows that are displayed, based on criteria you specify, add a `WHERE` clause.

The WHERE clause is also frequently referred to as the criteria clause, and all of the conditions in the WHERE clause are collectively called the query criteria.

In its simplest form, a `WHERE` clause consists of the `WHERE` keyword, followed by one or more conditions. When you run the query, the data in each row is compared to the condition, and only those rows that

Part II: Data Manipulation

satisfy the condition are included in the recordset. You can combine conditions by using the AND and OR comparison operators, and tests can be nested using parentheses to create tests of virtually infinite complexity. You can have up to 40 expressions linked together with AND and OR operators in a single query.

The following query has a WHERE clause with one condition. The condition tests the FirstName field and will include the row if the FirstName is Nancy.

```
SELECT *
FROM [Employees]
WHERE [FirstName] = "Nancy"
```

A test in the WHERE clause can work with fields and expressions. The following query has a WHERE clause with one condition. The condition is an expression, which uses the Left\$ VBA function to extract the first three characters of the postal code on each row, and uses the IN clause to see if those three characters exist in a list.

```
SELECT *
FROM [Customers]
WHERE Left$(Nz([PostalCode]), 3) IN ("980", "981")
```

When you include a field in an expression, the expression will be evaluated on each and every row. In this query, for each row in the Customers table, the query engine will compare the Postal Code of that row to the list. If the three characters from the field do exist in the list, the condition is satisfied, and the row will appear in the results list. Otherwise, the condition is not satisfied, and the row will not be included in the results list.

You can specify more than one test by using the AND or OR logical operators. This query shows all customers who have a First Name of either Nancy or Andrew.

```
SELECT *
FROM [Employees]
WHERE [FirstName] = "Nancy" OR [FirstName] = "Andrew"
```

You can use parentheses to group conditions, and specify order of precedence, just as in an algebraic expression. The following query shows all customers who are not in the United States, and any customers in California whose company name begins with the letter M:

```
SELECT *
FROM [Customers]
WHERE [Country] <> "USA" OR ([CompanyName] Like "M*" AND [Region] = "CA")
```

Comparison Operators

The following table lists each of the comparison operators you can use when creating WHERE conditions.

Operator	Purpose
<	Less than
<=	Less than or equal to

Operator	Purpose
>	Greater than
\geq	Greater than or equal to
=	Equal to
\neq	Not equal to
BETWEEN <i>lowValue</i> AND <i>highValue</i>	Range comparison
IN (<i>item1</i> , <i>item2</i> , ..., <i>itemn</i>)	List comparison

Matching on Strings

When you want to compare against a string value, you must delimit the string using either a double quote character ("") or a single quote character ('').

```
SELECT [ContactName]
FROM [Customers]
WHERE [CompanyName] = "La maison d'Asie"
```

If the string you are delimiting itself contains the delimiter character, you must escape each instance of the character within your string. To escape a quote character, you enter two adjacent quote characters in the string instead of one. For example, in the previous example, the string we are comparing against contains a single quote in the string. This is not a problem when we delimit the string with double quotes as we did in the example. But if we chose to use single quotes as our string delimiter, we would need to escape the single quote inside the string, like this:

```
WHERE [Company Name] = 'La maison d''Asie'
```

Matching on String Patterns Using Wildcard Characters

You can use the `LIKE` operator in a `WHERE` clause to perform pattern matching on strings. For example, if you want to match any record where the `Company` field begins with the letters `MI`, you can use the `LIKE` operator with the `*` (asterisk) wildcard character:

```
SELECT *
FROM [Customers]
WHERE [Company] Like "MI*"
```

The following table lists the wildcard characters you can use with the `LIKE` operator.

Wildcard Character	Meaning
<code>*</code> (Asterisk)	Match zero or more characters.
<code>?</code> (Question Mark)	Match exactly one character.
<code>#</code> (Hash Mark)	Match exactly one digit.

Part II: Data Manipulation

Wildcard Character	Meaning
[]	Matches exactly one character listed between the brackets.
[!]	Excludes exactly one character listed between the brackets.

Use the asterisk (*) to match zero or more characters in the string. A question mark (?) matches a single character only. The hash mark (#) matches numeric digits in the string. You can include multiple instances of the wildcard to create complex patterns.

For example, the following expression will match on Gray or Grey but not Greey:

```
WHERE [LastName] LIKE "GR?Y"
```

whereas

```
WHERE [LastName] LIKE "GR*Y"
```

will match Gray, Grey, Greey, and Gry.

You can mix and match pattern matching wildcard characters:

```
SELECT [ContactName]
FROM [Customers]
WHERE [ContactName] LIKE "M??R*"
```

This query will show only those customers who have a contact name that begins with *M*, followed by any two letters, followed by an *R*, followed by zero or more additional characters. If you run this query against northwind.mdb, one row is returned for Maurizio Moroni.

Use square brackets inside the string to specify a set or a range of characters to match. The following code

```
SELECT *
FROM [Employees]
WHERE [LastName] LIKE "[D-F]*"
```

returns each employee whose last name begins with the letter *D*, *E*, or *F*. Three employees match this filter (Davolio, Dodsworth, and Fuller). The following code

```
SELECT *
FROM [Employees]
WHERE [LastName] LIKE "[DFS]*"
```

returns each employee whose last name begins with either *D*, *F*, or *S*. This returns Davolio, Dodsworth, Fuller, and Suyama. You can use an exclamation point (!) to exclude characters in a range.

```
SELECT *
FROM [Employees]
WHERE [LastName] Like "[DFS] [ !M-P]*"
```

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

This pattern selects all employees whose last name has a *D*, *F*, or *S* in the first position, and which does not have an *M*, *N*, *O*, or *P* in the second position. This excludes the record for Dodsworth, which does have an *O* in the second position. When you specify a range, you must specify characters in ascending order (0 to 9, A to Z). If we had changed the specification to

```
WHERE [LastName] Like "[SDF] [!P-M]*"
```

the record for Dodsworth is *not* excluded because the range is not in ascending order (*M* comes before *P*). If you are not specifying a range of characters, the order is not significant ([*SDF*] returns the same results as [*DFS*]).

The following pattern will identify license plates that have a single digit followed by three letters followed by three digits:

```
WHERE [LicensePlate] LIKE "#[A-Z][!#][!0-9]###"
```

The patterns *[A-Z]*, *[!#]*, and *[!0=9]* are used interchangeably in this example, but there is a subtle difference. The first pattern will match only the letters from A to Z (international characters with diacritical marks are also matched), whereas the second and third patterns (which are equivalent) will match any non-digit. So, if our table has a record with a license plate of

```
1AB$123
```

this record will be matched because we specified that the fourth character needs to match any non-digit pattern. If we used the *[A-Z]* pattern for the third and fourth character positions, only alphabetical characters will be matched, and the plate with the \$ symbol will be excluded from the query results.

A character range can be used to search for wildcard characters that may be embedded in data. For example, add a customer to the Northwind database with a company named “Why not?! Expresso”. The following query uses a character range pattern to locate this record:

```
SELECT *
FROM [Customers]
WHERE [CompanyName] LIKE "*[?]*"
```

For more details on complex string pattern matching expressions, search help for “Like Operator”.

When working with a database in ANSI 92 mode (SQL Server Compatible Syntax-enabled), you must use the ANSI wildcard characters with the ALIKE (Ansi-LIKE) operator. In ANSI 92 mode, use the percent sign (%) in lieu of the asterisk () and the dot(.) in lieu of the question mark (?). When ANSI 92 mode is enabled, Access will automatically replace LIKE with ALIKE, but it does not automatically convert the wildcard characters.*

Matching on Data in a Range Using the BETWEEN . . . AND Operator

You will often want to create queries that match data that falls in a defined range. This is especially common with numeric data and date/time values. The Jet and ACE database engines provide the BETWEEN...AND operator to allow you to quickly compare against a range of data.

Part II: Data Manipulation

To use this operator in a WHERE condition, specify the field name or expression you want to compare, followed by the BETWEEN...AND operator, filling in the low and high value of the range.

```
SELECT [ProductName], [UnitsInStock]
FROM [Products]
WHERE [UnitsInStock] BETWEEN 0 AND 5
```

The BETWEEN...AND operator is inclusive, so values that match either endpoint of your range will be included in your results list. In the previous example, any product that has a UnitsInStock value of 0 or 5 are included in the results list, too.

Null vs. vbNullString

There is an important distinction between Null and vbNullString. The two are not equivalent in any way. Worse still, vbNullString is never Null!

Null represents the empty set. In a recordset, Null indicates the absence of data. It's the theoretical third option in a Boolean field (yes, no, and unknown), and the missing data everywhere else.

vbNullString represents an empty string. This is distinctly different from Null. An empty string exists — it just doesn't have anything in it. Null doesn't even exist. There's no string, no object at all.

Despite the poor choice of name, vbNullString is an enormously valuable constant that we recommend enthusiastically. It makes code dramatically easier to read, and reduces confusion resulting from use of the wrong type of quote (particularly troublesome if you work on SQL Server or in C/C++ where strings are delimited by single quote characters) or the dreaded “smart quotes” feature in Microsoft Word and other advanced word processors.

The vbNullString constant is available in VBA code only; the constant is not recognized inside a SQL string.

Matching on Null Values and Empty String Values

Queries frequently need to identify rows that contain (or do not contain) Null values. This is easy to accomplish in SQL, but there is a gotcha you need to be aware of. You cannot compare for equality against Null. Null represents an empty set, and because the set is empty, there is nothing to compare to! Instead, SQL provides the Is operator to test for Null values.

To test for Null, specify Is Null as the comparison. For example, if you use the ShippedDate field to track which orders have actually been shipped, you could identify all your pending orders with the following query:

```
SELECT *
FROM [Orders]
WHERE [ShippedDate] Is Null
```

This query returns each record where the ShippedDate value is Null. To reverse the logic and identify the rows where the ShippedDate field is not Null, simply add the Not keyword.

```
SELECT *
FROM [Orders]
WHERE [ShippedDate] Is Not Null
```

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

The `IsNull` VBA function is another way to test for `Null` values. It accepts an expression and returns `True` if the expression evaluates to `Null`. This is particularly useful inside the `IIF` (Intermediate IF) VBA function when you want to perform some conditional activity based on whether or not a field (or expression) is `Null`.

```
IsNull([Field])
```

If `[Field]` is `Null`, the `IsNull` function returns `True`; otherwise, it returns `False`. The argument takes an expression, so in addition to field references from a recordset, you can pass in variables, function calls, and constant expressions.

```
IsNull("Hello " & "World!")  
IsNull(MyCustomFunction("arg1", 2))
```

For example, it is very common to see queries that concatenate name elements together to form a full name for a form or report. Consider the following expression:

```
SELECT UCASE([LastName]) &  
      " " & [FirstName] &  
      " " & UCASE([MiddleInitial]) &  
      ". (" & [DOB] & ")" AS NameDOB  
FROM [Persons]
```

This expression combines the Last Name, First Name, Middle Initial, and Date of Birth data into a single column. The results are shown in the following table.

DOE, Elena . (11/11/1950)
SMITH, Seán . (12/12/1977)
JONES, Kitt D. ()
WALTHER, Markus O. (01/01/1980)
PETIT, Christopher . ()

There are two problems with this result. If there is no Middle Initial, we get an extra space and a period in the output, and if there is no Date of Birth, we get empty parentheses. We can use the `IsNull` and `IIF` VBA functions to fix both of these problems.

The `IIF` function accepts three arguments: a condition that evaluates to `True` or `False`, an expression to return if the condition is true, and optionally, an expression to return if the condition is false. Let's look at a simple example.

```
age = 20  
Debug.Print IIF(age >= 18, "Adult", "Minor")
```

When this code runs, the conditional expression `age >= 18` is evaluated. If the conditional expression evaluates to `True` (and in this example, it will because the `age` variable was set to 20), The second argument is evaluated and returned (in this example, the second argument is a constant string, and so the constant string is returned, and "Adult" is printed to the Immediate window).

Part II: Data Manipulation

When the conditional expression evaluates to `False`, the optional third argument is evaluated and returned when present. If you change the value of age to 15 and rerun the `IIF` expression, "Minor" will be printed to the Immediate window. If the conditional expression evaluates to `False` and the third argument is omitted, the `IIF` expression returns `Null`.

Now that we know how the `IsNull` and `IIF` functions work, let's use them to fix the Date of Birth parentheses problem.

```
SELECT UCASE([LastName]) &
      " " & [FirstName] &
      " " & UCASE([MiddleInitial]) & "." &
      IIF(Not IsNull([DOB]), " (" & [DOB] & ")") AS NameDOB
FROM [Persons]
```

Here, we place the code that surrounds the Date of Birth with parentheses in the second argument of an `IIF` function call. Our condition argument checks to see if the `DOB` field is not `Null` (i.e., do we have a Date of Birth value). If the `DOB` field is not `Null` (we *do* have a Date of Birth), we will surround the value of the `DOB` field with parentheses. If the `DOB` field is `Null`, the `IIF` function will return `Null` because we did not specify the third argument.

The results of this revised query are shown in the following table. The Date of Birth is shown only if a value exists in the record.

```
DOE, Elena . (11/11/1950)
SMITH, Seán . (12/12/1977)
JONES, Kitt D.
WALTHER, Markus O. (01/01/1980)
PETIT, Christopher .
```

To remove the extra space that appears if there is no value in the `MiddleInitial` field, we will again use an `IIF` function. But this time, instead of using `IsNull` to test the field, we will use the `Nz` VBA function.

The `Nz` function (also known as the Null-to-Zero function) is useful to ensure that an expression always has a value. If the expression evaluates to `Null`, the `Nz` function returns either 0, an empty string, or a value of your choice. The function accepts two arguments, an expression to evaluate, and an optional value to return if the expression is `Null`.

If the expression does not evaluate to `Null`, `Nz` returns the result of the evaluation.

If the expression results in `Null` and you specified a value in the second argument, the value of the second argument is returned. If you do not specify the second argument, `Nz` will try to determine whether or not a numeric or string value should be returned. If it determines a numeric value is appropriate, `Nz` returns 0; otherwise it returns an empty string (`vbNullString`, or `" "`). If you use `Nz` in a query, `Nz` will always return an empty string if the expression evaluates to `Null`. If you don't want an empty string returned, we recommend that you explicitly set the second argument to the default value you need returned.

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

In each of the following cases, if [Field] is not Null, the value of [Field] will be written to the Immediate window.

```
Debug.Print Nz([Field])           '// returns an empty string or 0 if myVar Is Null
Debug.Print Nz([Field], 0)         '// returns 0 if myVar Is Null
Debug.Print Nz([Field], "{NULL}")  '// returns "{NULL}" if myVar Is Null
Debug.Print Nz([Field], -999)      '// returns -999 if myVar Is Null
```

Let's see how we can use the Nz function to remove the extraneous space. As in the previous case, we move the code that concatenates the space to the Middle Initial field value into the second argument of an IIF function, so that it will be executed only if our condition evaluates to True.

Next, we create the conditional expression for the IIF. The expression is:

```
Trim$(Nz([MiddleInitial])) <> ""
```

This expression passes the value of the MiddleInitial field to the Nz function. If the field is Null, the Nz function will return an empty string to the Trim\$ function. In that case, Trim\$ will return an empty string (because when you Trim\$ an empty string, there are no characters to return); this will cause our comparison to fail:

```
SELECT UCASE([LastName]) &
      " " & [FirstName] &
      IIF(Trim$(Nz([MiddleInitial])) <> "", 
          " " & [MiddleInitial] & ".") &
      IIF(Not IsNull([DOB]), " (" & [DOB] & ")") AS NameDOB
FROM [Persons]
```

Because our comparison fails, IIF will *not* run the concatenation code in the second argument. Because we did not specify an expression in the third argument, the IIF returns Null. The following table shows the results of this final query: there are no extraneous spaces and dates appear only when they exist in the field!

DOE Elena (11/11/1950)

SMITH Seán (12/12/1977)

JONES Kitt D.

WALTHER Markus O. (01/01/1980)

PETIT Christopher

You may be wondering why the Trim\$ function is needed here. Trim\$ will ensure that we don't get an extra space if the MiddleInitial field is not Null, but contains an empty string or only space characters. If we don't remove leading and trailing spaces from the field, we still get the extra space (or multiple extra spaces if the MiddleInitial field contains multiple space characters). The Access grid will not allow you to enter extra spaces (it essentially performs a trim for you when you enter data into the grid), but if you import data from an external source that has spaces, or if your data was generated by code that (accidentally or not) appended spaces to the field, you could get unintended and unwanted white space in your results.

Part II: Data Manipulation

So that explains the Trim\$ function, but why do we need to use the Nz function? Quite simply, the Trim\$ function will fail with an “Invalid Use of Null” error if you pass a Null value into it. Using Nz guarantees that a non-Null value is passed into the Trim\$ function.

Matching on Dates

When you want to restrict rows based on data in a date/time field, you should use the hash mark (#) to delimit the date values. Also, the values you specify must be in standard U.S. data format (#mm/dd/yy#), even if you are using a localized version of Access. For example, to select order dates greater than April 26, 2008, you would write the following SQL statement:

```
SELECT *
FROM [Orders]
WHERE [OrderDate] > #4/26/2008#
```

Alternatively, you can use the DateValue VBA function, which is aware of the regional settings that are defined on your system and which will correctly match to dates specified using another culture. For example, if your regional settings are set up to format short dates using dd/mm/yy instead of mm/dd/yy, the following query will return the same data as the previous example, even though the date criteria is specified differently:

```
SELECT *
FROM[Orders]
WHERE [OrderDate] > DateValue("26/4/2007")
```

The DateValue function requires a string argument, so you should not use hash marks to delimit dates that you pass to this function.

You may also match dates in a range using the BETWEEN...AND operator. The following query returns all orders in Northwind placed between August 1, 1996 and September 15, 1996.

```
SELECT *
FROM [Orders]
WHERE [OrderDate] BETWEEN #8/1/1996# AND #9/15/1996#
```

Matching on GUID Values

Jet and ACE databases allow you to store GUID (Globally Unique Identifier) values in a numeric field with the field size value set to Replication ID. It is named this way because Access uses GUIDs internally for replicated databases, but this field may be used any time you need to store GUID values.

When you need to restrict rows based on a value in a Replication ID field, you specify the matching GUID value in the following special format:

```
{guid {01234567-89AB-CDEF-0123-456789ABCDEF}}
```

When you specify this {guid {...}} format, you must include a valid guid value between the braces. You cannot use pattern matching wildcards directly against a GUID field comparison. For example, the following query is invalid:

```
SELECT *
FROM [GuidTable]
WHERE [GuidField] LIKE {guid {*-89AB-*}}    ' // INVALID!
```

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

If you need to perform pattern matching against a GUID field, you can wrap the GUID field in a `CStr` expression to first convert the GUID to a valid string, and then apply pattern matching against the string. The following query will return the rows you expected from the previous query:

```
SELECT *
FROM [GuidTable]
WHERE CStr([GuidField]) LIKE "*-89AB-*"
```

Matching to Items in a List (IN Operator)

Use the `IN` operator to match data against a list of values. If you have a finite list of matching values, you can specify a comma delimited list of valid choices. When the query runs, for each row in your source data, the value of the field or expression in your condition is compared to the items in your `IN` list. If the item is found in the list, the condition is satisfied and the row will be included in the results list. If the item is not found, the row will be excluded from your results.

The following query will return all `Orders` placed by the three customers listed in the `IN` list:

```
SELECT *
FROM [Orders]
WHERE [Customer] IN ("ALFKI", "LETSS", "SANTG")
```

`IN` also works against lists of numbers and dates:

```
SELECT *
FROM [Orders]
WHERE [Order ID] IN (12332, 12357, 12369, 12243, 12992)
```

To use the `IN` operator when building a query using the Access Query Designer, simply type the `IN` keyword into the criteria row under the field you wish to filter against and include a comma-delimited list of items. Using the `IN` operator in the Access Query Designer is shown in Figure 6-5.

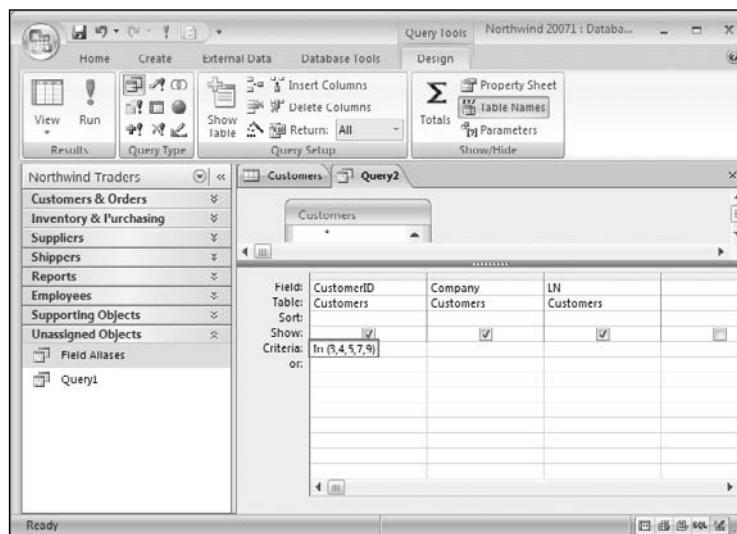


Figure 6-5

Part II: Data Manipulation

The IN operator is convenient when you have a list of data from another program that you need to query against. For example, you may find a list of part numbers on a Web page or in a PDF catalog that you want to query against. Simply copy those values to the clipboard, and then paste them in to an IN list in your query. You may need to massage the data in Notepad or another text editor to remove extraneous characters or punctuation, but this is often faster and more accurate than typing the data in by hand.

The IN operator can be used to match against the results of another query. We will discuss this in detail in the section on subqueries later in this chapter.

Sort Your Results: The ORDER BY Clause

Sorting your data makes it easier to identify patterns in your data. The ORDER BY clause is used to set the sort order for a query. One or more field names and/or expressions may be included, and for each item, a separate ascending (A to Z, 0 to 9) or descending (Z to A, 9 to 0) sort direction may be specified.

Order By affects only the presentation of the data and does not alter which rows are selected for display.

SQL allows sorting on expressions as well as field data. Add fields and expressions to the ORDER BY clause in the order you want your results sorted. The query results will be sorted on the first field in the ORDER BY clause, and then within each previously sorted group, by each additional field in the order specified.

The fields in an ORDER BY clause may appear in a different order from the fields specified in the field list. It is even possible to sort results based on data that is not displayed in the resulting recordset.

If you don't add an ORDER BY clause to your query, the SQL language does not guarantee that results will appear in any particular order. However, the Jet and ACE database engines will try to sort your data based on the primary key indexes you have defined on your tables. If there are no primary keys defined, then the data is generally returned in the order that the data was added to the database. This order is not guaranteed, however, so if you need your data to always appear in the same order each time you run the query, you should explicitly request a sort by adding an ORDER BY clause.

Setting the Sort Direction

The sort direction can be set for each field independently. By default, SQL will always sort in ascending order (A to Z, 0 to 9, earliest to latest). If you wish to sort in descending order (Z to A, 9 to 0, most recent to most distant), you must append the DESC predicate to the sort expression.

Although it is possible to explicitly request an ascending sort order by appending the ASC predicate to a sort expression, Access will remove the ASC predicate if you take your query into the Access designer and then subsequently save the query. Because of this, we usually don't bother explicitly specifying ascending sort orders.

To set the sort order in the Access Query Designer, choose either Ascending or Descending in the drop-down list that appears on the Sort row of your query. Access will apply sorts in order from left to right in the query grid.

If you wish to specify a sort on a field that you do not want to appear in the results list, add the field to the grid, set the sort order, and clear the Show check box in the grid. When writing SQL in the SQL View window, any valid field or expression may be added to the ORDER BY clause regardless of whether it exists in the field list with one exception. If the query contains a GROUP BY clause, then only fields and expressions that exist in the field list may be added to the ORDER BY clause.

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Let's look at an example of a moderately simple query with multiple sorts:

```
SELECT Trim$(Left$([ContactName],InStr([ContactName]," "))) AS [First Name],
       Mid$([ContactName],InStr([ContactName], " ")+1) AS [Last Name],
       [Country]
  FROM [Customers]
 WHERE [Country] IN ("USA", "FRANCE")
 ORDER BY [Country],
          [Region] ASC,
          Mid$([ContactName],InStr([ContactName], " ")+1) DESC
```

This query shows the First Name and Last Name of the customer, as well as their Country. We use expressions in the field list to split the ContactName field into separate First Name and Last Name columns in the query using the same expressions you saw earlier. We use a WHERE clause to restrict the results to show only customers located in the United States and France.

The data in the resulting recordset is sorted first by country. All the customers in France will be listed before the customers in the United States because the default sort order is Ascending, and France precedes USA alphabetically.

Within each country, the records are sorted by Region (even though we didn't ask to show the Region column!), again in ascending order. Notice that this time, we have explicitly requested the Region field be sorted in ascending order, by adding the ASC predicate after the field name; we could have left this off and it would still sort in ascending order.

Finally, the data is sorted within each region by Last Name. The Last Names will appear in descending order (Z to A) because we added the DESC predicate to request descending sort order.

The resulting recordset appears similar to the following table.

First Name	Last Name	Country
Daniel	Tonini	France
Annette	Roulet	France
Martine	Rancé	France
Laurence	Lebihan	France
Janine	Labrunе	France
Rene	Phillips	USA
Jaime	Yorres	USA
Jose	Pavarotti	USA
Liu	Wong	USA

Continued on next page

Part II: Data Manipulation

First Name	Last Name	Country
Paula	Wilson	USA
Fran	Wilson	USA
Art	Braunschweiger	USA

You may be wondering why we specified the `Last Name` field sort on the expression rather than on the field alias name. Although you can use a field alias name as arguments to expressions in the field list, you cannot use them anywhere else. When you need to refer to the results of an expression that you created in the field list anywhere else in the query, you must re-create the expression. It would be very nice to be able to say `[Last Name] DESC` in the `ORDER BY` clause, but if you do, you'll be prompted to enter a parameter value (we will discuss parameter values later in this chapter) and you won't get the sort you expect.

Sorting on Binary or Memo Fields

OLE Object fields cannot be added to the `ORDER BY` clause, and errors may occur when certain large memo and hyperlink fields are included in the `ORDER BY` clause. If an error occurs when sorting on binary, memo, or hyperlink fields, one workaround is to use the `CStr$` VBA function to convert the field data to a string, and then sort on the results of the `CStr$` expression.

To sort on a memo field that is too large to be sorted directly, consider using the `Left$` VBA function to return the first part of the memo field. Often, this will provide enough data to give a meaningful sort.

```
SELECT * FROM [Employees] ORDER BY Left$([Notes], 50)
```

If you need to sort on data embedded deep within a memo field, you can use the `Mid$` function in combination with the `Instr` function to isolate a string that you can sort on. For example, if you have a memo field that contains text information imported from an enterprise application, you may be able to search on data deep in the field using an expression that locates a known token (in this example, the string `CustomerID`), and returns enough data to accomplish your sort.

```
SELECT * FROM [CustomerExtract]
ORDER BY Mid$([SAPExtractText], Instr([SAPExtractText], "CustomerID:"), 20)
```

Add Data from More Than One Table Using Table Joins

So far, all of the examples we have looked at involve selecting data from only one table. However, it is frequently necessary to combine data from more than one table. SQL makes this possible by using table join specifications.

Many people find joins intimidating, but once you grasp the concept, they really aren't that difficult.

A table join allows you to link two tables together on some common piece of information. For example, you may have an `Employees` table that lists data about your employees, and an `Orders` table that contains

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

information on each of your orders. If you ever need to combine data from these tables together (as you are very likely to on an invoice, or an order entry form), you need to make sure there is a common link between the two tables. For this scenario, the common link will likely be an `EmployeeID` field.

The common fields do not need to have the same name (although it is generally considered a best practice to have them share the same name), but they should share a common data type, and they must represent similar data in the same way (Employee 1 in the `Employees` table is Nancy Davolio; orders with `EmployeeID = 1` should be placed by Nancy Davolio).

Any time you have a common link between two tables, you can create a join between the tables. The join is represented with a table join specification. The basic form of any join is:

```
[LeftTableName] jointype [RightTableName] ON  
[LeftTableName]. [LeftCommonField] = [RightTableName]. [RightCommonField]
```

where `jointype` is one of `INNER JOIN`, `LEFT JOIN`, or `RIGHT JOIN`.

In addition to an equal comparison, you can use any of the logical comparison operators (except `BETWEEN...AND` and the `IN` operators) to build the join; however, the equal join is by far the most common relation you will create.

A join may be created on more than one field by combining multiple joins together with an `AND` operator.

```
[LeftTableName] jointype [RightTableName] ON  
([LeftTableName]. [LeftCommonField1] = [RightTableName]. [RightCommonField1] AND  
[LeftTableName]. [LeftCommonField2] = [RightTableName]. [RightCommonField2])
```

Inner Joins

An *inner join*, also known as an *equi-join*, combines two tables together where there is matching data in both tables. In our example of `Employees` and `Orders`, an inner join will combine records in the two tables where there is both an employee and an order. A record will be created for each employee that has orders and for every order that is associated with an employee. But if an employee didn't place any orders, that employee will not show up in the results. And if any orders don't have an associated employee, those orders will not show up in the results.

Using our `Employees/Orders` example, let's look at a basic inner join query:

```
SELECT [Employees]. [EmployeeID], [Employees]. [LastName], [Orders]. [OrderID]  
FROM [Employees] INNER JOIN [Orders]  
ON [Employees]. [EmployeeID] = [Orders]. [EmployeeID]
```

This query will join the two tables based on the common `EmployeeID` field. In this example, the `Employees` table is our left table, and the `Orders` table is our right table. In the `ON` clause, we link the two tables in the same order they appear in the join statement.

With this join specification, the query engine will match up records in each table, combining the matching employee record from the `Employees` table with each matching order from the `Orders` table. Sample data used to illustrate join types is shown in Figure 6-6.

Part II: Data Manipulation

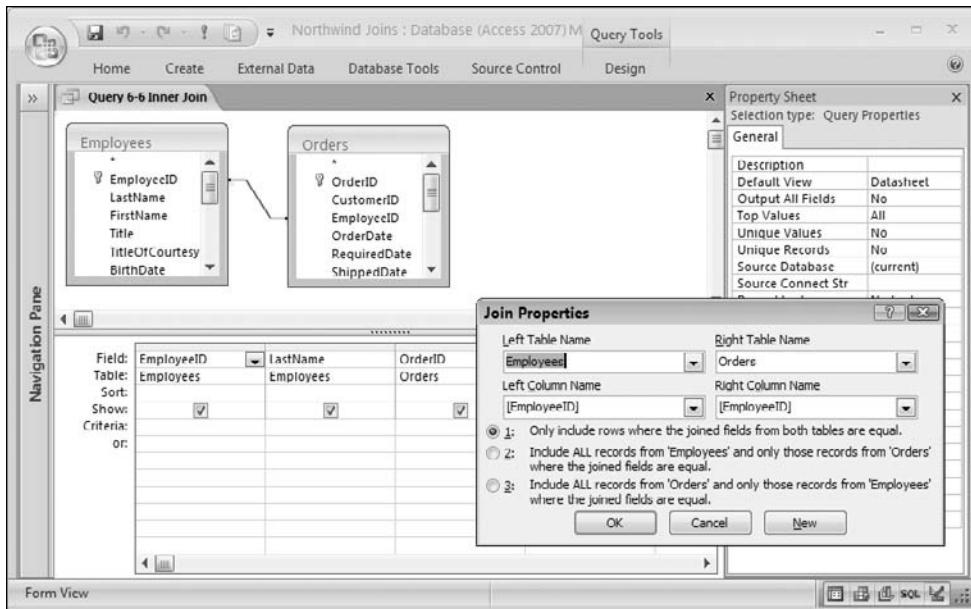


Figure 6-6

Outer Joins

An outer join returns all of the records in one table, and only the matching records in the other table. For example, if you have an `Employees` table and an `Orders` table joined in a query, you might use an outer join to show every employee in the recordset, and each order (if any) placed by the employee. With an inner join, you would see only the employees that actually have orders in the `Orders` table. With an outer join, you can ask for every single record in one of the tables (`Employees`), and additionally display the matching data in the other table (`Orders`).

There are two types of outer joins: a *left join* (also referred to as a *left outer join*) and a *right join* (sometimes referred to as a *right outer join*). A left join selects all of the records in the left table, and if there happen to be matching records in the right table, the matching records are also included. If there are no matching records in the right table, null values will be returned for any fields selected from the right table.

A right join is the opposite of a left join. It selects all of the records in the right table, and if there happen to be matching records in the left table, the matching records are also included. As with the left join, if there are no matching records in the left table, null values will be returned for any fields selected from the left table.

A left outer join is shown in Figure 6-7.

What is the Left table? It's the first table, listed on the left side of the join type in the join expression. If you build a query visually using the SQL designer in Access, it's the table you dragged from when creating the link between tables. Conversely, the Right table is the second table, listed on the right side of the

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

join type. In the example that follows, Employees appears on the left side of the “Left Join” join type and so it is the left table, and Orders is the right table.

```
... FROM Employees Left Join [Orders] ON ...
```

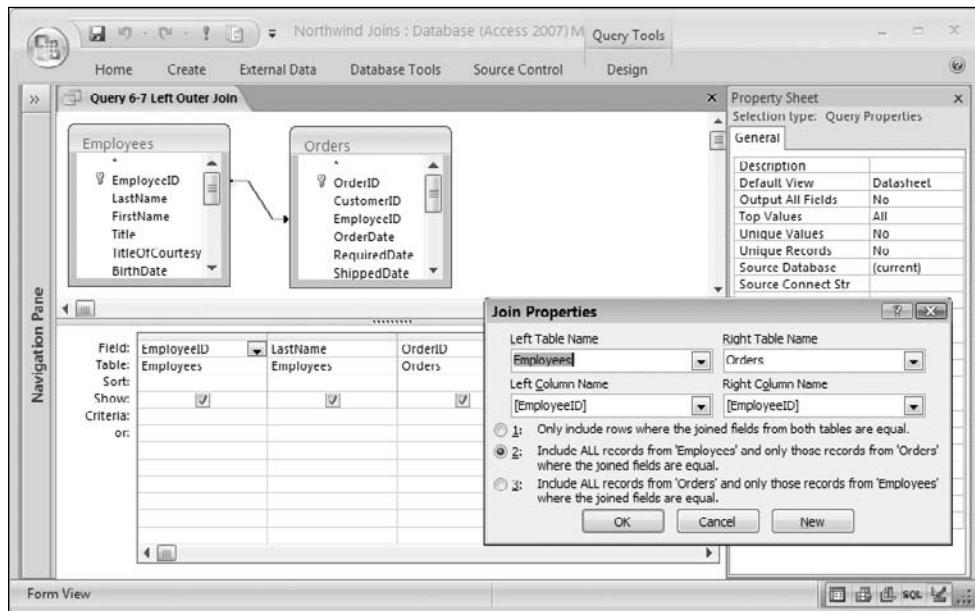


Figure 6-7

When building a query graphically, the right table is the table you dragged to when creating the link. If you design a query graphically and Access creates an automatic join between two tables, Access will make the left table the one side, and the right table the many side of the relationship.

In the Access Query Designer, an outer join is represented by an arrow, pointing from one table into the other table. A left Outer Join is represented graphically as an arrow pointing from the left table to the right table. The arrow points from the table that will have all records displayed to the table that contains matching data that will be displayed only if matching data exists.

A right outer join is the opposite of a left outer join. In a right outer join, all of the records in the right table are selected, and any matching rows from the left table are also included. If there is no matching row in the left table, null values will be returned for fields selected from the left table.

Our Employees Right Outer Join Orders example is shown in Figure 6-8.

You can choose the type of outer join from the Access Query Designer from the Join Properties dialog box. Double-click on a join line in the Access Query Designer, or right-click on any join line and choose Join Properties to view this dialog box, as shown in Figure 6-9.

Option 1 always creates an inner join. Option 2 always creates a left join, and Option 3 always creates a right join.

Part II: Data Manipulation

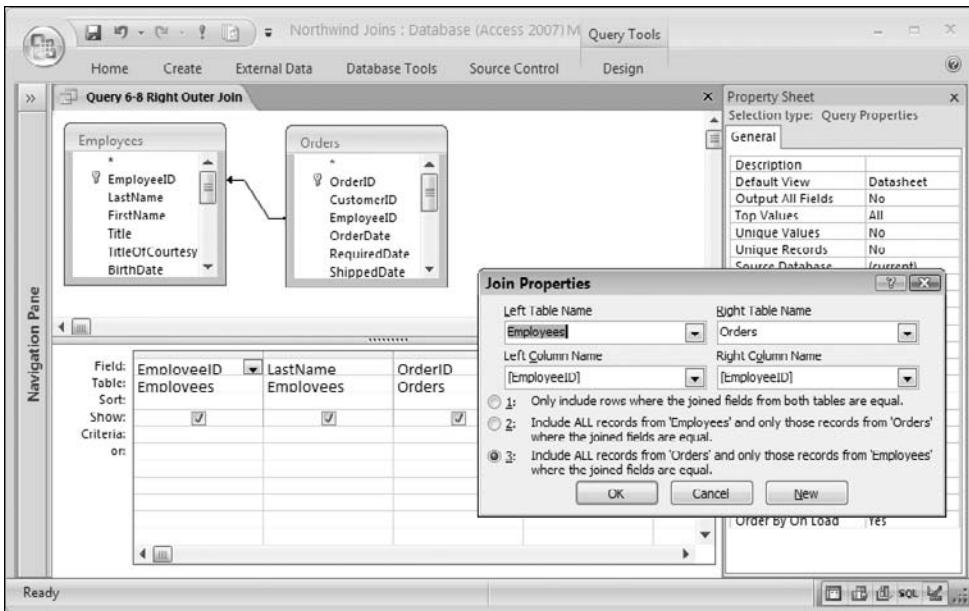


Figure 6-8



Figure 6-9

Joining More Than Two Tables Using Nested Joins

Although you can join only two tables in a single join expression, you may have up to 16 joins in a single query, by nesting join expressions. To nest a join, create a standard join expression between two tables (it can have multiple fields in the join, but only the left and right tables). Once that join expression is created, treat that join expression as the left table of the second join expression!

Put parentheses around the join expression (not required, but it simplifies debugging and makes your SQL easier to read later on) and then add a join type and right table name, followed by the ON keyword and your field list for the second join expression.

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Let's walk through an example. We'll start with a simple query against the `Customers` table:

```
SELECT [Customers].[CustomerID], [Customers].[CompanyName]
FROM [Customers]
```

This query simply lists each customer and his or her company name.

Next, we add the `Orders` table. We want to see every customer listed, and if they have any orders, some information about each order. If they don't have any orders, we still want to see the customer information. This sounds like an outer join (because we want all records from one table and matching records from the other table). So let's modify our query to add a left outer join to the `Orders` table.

```
SELECT [Customers].[CustomerID], [Customers].[CompanyName],
       [Orders].[OrderID]
  FROM [Customers] Left Join [Orders] ON
       [Customers].[CustomerID] = [Orders].[CustomerID]
```

We added the `OrderID` field from the `Orders` table to our field list because we want to see the `OrderID` in the resulting recordset. Then we built a left join between the `Customers` and `Orders` table. Because the `Customers` table appears on left side of the join type declaration, `Customers` is the left table and `Orders` is our right table. Because we asked for a left join, this query will display all rows from the left table (`Customers`) and matching rows from the right table (`Orders`).

That sounds like what we have asked for! Run the query to verify you are seeing the data you expect. (`Customers` FISSA and PARIS have no orders, so the `OrderID` column for those customers has a Null value.)

Now, let's find out who shipped each order. We will modify our query further, to add a join to the `Shippers` table:

```
SELECT [Customers].[CustomerID], [Customers].[CompanyName],
       [Orders].[OrderID], [Shippers].[CompanyName]
  FROM (
    [Customers] Left Join [Orders] ON
         [Customers].[CustomerID] = [Orders].[CustomerID]
  ) Left Join [Shippers] ON
       [Orders].[ShipVia] = [Shippers].[ShipperID]
```

We added the `CompanyName` field from the `Shippers` table to our field list so that we can see the name of the Shipping Company. Notice that we now have two fields named `CompanyName` in our field list. This is not a problem because we have disambiguated the `CompanyName` reference by fully qualifying it with the associated table. Try removing the table name and running the query; you will receive an error message indicating that the field could refer to more than one table in the `FROM` clause of the query.

Notice that the field names that create the join between `Orders` and `Shippers` are not the same. This is perfectly fine, and very common in real-world applications. The important thing is that the data types of the fields are compatible (this would not work if one was a string value containing alphanumeric codes and the other was a Guid) and that the data maps correctly if we have numeric codes — as we do in Northwind — that a 1 in the `ShipVia` field corresponds to the shipper identified by `ShipperID` 1 in the `Shippers` table.

Part II: Data Manipulation

Run this query to verify you are getting the expected results. You will see null values in the CompanyName field from the Shippers table for every order that has not yet shipped (or in the event a shipper code was entered into an order that doesn't exist in the Shippers table). Finally, we mentioned earlier that table aliases can be useful when working with joins. Complicated SQL queries can grow in size quickly, making debugging a bit troublesome. By using short table alias names, the size of the SQL can be reduced, and especially when working with large numbers of tables, make reading the SQL a bit simpler.

The following is our final three-table join query rewritten to use short, one-character table aliases:

```
SELECT [C].[CustomerID], [C].[CompanyName],
       [O].[OrderID], [S].[CompanyName]
  FROM (
    [Customers] AS C Left Join [Orders] AS O ON
      [C].[CustomerID] = [O].[CustomerID]
  ) Left Join [Shippers] AS S ON
    [O].[ShipVia] = [S].[ShipperID]
```

When a table is aliased, the alias name must be used instead of the full table name everywhere except in the initial declaration of the table (which is immediately followed by the `AS` keyword).

If any of the tables contain multi-valued lookup fields, the maximum limit of 16 joins in a query is reduced by one for each MVF field in each table in the query, regardless of whether or not you use the MVF field in the query. This is because Access creates a hidden join under the covers for each MVF field in a table.

Self Joins

A *self join*, also known as a self-referential join, refers to a normal inner join or outer join, where both the left and right tables are the same. They are often used to represent hierarchical relationships contained within a single table.

For example, the Employees table contains an EmployeeID field, which identifies the employee, and a ReportsTo field, which contains the EmployeeID of their manager. We can use a self-join to include information on the manager as well as information on the employee.

To create a self join in SQL, create a standard inner join or outer join query, but use the same table name for both the left table and the right table. You will need to alias one of the tables (generally the second instance) to ensure that all object names are unique and that the query processor can properly resolve all the objects in your field.

The following query displays the employee name and the manager's name:

```
SELECT [Employees].[EmployeeID],
       UCASE$(NZ([Employees].[LastName], "")) & " " & [Employees].[FirstName] AS [Employee],
       [Employees].[ReportsTo],
       [Manager].[EmployeeID] AS [ManagerID],
       UCASE$(NZ([Manager].[FirstName], "")) & " " & [Manager].[LastName] AS [Manager Name]
  FROM Employees LEFT JOIN Employees AS Manager
    ON [Manager].[EmployeeID] = [Employees].[ReportsTo]
```

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Because we used a `LEFT JOIN` here, Andrew Fuller appears in the results list, even though Andrew is at the top of the food chain and does not have a manager himself. If we change the `LEFT JOIN` to an `INNER JOIN`, Andrew will be dropped from the list because there is no value in Andrew's `ReportsTo` field, and so there is no matching record in the `Manager` table.

To create a self join in the Access Query Designer, add the table to the Query Designer. Then, add it a second time. Consider providing a meaningful alias for the second instance (in this example, the first instance retains the table name `Employees`, while the second instance is aliased as `Manager`). Click on the common field in the first instance (`ReportsTo`) and drag it over the join field in the second instance (`EmployeeID`). Then, drag fields from either table into the grid to create the query you need. You can adjust the type of join by double-clicking on the join line to launch the Join Properties dialog box.

Cartesian Products

A *Cartesian product*, also known as a Cartesian join or a cross-product, is a special type of join that returns permutations of records in both tables. They return every possible combination of rows from two tables. There is no common field and no attempt is made to match records between the two tables. The need for a Cartesian product (also known as a Cartesian join) is extremely rare, generally occurs in very specific circumstances. If you need one, you'll more than likely know you need one.

You create a Cartesian product in SQL by specifying two or more tables in the `FROM` clause without linking them with an `OR` operator:

```
SELECT *
FROM [Customers], [Orders]
```

Except for a very few specialized cases, most Cartesian products are created when you forget to properly join the tables in your query. If you run a query and it returns an unexpectedly large number of rows (orders of magnitude more than you expected), or your query is taking so long to run you think it may have crashed, you probably have an accidental Cartesian product, as shown in Figure 6-10. Press `Ctrl+Break` to cancel the query and check your design for valid join clauses.

One scenario in which a Cartesian product is useful is when you need to pre-populate a table with template records to be filled out at a later time. For example, you may want to track customer contacts during a marketing campaign. You want to contact each customer three times during this campaign with a specific message. An Append query based on a Cartesian product between the `customers` table and a template table (which contains three template records) can populate the `contacts` table with three contact records for each customer will accomplish this task quickly and efficiently.

To illustrate this scenario, run the following Data Definition Language query to create a table to track customer contacts in the `Northwind` database (Data Definition Language syntax is discussed in depth later in this chapter).

```
CREATE TABLE CustomerContacts
(
    ContactID AUTOINCREMENT PRIMARY KEY,
    CustomerID text(5),
    ProjectID long,
    EmployeeID long,
    ContactType text(255),
```

Part II: Data Manipulation

```
Completed bit,  
DueDate datetime,  
DateCompleted datetime,  
Notes memo  
)
```

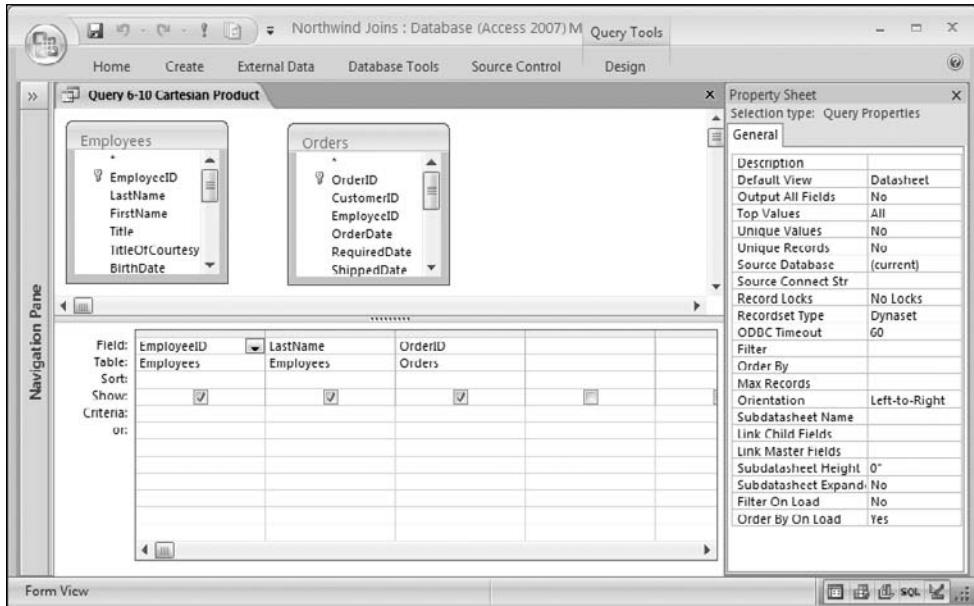


Figure 6-10

Next, run the following DDL query to create a table to hold the template records:

```
CREATE TABLE CustomerContactTemplates  
(  
    ProjectID long,  
    EmployeeID long,  
    ContactType text(255),  
    DueDate datetime,  
    Notes memo  
)
```

Add the following three rows to the CustomerContactTemplates table.

ProjectID	EmployeeID	ContactType	DueDate	Notes
1		Welcome To Northwind Call		
1		One Week Follow up call		
1		One Month Renewal Reminder		

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Finally, create the following Append query (we discuss Append queries in detail later in this chapter):

```
INSERT INTO [CustomerContacts]
(
    [CustomerID], [EmployeeID], [ContactType], [DueDate], [ProjectID], [Notes]
)
SELECT
    [CustomerID], [EmployeeID], [ContactType], [DueDate], [ProjectID], [Notes]
FROM
    [Customers], [CustomerContactTemplates]
```

This query appends three rows (one for each row in the CustomerContactTemplates table) for each customer in the Customers table into the CustomerContacts table. Once the CustomerContacts table is populated, it can be used by the employees to track who still needs to be contacted during this marketing campaign.

Prompt Users for Data with Parameters

Sometimes you need to create a query that relies on external data provided by your users when they run the query. Or, you may want to pass parameters to your query from code. SQL allows you to create queries with parameters that can be filled in with data each time the query is run.

A parameter may be used in a field list or anywhere a field name can be used (WHERE clause, ORDER BY clause, GROUP BY clause, or HAVING clause). When you run a query that contains a parameter, a dialog box is displayed with the name of the parameter, and a text box to enter a value in. You should construct parameters with names that make good prompts. The following query contains a parameter named "Enter Country Name."

```
SELECT *
FROM [Employees]
WHERE [Country] = [Enter Country Name]
```

When the query is run interactively, Access will display the parameter name in a dialog box. Your user can enter a value for the parameter, and the query will use the value entered by the user to prepare the resultset. Figure 6-11 shows an Enter Parameter Value dialog box.

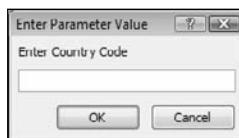


Figure 6-11

From DAO, you can explicitly set the value of the parameters in the query prior to calling the OpenRecordset or Execute method.

Part II: Data Manipulation

Form and Report Parameters

You can add parameters that reference the value of a control on a form or report. These parameters take the form:

```
Forms! [FormName]! [ControlName]  
Reports! [ReportName]! [ControlName]
```

When the query processor sees a parameter name in this format, it checks to see whether or not the referenced Form or Report is open in the current Access environment. If it is, it will load the parameter with the value of the specified control. If the form or report is not open, or if the specified control name does not exist in the object, the parameter will be treated as a standard parameter, and a parameter value dialog box will be displayed.

The following query will display all orders placed by the employee who placed the order that is currently being displayed in the Orders form. If the Orders form is not opened, then a parameter value prompt will be displayed.

```
SELECT [OrderID], [EmployeeID]  
FROM [Orders]  
WHERE [EmployeeID] = [Forms]! [Orders]! [EmployeeID]
```

Improve Performance and Readability with the Parameters Declaration

You can improve the performance of your queries by explicitly declaring your parameters. This instructs the query processor what data types each parameter will be, and allows the processor to pre-compile portions of your query. Declaring your parameters also ensures that your queries are more readable and self-documenting.

To declare parameters in SQL, you use a `PARAMETERS` declaration. This declaration appears as the first item in your query, immediately preceding the `SELECT` keyword. It consists of the name of the parameter and the data type. If you have more than one parameter in your query, define each one in a single `PARAMETERS` declaration, separating each definition with a comma. Finally, you must insert a semi-colon (`;`) character after the last parameter definition and before the `SELECT` keyword.

```
PARAMETERS [Enter Country Name] Text(255);  
SELECT *  
FROM [Employees]  
WHERE [Country] = [Enter Country Name]
```

To use a parameter in your query, insert the parameter name anywhere you would use a field name. You can use a parameter value in the field list, `WHERE` clause, `ORDER BY` clause, `GROUP BY` clause, and `HAVING` clause. As with field names and aliases, if your parameter name contains embedded spaces or punctuation, you must enclose the parameter name in square brackets.

When declaring a Text parameter, the SQL language specifies a parameter to indicate the length of the text parameter. In practice, Access ignores this value and in fact, it will replace any values you may include with the value 255 when you save a query from the designer. We generally omit the value altogether, merely including the Text keyword in text parameter declarations.

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

To declare parameters using the Access Query Designer, right-click on the top portion of the Query Designer and choose Parameters from the context menu. Access 2007 users can choose the Parameters item from the Show/Hide group of the Query Design ribbon. Figure 6-12 shows the Display of the Query Parameters dialog box in the Access Query Designer.

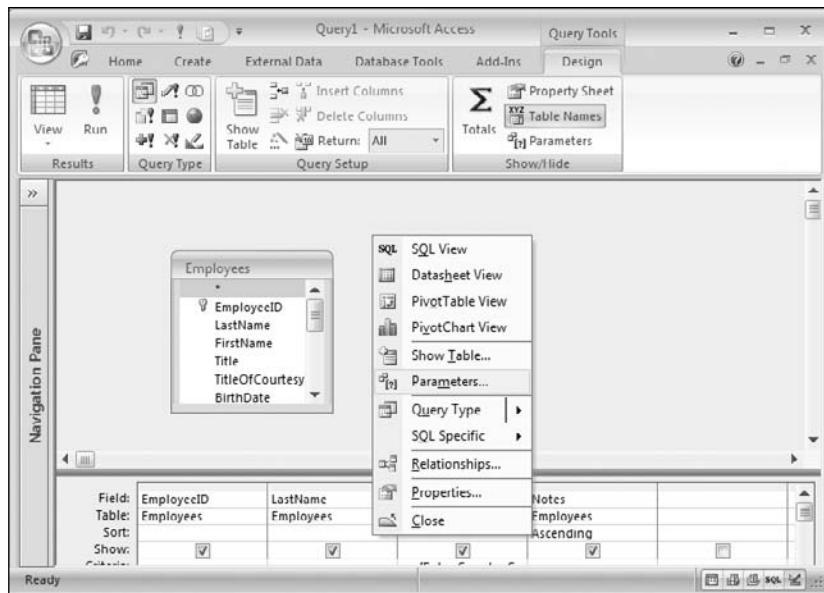


Figure 6-12

In the Query Parameters dialog box (see Figure 6-13), you enter the name of the Parameter, and select the data type from the drop-down list.

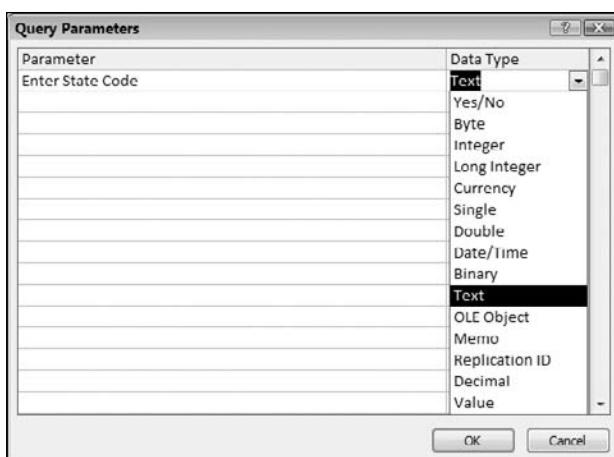


Figure 6-13

Part II: Data Manipulation

Accidental Parameters

The Jet and ACE database engines treat any unresolvable field name as a parameter. If you misspell the name of a field anywhere in your query, or use a field alias outside of the field list, or forget to use a table alias when you've defined one, the query engine will not be able to resolve the reference to an object in your database. When this happens, Access will display a parameter value prompt for the unrecognized token.

If you have declared a parameter, you will receive a prompt even if you don't use the parameter anywhere in your query. When trying to track down an unexpected parameter value prompt, be sure to check for a parameter declaration in your query.

If you receive unexpected Enter Parameter Value dialog boxes when you run your queries, check for misspelled database object names. Check for unused parameter declarations.

Add Data from Other Databases Using the IN Clause

In addition to tables in a Jet or ACE database that contains the query, you can reference tables in other Jet or ACE databases, as well as tables from other database programs that have a Jet ISAM registered on your system.

ISAMs (Installable Sequential Access Method drivers) allow the Jet and ACE database engines to work with data stored in other database formats. ISAMs are available for the following formats:

- Jet/ACE database files
- Microsoft Excel spreadsheet files
- Paradox/Borland Database Engine files
- Comma-separated and tab-delimited text files
- HTML files
- SharePoint list

When you need to reference a Jet or ACE table in another .mdb or .accdb file, you use an `IN` clause (not to be confused with the `IN` comparison operator we discussed earlier — that is used to match against items in a list).

```
SELECT *
FROM [Customers] IN anotherjetdatabase.mdb
```

You can also use UNC references to refer to databases stored on a network:

```
SELECT [LastName]
FROM [Customers] IN \\server\\share\\anotherjetdatabase.mdb
```

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Finally, tables in the local database can be joined to tables in a foreign database using a table join specification with an IN clause:

```
SELECT [ContactName], [OrderID], [OrderDate]
FROM [Customers] INNER JOIN [Orders] IN \\server\share\anotherjetdatabase.accdb
ON [Customers].[CustomerID] = [Orders].[CustomerID]
```

A complex connection string that includes one or more options can be specified using the IN clause:

```
SELECT *
FROM [ListName] IN
"WSS;HDR=NO;IMEX=2;ACCDB=YES;DATABASE=server;LIST=listguid;VIEW=;RetrieveIds=Yes;"
```

For additional examples of working with the IN clause, search Access help for “IN clause.”

The ConnectionStrings.com Web site has lots of information on working with connection strings and template strings already built for just about every supported database server and file format. (www.connectionstrings.com).

Selection Predicates

By default, the database engine assumes you want to return all rows that match your criteria. However, there are several predicates that allow you to fine-tune the results set. These predicates run after the rows are restricted by your WHERE clause and sorted by your ORDER BY clause.

ALL Predicate

By default, the database engine assumes you want to return all rows that match your criteria. You can explicitly demand that the engine return all rows by specifying the ALL predicate.

```
SELECT ALL [CustomerID], [CompanyName], [ContactName]
FROM [Customers]
```

In practice, because ALL is the default behavior if no other predicate is specified, the ALL predicate is rarely, if ever, used.

TOP Predicate

The TOP predicate directs the SQL engine to limit the number of rows that are returned. There are two forms to a TOP predicate:

TOP *n* — Return the top *n* rows of the recordset. The query’s recordset is generated, and then the first *n* rows are returned. In the following query, regardless of the number of rows that satisfied the HAVING clause, only the first six rows will be returned:

```
SELECT TOP 6 [Orders].[CustomerID], Count([Orders].[OrderID]) AS [Order Count]
FROM [Orders]
GROUP BY [Orders].[CustomerID]
```

Part II: Data Manipulation

```
HAVING Count([Orders].[OrderID]) > 5  
ORDER BY Count([Orders].[OrderID]) DESC
```

The following recordset is returned when this query is executed.

Customer	Order Count
Save-a-lot Markets	31
Ernst Handel	30
QUICK-Stop	28
Folk och f� HB	19
Hungry Owl All-Night Grocers	19
Rattlesnake Canyon Grocery	18
Berglunds snabbk�p	18
HILARIO�-Abastos	18

Close examination of this recordset shows eight rows returned, and yet the query asked for only six rows. This is explained by looking at the data in the OrderCount column. The sixth row of the recordset has an Order Count of 18, and there are three customers who had 18 orders. When there is a tie, all records in the tie are returned, even if this exceeds the Top count.

TOP *n* PERCENT — Returns the top *n* percent of the recordset. The query's recordset is generated, and then the first *n* percent of the rows are returned. In the following query, if there were 100 rows that satisfied the HAVING clause, the first five records (5 percent) are returned.

```
SELECT TOP 5 PERCENT [CustomerID], Count([OrderID]) AS [Order Count]  
FROM [Orders]  
GROUP BY [CustomerID]  
HAVING Count([OrderID]) > 5  
ORDER BY Count([OrderID]) DESC
```

The following recordset is returned.

Customer	Order Count
Save-a-lot Markets	31
Ernst Handel	30
QUICK-Stop	28

Customer	Order Count
Folk och fä HB	19
Hungry Owl All-Night Grocers	19

These values may be specified in the Access UI by setting the value of the Top Values property in the query property sheet. The property sheet provides several suggested values that you can choose from, or you can enter any arbitrary value in the property box.

DISTINCT Predicate

The DISTINCT predicate instructs the query engine to remove duplicate rows of data in the table into a single row in the resultset.

When DISTINCT is specified, duplicate rows in the resultset are collapsed into one. DISTINCT looks only at the fields in the resultset to determine what to collapse. If the recordset has any rows that duplicate data across all the fields in the resultset, adding the DISTINCT predicate will remove the duplicate rows.

The following query displays the Country that each order was shipped to:

```
SELECT [ShipCountry]
FROM [Orders]
```

There is no DISTINCT predicate in this query, and so it returns 831 records, one row for each order in the Orders table:

```
SELECT DISTINCT [ShipCountry]
FROM [Orders]
```

When we add the DISTINCT predicate, all of the duplicated rows are removed, resulting in 22 rows.

If we add an additional column to the query, DISTINCT will remove rows that have the same content in each field in the recordset.

```
SELECT Orders.[ShipCountry], Orders.[ShipRegion]
FROM Orders
```

When this query is run, 36 rows are returned. Multiple rows exist for several countries (Brazil, Canada, UK, US, and Venezuela), but there are no duplicates among the combination of the ShipCountry and ShipRegion fields.

You can often achieve the the same results using a GROUP BY clause. However, DISTINCT does not require any aggregation of data, and so is more performant if you don't need aggregation in your results. If you consider our example, DISTINCT is the best choice if all we want is a list of unique Countries that our orders were shipped to. If, however, we need any aggregation (how many orders went to a country, what is the average order amount, what is the largest/smallest order amount per country, and so on), then GROUP BY is required.

Part II: Data Manipulation

The DISTINCT predicate is reflected in the Access Query Designer UI by the Unique Values property of a query. If the Unique Values property is set to Yes in the property sheet, the DISTINCT predicate is included in the SQL that is generated.

Aggregating Data

So far, all of the queries we have discussed return detail data for each and every row included in the criteria. SQL also allows you to group your data and report values against groups of data. The process of grouping your data is called aggregating data.

Aggregating data involves two steps: separating your data into groups, and then calculating statistics on the groups. You may also want to filter your results to include or exclude rows based on the calculations you perform. SQL makes these tasks quite simple.

Bucket Your Data: The GROUP BY Clause

SQL uses the GROUP BY clause to define how you group your data. When a field appears in a GROUP BY clause, the rows that share similar value in the grouped field are collapsed into one row in the resultset. An example will make this clear.

```
SELECT [EmployeeID]
FROM [Orders]
GROUP BY [EmployeeID]
```

The result of this query is nine rows that list the unique names of each employee who placed at least one order. This query looks at all of the rows in the Orders table and groups them based on the values in the EmployeeID field. Every order that was placed by Nancy Davolio is collapsed into one row in the resultset, just as every order placed by Andrew Fuller is grouped into one row. Once these groups are created, you can analyze the groups and report a variety of statistics over the groups.

Frequently, when you want to show a list of unique values in a set of data, a simple GROUP BY clause like this will serve your needs. You can group on more than one field, by simply adding additional fields to the GROUP BY clause. You can also group on the result of expressions.

```
SELECT [EmployeeID],
       Month([OrderDate]) As [Order Month]
  FROM [Orders]
GROUP BY [EmployeeID], Month([OrderDate])
```

This query passes the OrderDate field to the Month VBA function, which returns an integer from 1 to 12 representing the month the order was placed. Then, each row is grouped by both EmployeeID and the Month value. The result of this query is a list of 108 rows showing the unique combination of EmployeeID and the Month value.

Aggregate Functions

When working with grouped data, you will usually want to report on some statistics on the grouped values. The previous example would be much more interesting if we reported how many orders were

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

placed by each employee each month, or the average amount of the orders in each month. We can do this easily using one or more SQL aggregate functions.

SQL aggregate functions provide a way to report statistical information on data in your query. They work on grouped data within your query, but you can also use them in queries without a `GROUP BY` clause. For example, the following query reports the grand total of the freight charges for every order in the `Orders` table:

```
SELECT Sum([Freight]) FROM [Orders]
```

The following query uses SQL aggregate functions to report a variety of statistics on grouped data in the report:

```
SELECT [EmployeeID],  
       Month([OrderDate]) AS [Order Month],  
       Count([OrderID]) AS [Count of Orders],  
       Min([Freight]) AS [Cheapest],  
       Avg([Freight]) AS [Average],  
       Max([Freight]) AS [Most Expensive]  
  FROM [Orders]  
 GROUP BY [EmployeeID], Month([OrderDate])
```

Here we have added the `Count` aggregate function into the field list, which returns the number of items that exist within a grouping. In this case, we are counting the number of items that exist in the `OrderID` field in each grouping. We chose the `OrderID` field because it is the primary key field for this table, and is therefore guaranteed to contain a value for each row.

The `Min` and `Max` functions return the smallest and largest freight charge within the group. Finally, the `Avg` function calculates the arithmetic mean (or as I like to call it, the average) for each freight charge.

The following table lists the most commonly used SQL aggregate functions.

SQL Aggregate Function	Description
<code>Count(fieldname)</code>	Returns the number of items in a grouping.
<code>Avg(fieldname)</code>	Computes the average of values in a grouping.
<code>Sum(fieldname)</code>	Computes the sum of values in a grouping.
<code>First(fieldname)</code> <code>Last(fieldname)</code>	Returns the first or last item in a grouping. Unless you specify an <code>ORDER BY</code> clause, these values will be random.
<code>Min(fieldname)</code>	
<code>Max(fieldname)</code>	Returns the smallest or largest value in a grouping.

The `First` and `Last` functions can be tricky. Unless you have an `ORDER BY` clause in your query, the data in the groupings is not guaranteed to exist in any particular order, and so the value returned by

Part II: Data Manipulation

these functions cannot be predicted. The following query demonstrates how the `First` value is not always the lowest value, and the `Last` value is not always the highest value:

```
SELECT [CategoryID],  
       First([ProductName]) AS FirstOfProductName,  
       Min([ProductName]) AS MinOfProductName,  
       Last([ProductName]) AS LastOfProductName,  
       Max([ProductName]) AS MaxOfProductName  
  FROM [Products]  
 GROUP BY [CategoryID]
```

Category	FirstOfProduct-Name	MinOfProduct-Name	LastOfProduct-Name	MaxOfProduct-Name
Beverages	Chai	Chai	Lakkalikööri	Steeleye Stout
Condiments	Aniseed Syrup	Aniseed Syrup	Original Frankfurter grüne Soße	Vegie-spread
Confections	Pavlova	Chocolade	Scottish Longbreads	Zaanse koeken
Dairy Products	Queso Cabrales	Camembert Pierrot	Mozzarella di Giovanni	Raclette Courdavault
Grains/Cereals	Gustaf's Knäckebröd	Filo Mix	Wimmers gute Semmelknödel	Wimmers gute Semmelknödel
Meat/Poultry	Mishi Kobe Niku	Alice Mutton	Pâté	Tourtière
Produce	Uncle Bob's Organic Dried Pears	Longlife Tofu	Longlife Tofu	Uncle Bob's Organic Dried Pears
Seafood	Ikura	Boston Crab Meat	Röd Kaviar	Spegesild

Functions also exist for calculating Standard Deviations and Variations. For a complete list of the SQL Aggregate functions and detailed documentation, search Access help on “SQL Aggregate Functions.”

When working with grouped data, there are some restrictions on what can appear in the field list. If you have a `GROUP BY` clause in your query, the field list can contain any field or expression that exists in the `GROUP BY` clause, as well as constant values. The field list may also include other fields or expressions that are not in the `GROUP BY` clause, but only when passed as arguments to one of the SQL Aggregate functions. You may have additional fields in the `GROUP BY` clause that do not appear in the field list. In this case, your data will be grouped on each field in the `GROUP BY` clause even though the grouped result is not displayed in the recordset.

Filter Your Data Based on Bucketed Data: the HAVING Clause

If you have grouped data in your query, you can restrict the rows that are returned based on the grouped data, but not in the way you might expect. When filtering on grouped data, you do not use the WHERE clause. The HAVING clause allows you to filter on calculated aggregated values.

The HAVING clause is created just like a WHERE clause, except that any field or expression you put in the HAVING clause must appear in the GROUP BY clause.

In the following example, we add a HAVING clause to our example, to show only those orders placed by employees 1, 2, or 3 between April and June with a total freight charge of greater than \$60. This takes our 108 rows down to nine. Both the field and the expressions in the HAVING clause also appear in the GROUP BY clause.

```
SELECT [EmployeeID],  
    Month([OrderDate]) AS [Order Month],  
    Count( [OrderID]) AS [Count of Orders],  
    Min([Freight]) As [Cheapest],  
    Avg([Freight]) AS [Average],  
    Max([Freight]) AS [Most Expensive]  
FROM [Orders]  
GROUP BY [EmployeeID], Month([OrderDate])  
HAVING [EmployeeID] < 4 AND  
    Month([OrderDate]) Between 4 And 6 AND  
    Avg([Freight]) > 60
```

Comparing the WHERE Clause and the HAVING Clause

It is very common to create a query that has both a WHERE clause and a HAVING clause. The distinction between the two is subtle but critical to generating correct results from your query.

When you have a WHERE clause in your query, the WHERE criteria is applied *before* your data is grouped, and the HAVING clause is applied *after* the data has been grouped. Separating data into groups is an expensive operation. You can improve the performance of your grouped queries — quite dramatically in the case of large data sets — by adding a WHERE clause to filter out unnecessary data before we perform the grouping.

Consider the previous example. Because there is no WHERE clause in the query, we will group every row in the Orders table. In a production system, this table will likely be enormous. If we moved the EmployeeID restriction out of the HAVING clause and into the WHERE clause, we will be grouping only the orders placed by employees 1, 2, and 3. Because the EmployeeID restriction does not require aggregation in order to be interpreted correctly, it can be safely moved to the WHERE clause. Likewise, the date restriction (only orders between April and June) does not require aggregation, and so it can be moved as well.

The following query returns the exact same recordset as the previous example, but in a much more performant way:

```
SELECT [EmployeeID],  
    Month([OrderDate]) AS [Order Month],  
    Count( [OrderID]) AS [Count of Orders],  
    Min([Freight]) As [Cheapest],
```

Part II: Data Manipulation

```
Avg([Freight]) AS [Average],  
Max([Freight]) AS [Most Expensive]  
FROM [Orders]  
WHERE ([EmployeeID] < 4) AND (Month([OrderDate]) Between 4 And 6)  
GROUP BY [EmployeeID], Month([OrderDate])  
HAVING Avg([Freight]) > 60
```

If your restriction does not require aggregated data, place it in the WHERE clause instead of the HAVING clause.

Domain Aggregate Functions

VBA provides several functions to aggregate data in code without the need to create a standalone query. Each of SQL aggregate functions that we have previously discussed has a corresponding domain aggregate function defined by VBA.

Domain Aggregate Function	Description
DCount(<i>what, recordsource, criteria</i>)	Returns the number of items
DAvg(<i>what, recordsource, criteria</i>)	Computes the average of values
DSum(<i>what, recordsource, criteria</i>)	Computes the sum of values
DFirst(<i>what, recordsource, criteria</i>) DLast (<i>what, recordsource, criteria</i>)	Returns the first or last item
DMin(<i>what, recordsource, criteria</i>)	
DMax(<i>what, recordsource, criteria</i>)	Returns the smallest or largest value

Domain aggregate functions differ in that they take three string arguments. In addition to the fieldname or expression that you want to generate the statistics against, you must also specify the source of the data, and optionally, a WHERE clause that filters the data you want to report on. For example, the following expression reports the number of orders (88) placed between April and June, 1998:

```
DCount("[OrderID] ", "[Orders]", "[OrderDate] BETWEEN #4/1/1998# AND #6/30/1998#")
```

Action Queries

Up to this point, we have discussed only select queries that return information, but do not alter the data. You can also use SQL to add, change, and delete records in your database. Queries that allow you to alter data are collectively known as *action queries*. In this section, we look at each of the types of action queries you can create.

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Action queries can be dangerous! Unlike SELECT queries, action queries result in changes to the data in your database, and it is very easy to accidentally update or delete the wrong data.

* *ALWAYS ensure you have backup copies of your database before proceeding with update queries.*

* *ALWAYS ensure your action queries are working on the correct data by running the query as a select query first.*

Access 2007 introduced the concept of enabling advanced features. Action queries are blocked in a database that is not enabled. By default, when you open a database in Access 2007, the database is not enabled, and so action queries cannot be run. Databases must be enabled in order to run an action query. If action queries are not working for you, look for a message in the security bar. Click the Options button and choose to enable content.

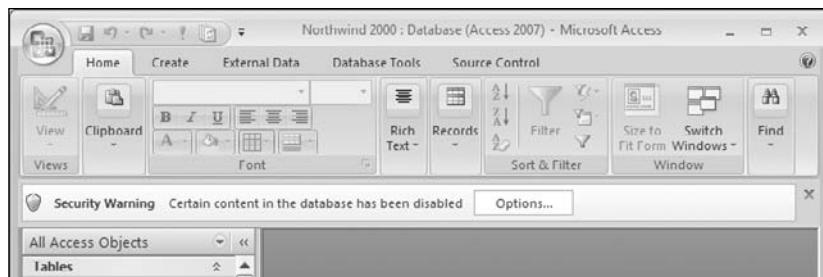


Figure 6-14

There are several different types of action queries. SQL statements exist to create queries that:

- Create a new table based on data in existing tables (referred to in Access as Make Table queries)
- Add data to existing tables (Append queries)
- Update data in existing tables (Update queries)
- Delete data in existing tables (Delete queries)

We will look at each of these types of queries, beginning with the Make Table query.

Make Table Query

Make Table queries allow you to copy a set of rows from one table or query into a brand new table. These queries are SELECT queries with an INTO keyword that indicates the name of the destination table. Any SELECT query (except queries with multi-valued lookup fields) can be converted to a Make Table query.

This type of query is potentially destructive; if the destination table name you specify already exists, Access will offer to delete the existing table and replace it with new data. If you execute a Make Table query from code or from a macro with warnings disabled, it will overwrite the existing data without warning.

Part II: Data Manipulation

The following Make Table query will copy selected information from the `Orders` table into a new table named `NancysOrders`.

```
SELECT [OrderID],  
       [CustomerID] AS [Customer Number],  
       [OrderDate]  
INTO [NancysOrders]  
FROM [Orders]  
WHERE [EmployeeID] = 1
```

Each field in the field list will become a field in the new table. The new field will have the datatype of the field in the source table, and will be created with the name of the source field, or the field alias if one is defined. The new field will not inherit any custom property values from the source field.

The new table, named `NancysOrders`, will contain three fields. The first, `OrderID`, will be created as a long integer because `OrderID` in the `Orders` table is a long integer. The second field will be named `Customer Number` because the query defined a field alias for the `CustomerID` field. The field will be defined as a long integer like the source field, but it will not show the Customer Name by default because the lookup properties will not be carried over to the new table. Finally, the `OrderDate` field will be created as a `DateTime` field.

To test the query before you actually create the table, remove the `INTO` statement and run the select query to verify you are selecting the correct data. In the Access Query Designer, you can preview the data that will be copied into the new table by choosing the Datasheet View button on the Views group on the Home ribbon.

Append Query

Before we get started talking about Append queries, we want to reiterate the warning we made at the beginning of this chapter. Append queries will add new records to your database. Inevitably, while learning to create action queries, you'll try something out that does something unexpected, and that can result (in the case of an Append query) in hundreds, if not thousands, of new rows appearing in your tables. We have no doubt we've inadvertently appended millions of unanticipated, unwanted, and very undesirable rows throughout the years!

Please let our experiences encourage you to make backup copies of your database before you start playing with action queries. And until you have a finalized working query design, always run the query as a select query first to ensure you are selecting the correct rows and updating the correct fields.

Okay, now that you are safely backed up, let's look at the *Append query* (also known as an *insert-into query*). When you have data in one table that you would like to add to another existing table, or you have rows in a table that you want to duplicate within the same table, an Append query is your solution. This query allows you to select rows in one or more tables, and add the data in those rows into an existing table (if you need to create a new table, you can use a Make Table query).

You may want to do this if you have data that you wish to use as a template. For example, every month, we run a test pass that requires a result to be manually logged from every manual test case we run. We can use an Append query to add a row to our `TestPassResults` table for every manual test case we have. Then, when we run our test pass, we can query for all those results records, and use a form to indicate whether they passed or failed. We know each case we need to run, we can see which cases have not yet been run, and we can generate a report of our results.

Append Multiple Records Using a **SELECT** Query as the Data Source

To create an Append query in SQL, you use an `INSERT INTO` statement. This statement consists of the keywords `INSERT INTO`, followed by the name of the destination table or query that will receive the records, followed by a list of fields in the destination table that will receive the data. After the `INSERT INTO` statement, specify a standard `SELECT` query, which retrieves the rows you want to add to the destination table. The `SELECT` query is a standard select query and may include all of the elements in a `SELECT` query, including a `WHERE` clause, `ORDER BY` clause, `GROUP BY` clause, and/or `HAVING` clause.

```
INSERT INTO [DestinationTableName]
    (destField1, destField2, ..., destFieldn)
SELECT srcField1, srcField2, ..., srcFieldn
FROM [sourceTableName] ...
```

You can add records into any updatable table or query. It is important that the number fields in the destination field list matches the number of columns in the source field list, and that the datatypes in each table are compatible. The column names do not need to match (although in many cases they will). What is important is the ordinal position of the columns, and that the datatypes of the columns are compatible. If the data in the source query is sorted using an `ORDER BY` clause, the records will be created in the destination table in the order defined by the `ORDER BY` clause. If the number of fields in the destination field list and the source field list do not match, you will receive a runtime error when you run the query.

For each row in the source recordset, a new row will be created in the destination table. The data from the first column will be inserted into the first column of the destination recordset, repeating until all the columns have been copied.

Let's look at an example, using the `TestResults` scenario we described earlier:

```
INSERT INTO [TestResults]
    ([TestCaseID], [TestPassName], [RunDate])
SELECT [TestCase].[TestCaseID], "August 2008 Test Pass", #8/22/2008# AS [RunDate]
WHERE [TestCase].[Enabled] = True
ORDER BY [TestCase].[Category], [TestCase].[TestCaseID]
```

The `SELECT` query returns the `TestCaseID` and two constant expressions for each test case that is enabled. The results will be sorted first by category, then by the `TestCaseID`. The `INSERT INTO` statement instructs the query processor to add one row into the `TestResults` table for each row returned by the `SELECT` query. The data in the `TestCaseID` field in the source table will be copied into the `TestCaseID` field of the new row in the `TestResults` table. The constant expression "August 2008 Test Pass" will be copied into the `TestPassName` field of the new row, and finally, the date value for August 22, 2008 will be copied into the `RunDate` field of the new row. Because the source table was sorted, the new records will be added to the results table in the sorted order.

Creating a Multi-Record APPEND Query Using the Access Query Designer

To create an Append query in the Access Query Designer, first build the `SELECT` query as you normally would. Run the select query to verify that you will be selecting the correct set of rows to add to your destination table.

Once you have run the `SELECT` query and verified that you are selecting the correct rows, select the Query Design ribbon and choose the Append button from the Query Type group. When you choose this button,

Part II: Data Manipulation

an Append dialog box is displayed, which allows you to choose the name of the destination table. You may optionally choose a table in another database from this dialog box (which will add an `IN` clause following the destination field list). The Append dialog box is shown in Figure 6-15.



Figure 6-15

Once you have specified the destination table and closed the Append dialog box, you will notice a new row appears in the query grid, named Append To. Each cell of this row contains a drop-down list that allows you to specify which field in the destination table will receive the data for this column of the results list. For each column of the `SELECT` query that you wish to copy to the new row in the destination, choose a target field in the Append To row. Choosing destination fields in the Query Designer is shown in Figure 6-16.

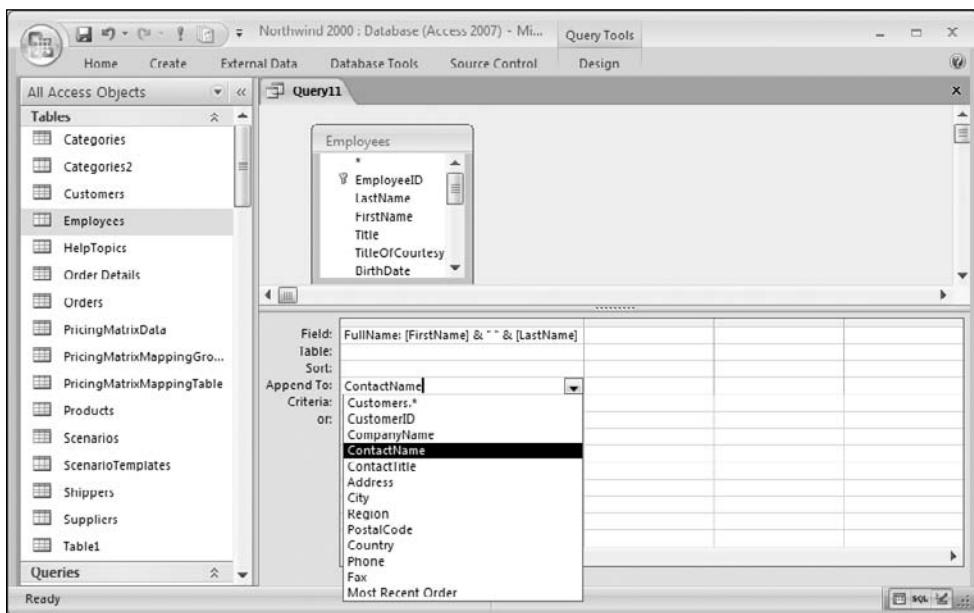


Figure 6-16

As a final sanity check, before running the APPEND query, choose the Datasheet View button in the Views group of the Home ribbon. This will show you the results of the `SELECT` query that will be used to populate the new rows.

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Once you are satisfied that you are selecting the correct rows, switch back to design view, and then choose the Run button in the Views group of the Home ribbon. By default, you will see a warning message indicating how many rows you will be appending to the destination table (you can disable the warning messages by going to the Access Options dialog box, choosing the Advanced tab, and clearing the Confirm Action Queries check box). When you click Yes on this dialog box, the records will be added to the destination table. This action is NOT reversible; have you made your backups yet?!

Append a Single Record Using a Value List as the Data Source

When you want to add a single record to a table, you can create a SELECT query that returns only one record, but it is frequently easier and faster to use a VALUES list. A VALUES list replaces the SELECT query of the previous example.

```
INSERT INTO [DestinationTableOrQueryName]
    (destField1, destField2, ..., destFieldn)
VALUES (value1, value2, ..., valuen)
```

This technique is particularly useful when you need to append constant values or the results of an expression to a table. For example, you may have a log table in your application, and you may want to add a new log entry to the table.

```
INSERT INTO [LogTable]
    ([LogDate], [LogCategory], [LogMessage], [LogResult])
VALUES (Now(), "Category1", "No errors were recorded", "PASS")
```

In this query, a single row will be added to the LogTable table. The current date and time is returned by the Now VBA function and will be copied into the LogDate field of the new row. The constant string value “Category1” will be copied into the LogCategory field. The constant string value “No errors were recorded” will be copied to the LogMessage field, and finally, the constant string value “PASS” will be written to the LogResult field.

Creating a Single-Record Append Query Using the Access Query Designer

Creating a single-record Append query is identical to creating the multiple-record Append query with one exception. Because you don’t want to select records from an existing table, don’t put a table in the Query Designer! Simply add your expressions in the field name row of the grid. Run the query as a SELECT query to ensure you have entered the expressions correctly, and then choose the Append button in the ribbon just like before. It feels weird the first several times you build a query without a backing table, but it works well! Try it out.

UPDATE Query

Pardon me for nagging, but have you made a backup of your data?! One of our examples will show you how an update query can go disastrously wrong. Don’t risk your data! Do as the professionals say and keep yourself backed up!

Now that you are protected against disaster, we’ll look at UPDATE queries. These queries allow you to identify a set of records that exist in a table, and change values in one or more fields. The new values can be set to constants (change the sales tax rate from 8.2 to 8.5 percent), the results of expressions (run all ContactName fields through the StrConv VBA function to ensure each value is proper cased), or the values stored in a field in another table or query.

Part II: Data Manipulation

By their very nature, update queries are destructive. The values currently stored in the fields to be updated will be replaced with the new values you define in the query. You cannot undo the results of an update query, so once you have confirmed you want to replace the values, they are gone.

To create an update query in SQL, use an UPDATE statement. The basic form of an update query looks like this:

```
UPDATE [TableOrQueryToUpdate]
SET [FieldToUpdate] = newvalue
WHERE whereclause
```

The `newvalue` you set can be a constant, the results of an expression, or values stored in a field. The `SET` clause accepts multiple field update expressions. If you need to update more than one field in the update table, you can add additional fields, separated by commas.

By way of example, you may have noticed the order information in the Northwind database is pretty old. We can use an UPDATE query to freshen the data by changing the order dates to something a little more current. We will add 11 years to each date, and remove the `Shipped Date` field so that we can work on fulfilling some orders:

```
UPDATE [Orders]
SET [OrderDate] = DateAdd("yyyy", 11, [OrderDate]),
    [RequiredDate] = DateAdd("d", 14, DateAdd("yyyy", 11, [OrderDate])),
    [ShippedDate] = Null
WHERE [OrderID] < 10300
```

This query uses the UPDATE statement to tell the query processor that we will be changing data in the `Orders` table. The WHERE clause says that only those orders with an `OrderID` of less than 10300 will be updated.

The SET clause contains three field update expressions, separated by commas. The first expression uses the `DateAdd` VBA function to add 11 years to whatever value is in the `OrderDate` field. The second expression sets the `RequiredDate` to the *new OrderDate + 14 days*, using a nested `DateAdd` function. Finally, the third expression sets the value of the `ShippedDate` field to `Null`.

Creating an Update Query Using the Access Query Designer

The first step to creating an update query in the designer is to create the SELECT query that will locate the rows that you want to change. Be sure to add a column for every field you want to update. You can add additional columns if you need them to verify that you are selecting the correct rows. If you will be updating a field with the results of an expression, you should add each of the expressions as an additional column in the query so that you can verify that your expressions are returning the expected results.

Run the SELECT query to verify everything is as expected. Then, click the Update button on the Query Design ribbon. When you click this button, you will notice that the grid changes: the Sort and Show rows are removed, and an Update To row is added. An update query will update any column that has a value in the Update To row. If there is no value in the Update To row for a field, that field will not be updated.

For each field you wish to update, enter the new value in the Update To row. If the value is the result of an expression that you created as a column, simply copy the expression from the Field row to the appropriate Update To row (be sure *not* to include the field alias, just the expression).

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

Frequently, you will want to update one field with values from another field. For example, you may wish to locate all orders that do not have a required date entered, and set the required date to the order date + 5 days. The following `SELECT` query will locate all orders without a required date and displays the order date and the expression that calculates the new required date.

```
SELECT [RequiredDate],  
       [OrderID],  
       [OrderDate],  
       DateAdd("d", 5, [OrderDate])  
  FROM [Orders]  
 WHERE [RequiredDate] IS Null
```

Run this query to verify you have selected the correct records (if you run this against the `Orders` table in Northwind, manually delete the Required Date value from several rows so that you can see this query in operation). Once you are satisfied that the correct rows were selected (only those rows that don't have a required date) and that the expression is correct, take the query back to design view. Choose the Update button on the Query Design ribbon; notice that a new row is added to the grid. Copy the `DateAdd` expression (without the field alias!) to the Update To row under the `Required Date` field. Because this is the only column with a value in the Update To field, it is the only field that will be updated.

Use care when you add data to the Update To row. It is very easy to enter data incorrectly with potentially disastrous results! If you need to reference data in another field in the Update To row, you *must* enclose the field name in square brackets, even if the field doesn't contain embedded spaces or punctuation. Any text you enter into this row is treated as a string. Let's look at how this can create an unintended mess.

Let's say, for some odd reason, you want the company name stored in uppercase. To do this, create an update query against the `Orders` table. Pull the `CompanyName` field into the grid, and then click into the Update To field and enter the following:

```
UCase$(CompanyName)
```

Click out of the cell. Even though `CompanyName` is a field in the `Orders` table, Access put quotation marks around the name! If you run this query, Access will take the string `CompanyName`, convert it to uppercase, and replace every company name value with the literal string `COMPANYNAME`. Not at all what you intended! If you add square brackets around the field name, you will end up with the results you expected.

DELETE Query

As their name implies, `DELETE` queries delete data. Entire rows of data! Unrecoverable data! Make a backup, and then let's have fun deleting data.

There are many scenarios in which you might want to remove rows from a table. You may have backed up your data to an archive database, and now want to clean out the production database. Perhaps you ran an append query and accidentally added a bunch of data. When you need to remove rows from a table, you can use a `DELETE` query. With a `DELETE` query, you can remove just one row, a set of rows that meet your criteria, or every row in your table. The basic form of a `DELETE` query looks like this:

```
DELETE FROM [TableOrQueryToDeleteFrom]  
WHERE whereclause
```

Part II: Data Manipulation

Use the `DELETE FROM` statement to indicate which table or query you want to remove data from. You add a `WHERE` clause to select which rows you want to delete. You can omit the `WHERE` clause to delete every row in the table. The following query will remove every row from the `Orders` table.

```
DELETE FROM [Orders]
```

This query will remove every order from the `Orders` table where the order was shipped by Speedy Express.

```
DELETE FROM [Orders]
WHERE [ShipVia] = 1
```

If you need to join additional tables to your query in order to identify the rows you want to delete, you use a slightly different syntax.

```
DELETE [Orders].*
FROM [Customers] INNER JOIN [Orders]
    ON [Customers].[CustomerID] = [Orders].[CustomerID]
WHERE [Customers].[Country] In ("USA", "Canada", "France")
```

In this query, we specify that we want to delete rows from the `Orders` table using the `[Orders].*` expression.

`DELETE` queries are incredibly easy to create. Making backups of your data is critical when you are in the process of developing your `DELETE` queries. Like all action queries, you cannot undo the results of a `DELETE` query.

Crosstab Queries

Crosstab queries (also referred to as *pivot queries* or *transform queries*) allow you to display summary data in two dimensions.

When you create a crosstab query, you need to choose one or more fields that appear at the beginning of each row, and exactly one field that will appear in each column. Then you can choose a field to summarize that represents the intersection of the row and column values. For example, you may want to analyze our shipping patterns to see whether certain employees are favoring one shipping company over another. You can use a crosstab query to show this data visually.

A crosstab query is a simple `SELECT` query that displays each column you want to see as row headings, with a `GROUP BY` clause set on each of the fields you choose for your row headings. Then, a `TRANSFORM` statement is added before the `SELECT` statement, which creates an aggregation representing the value you want displayed at the intersection of each row and column. Finally, a `PIVOT` statement is added after the `SELECT` query with the field you want to display in each column.

```
TRANSFORM Count([OrderID])
SELECT [EmployeeID],
    Count([OrderID]) AS [Order Count]
FROM Orders
GROUP BY [EmployeeID]
PIVOT [ShipVia]
```

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

This query displays the information we need, but the row and column headings aren't very informative; it would be much nicer to see "Speedy Express" in the column header instead of the ID value 1.

Crosstab Query					
Employee	Order Count	1	2	3	
Davolio, Nancy	123	38	44	41	
Fuller, Andrew	96	35	36	25	
Leverling, Janet	127	36	45	46	
Peacock, Margaret	156	46	70	40	

We can create a more expressive SELECT query that will provide that information, by joining the Orders table to the Shippers table. The common fields are the ShipVia field in the Orders table, which contains the ShipperID that we can match to the ShipperID field in the Shippers table.

```
TRANSFORM Count([OrderID])
SELECT [EmployeeID],
       Count([OrderID]) AS [Order Count]
FROM [Shippers] INNER JOIN [Orders] ON [Shippers].[ShipperID] = [Orders].[ShipVia]
GROUP BY [EmployeeID]
PIVOT [CompanyName]
```

We have added a join between the Orders and Shippers table, and changed the PIVOT statement to display the CompanyName from the Shippers table instead of the ShipperID from the Orders table.

This query can still be improved by bringing in the Employee Name instead of the Employee ID. We can do this by adding one more join to the SELECT query:

```
TRANSFORM Count([OrderID])
SELECT [LastName],
       [FirstName],
       Count([OrderID]) AS [Order Count]
FROM [Shippers] INNER JOIN
      ([Employees] INNER JOIN [Orders]
      ON [Employees].[EmployeeID] = [Orders].[EmployeeID])
      ON [Shippers].[ShipperID] = [Orders].[ShipVia]
GROUP BY [LastName], [FirstName]
PIVOT [CompanyName]
```

We have added a join between the Employees table and the Orders table on the common EmployeeID field. We changed the field list of the SELECT query to show the FirstName and LastName fields from the Employees table instead of the EmployeeID from the Orders table. Finally, we put a group by on the LastName and FirstName fields because both appear in the row heading.

Part II: Data Manipulation

Ordering Column Headings

By default, the columns appear in alphabetical order. This is not so much a problem with names as we have with the shipping companies, but it probably is an issue when you show date values. Let's look at an example, and see how we can fix the problem.

In this example, we are still analyzing shipping information, but this time we want to know how many orders from the first quarter of 1997 were shipped with each carrier, by month. The following crosstab query presents this data in an easy-to-use format.

```
TRANSFORM Count([OrderID])
SELECT [ShipVia]
FROM [Orders]
WHERE ((([OrderDate]) Between #1/1/1997# And #3/31/1997#))
GROUP BY [ShipVia]
PIVOT Format([OrderDate], "mmm")
```

Notice that we added a WHERE clause to the SELECT query to return only data from the first three months of 1997. Also, we are using the Format VBA function to display the OrderDate field as the month name abbreviation. The table that follows shows the results of this query.

Ship Via	Feb	Jan	Mar
Speedy Express	7	14	5
United Package	15	11	14
Federal Shipping	7	8	11

This looks great, save one problem. The months are out of order! Just like the earlier example where we had company names in the column headers, here the columns appear in alphabetical order.

We can force Access to display columns in an arbitrary order by using an IN clause with the PIVOT statement:

```
TRANSFORM Count([OrderID])
SELECT [ShipVia]
FROM [Orders]
WHERE ((([OrderDate]) Between #1/1/1997# And #3/31/1997#))
GROUP BY [ShipVia]
PIVOT Format([OrderDate], "mmm") IN ("Jan", "Feb", "Mar")
```

If the PIVOT statement contains an IN clause, the crosstab query results will have one column for each item in the IN clause, in the order they appear in the IN clause. In this case, the results will have three columns (in addition to any row header columns) labeled, in order from left to right, Jan, Feb, and Mar.

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

You can add additional column labels to the `IN` clause, even if there is no data to fill the column. For example, if we changed the `PIVOT` statement to the following:

```
PIVOT Format([OrderDate], "mmm") IN ("Jan", "Feb", "Mar", "Apr", "May", "Jun")
```

we will see six columns in the query (in addition to the row header column). However, our query has restricted the rows to those from January through March, 1997, so the `Apr`, `May`, and `Jun` columns will be empty.

Sometimes you will see empty columns in your crosstab query that you expected to be filled with data. This frequently occurs if you make a typographical error in the `IN` clause, or when the data doesn't match the column name. Access looks for an exact match between the column value of the `PIVOT` statement and the column name defined in the `IN` clause. If data exists that does not match a column name, it is discarded. If a column name is defined for which no data exists, the column appears in the results list, but no data appears in the column.

Common Expressions Used with Crosstab Queries

In the previous example, we used the `Format` VBA function to display date information broken out by month. There are several expressions that are commonly used with the `Format` function in crosstab queries to give meaningful column names, as shown in the following table.

Expression	Purpose
<code>Format([TextField], "mmmm")</code>	Group and display date data as a month name.
<code>Format([TextField], "mmm")</code>	Group and display date data as a month name abbreviation.
<code>Format([TextField], "q")</code>	Group and display date data as a quarter number (1 through 4).
<code>Format([TextField], "yyyy")</code>	Group and display date data as a four-digit year value.
<code>Format([TextField], "ddd")</code>	Group and display date data by day name (first three letters of the day name).

For a complete list of the format strings, search Access Help for “Format Function.”

Partition Function

The `Partition` function allows you to divide values in a field into ranges. You can choose a lower bound, an upper bound, and a range unit value. Let's look at an example. We will display the number of products split into price ranges. The following query will display a price range, in \$10 increments, and show how many products fit into each range:

```
SELECT Partition([UnitPrice],0,60,10) AS [Unit Cost],  
       Count([ProductID]) AS Items  
  FROM [Products]  
 GROUP BY Partition([UnitPrice],0,60,10)
```

Part II: Data Manipulation

The following table displays the results of this query.

Unit Cost	Items
0: 9	8
10:19	30
20:29	14
30:39	12
40:49	6
50:59	2
61:	5

This indicates that there are 14 products that cost between \$20 and \$29.99. We often use the output of the partition function to define a column heading field in a crosstab query. The following query displays the number of products in each category, broken out by price range. We have used a \$15 price increment, capped at \$60. Any products that cost more than \$60 are lumped together in one category, labeled “61”:

```
TRANSFORM Count([ProductID]) AS CountOfProductID
SELECT [CategoryID]
FROM [Products]
GROUP BY [CategoryID]
PIVOT Partition([UnitPrice],0,60,15)
```

The following table shows the results of this query.

Category	0:14	15:29	30:44	45:59	61:
Beverages	4	6		1	1
Condiments	2	8	2		
Confections	6	3	2	1	1
Dairy Products	2	2	5	1	
Grains/Cereals	3	2	2		
Meat/Poultry	1	1	2		2
Produce	1	1	1	2	
Seafood	5	5	1		1

UNION Queries

Union queries allow you to combine the results of two or more `SELECT` queries into one. Each `SELECT` query can be completely different from the others. They do not need to pull data from the same tables.

In their simplest form, `UNION` queries are simply two standard `SELECT` queries joined by the `UNION` keywords.

```
SELECT [ContactName]
FROM [Customers]
UNION
SELECT UCASE$([Last Name]) & " " & [First Name]
FROM [Employees]
```

As you can see, this query consists of two standard `SELECT` queries, with the `UNION` keyword joining them.

In a `UNION` query, the first `SELECT` query has a special responsibility. It defines the names of the resulting columns in the query results set, and it controls the data types of the columns. If you add a text column, a date column, and a numeric column to the first `SELECT` query's field list, in that order, then all subsequent `SELECT` queries must also place date/time data into the middle column, and a numeric value into the third column. The subsequent `SELECT`s may use expressions to convert data to the appropriate data type, if necessary.

Each `SELECT` statement must project the same number of columns. This is an absolute requirement for `UNION` queries; you will receive an error if you attempt to run a `UNION` query with mismatched column counts. When a `UNION` query combines the multiple `SELECT` queries, it does so by matching columns based on ordinal position. The data from the first column in the second query gets added to the first column in the resulting recordset, data from the second column goes into the second column, and so on.

The first `SELECT` must create a column for each resulting column in the results set. You can specify an empty string to create a placeholder column if there is no data in the first select for a given column.

Time for an example! During the holidays, we want to send greeting cards to some of our employees, customers, and suppliers. We have mailing address information for all of them, but it is contained in three separate tables. We want to consolidate this information into one table so that we can efficiently print out mailing labels. We have some unique requirements for which people we want to send cards to that differ based on the type of address. We only want to send cards to customers in the USA, France, Norway, Canada, and Germany. Oddly, we don't want to send cards to our suppliers in Germany. Finally, our UK subsidiary has its own greeting card system, so we only want to send cards to our employees based in the United States.

We can use a `UNION` query to do this very easily. Simply create three `SELECT` queries, one each for the three tables, specifying the correct criteria for each table. Make sure that each query contains the same number of columns, and that the data types that go into each column are compatible (don't try to put a string value into a column that is defined as numeric in the first query). In our case, all of our columns in the first `SELECT` are text columns so there is no worry about a data type mismatch.

```
SELECT "Customer" As Type, [ContactName], [CompanyName],
[Address], [City], [Region], [PostalCode], [Country]
FROM [Customers]
```

Part II: Data Manipulation

```
WHERE [Country] IN ("USA", "France", "Norway", "Canada", "Germany")
UNION
SELECT "Employee", [FirstName] & " " & [LastName], Null,
       [Address], [City], [Region], [PostalCode], [Country]
FROM [Employees]
WHERE [Country] = "USA"
UNION
SELECT "Supplier", [ContactName], [CompanyName],
       [Address], [City], [Region], [PostalCode], [Country]
FROM [Suppliers]
WHERE [Country] <> "Germany"
```

The column names for our resulting recordset are defined by the names of the columns in the first SELECT query. We have defined a constant value in the first column (which we aliased in the first SELECT as Type) to identify which type of record each row is.

The schema of the Employees table is slightly different from the Customers and Suppliers table so we need to combine the FirstName and LastName fields into a single field to match the other tables. Also, because there is no CompanyName field in Employees, we fill that column with Null.

Running this query against northwind.mdb results in 70 address records that we can use to generate our mailing labels. Each of our WHERE clauses was respected: There are no suppliers from Germany in our list, and only our employees based in the United States are listed. However, these 70 records appear in order by the type of entry. We'd much prefer to sort them based on the addresses, so we can get discounts on our postage.

Sorting a UNION Query

It's very easy to sort records in a UNION query so long as you remember that you must set the sort order after all of the SELECT queries. However, when you add fields to the ORDER BY clause, you must specify the field names that appear in the first select clause.

Let's add a sort to each of our SELECT queries to demonstrate that only the last ORDER BY clause is respected.

```
SELECT "Customer" As Type, [ContactName], [CompanyName],
       [Address], [City], [Region], [PostalCode], [Country]
FROM [Customers]
WHERE [Country] IN ("USA", "France", "Norway", "Canada", "Germany")
ORDER BY PostalCode
UNION
SELECT "Employee", [FirstName] & " " & [LastName], Null,
       [Address], [City], [Region], [PostalCode], [Country]
FROM [Employees]
WHERE [Country] = "USA"
ORDER BY City, Address
UNION
SELECT "Supplier", [ContactName], [CompanyName],
       [Address], [City], [Region], [PostalCode], [Country]
FROM [Suppliers]
WHERE [Country] <> "Germany"
ORDER BY Country, PostalCode, ContactName
```

Even though each `SELECT` has an `ORDER BY` clause, only the last `ORDER BY` is respected. Our results appear sorted first by country, within each country by postal code, and within each postal code, by the contact name.

Using the Designer to Create Union Queries

The Access Query Designer does not allow users to create union queries directly. However, union queries are really nothing more than several select queries strung together, and we can definitely use the graphical designer to create `SELECT` queries. It's actually quite easy to create a union query in the designer.

First, create a select query in the designer. This query will be the first query in your union query, and so you must make sure this query contains one column to represent each column in your final output. If you expect to need more columns in later queries, add a placeholder column here using the expression "" (an empty string) with an alias that will be the name of the column in the final result. Run this query to verify that your SQL is correct. Finally, switch this query into SQL view, remove the semicolon at the end of the SQL statement, and add a `UNION` keyword that will join this query to the next query you create.

Now, create your second select query in a new designer window. Following the rules of union queries, make sure this query contains the same number of columns as the first query you created, and make sure that the data types are compatible (i.e., don't try to match a string field in this query to a column in the first table that is a numeric data type, unless you first convert the data with a VBA conversion function). Run this query to verify that your SQL is correct, and then switch it into SQL view. Copy the contents of this SQL window to the clipboard, and then switch to the first query and paste the content into that SQL window after the `UNION` keyword.

You can repeat this process until you have all of your separate `SELECT` queries created and verified. It's always a good idea to run the query to verify that the SQL is valid and that it returns the expected results at each step. We also suggest saving the query at each step of the way.

Subqueries

Access makes it very easy to create complex queries by basing one query on the results of another query. This technique allows reasonably novice users to create relatively simple queries, and combine them into much more sophisticated queries. It also can create a maintenance nightmare because a change to one query will cascade into the queries that depend upon it. The Object Dependencies task pane can help, but it doesn't provide a holistic view of your database object dependencies.

You can use subqueries to provide the same capabilities as dependent queries, but because a subquery contains everything it needs to return the final results set in one SQL statement, you have a much simpler maintenance story.

Subqueries can be used anytime you need to create a query based on the results of another query. The simplest subqueries can even be created in the Access Query Designer, although the more complicated subqueries are generally easier to create by authoring the SQL statement directly. You can use a subquery inside `SELECT`, `Make Table`, `UPDATE`, `Append`, `DELETE` and certain Crosstab queries.

Part II: Data Manipulation

IN Predicate

Earlier in this chapter, we looked at how to use the `IN` operator to match against items in a list. You can use a subquery to specify the values that should appear in the list. Instead of supplying a comma-separated list of values, you can specify a `SELECT` query that returns one (and only one) column of data. The `SELECT` query must be enclosed within parentheses and must be immediately preceded by the `IN` predicate.

The following query identifies all orders that were shipped late that had at least one product in the order with more than 50 units.

```
SELECT [OrderID], [ShippedDate], [RequiredDate]
FROM [Orders]
WHERE [OrderID] IN
(
    SELECT [OrderID]
    FROM [Order Details] WHERE [Quantity] > 50
)
AND [ShippedDate] > [RequiredDate]
```

The subquery identifies each Order that has a product with more than 50 items. This query returns one and only one field in the field list. If a subquery returns multiple columns (for example, if we returned the `OrderID` and the `Quantity`), an error is displayed.

The main query uses the subquery in the `WHERE` clause to display only those orders with large quantities. It then further restricts the results to those orders that were shipped late.

This concludes our look at the Data Manipulation Language features of the SQL language. By now, you should have an appreciation of the power and flexibility that SQL provides for working with your data. In the next section, we look at the key features that SQL provides for creating and manipulating schema.

Data Definition Queries

Where *Data Manipulation Language* (DML) allows you to create, query, and report on all aspects of the data in your database, *Data Definition Language* (DDL) provides similar functionality for working with your database schema.

With DDL, you can create new tables with any of the available data types and features (including a few features you can get only through DDL). You can create new indexes and relationships, and views and procedures in ADP solutions.

In ACE and Jet, data definition queries only work with ACE and Jet databases. If you need to create schema objects in other database formats, you must use ADO or DAO. Also, DDL cannot be used to create multi-valued lookup fields.

You cannot batch DDL queries together, as you can in SQL Server or some other database programs. In ACE and Jet, you must create separate queries for each DDL action you need to perform. For example, you might create a `CREATE TABLE` query to create a table, and then separate `CREATE INDEX` queries to create indexes on the table.

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

One beauty of DDL queries is that you can create very efficient code to alter schema. With one string, you can replicate many lines of DAO or ADO code. The string can be passed to the Execute method of a Database object, or to a RunSQL macro command and the query will be run.

Create Table

The CREATE TABLE statement is used to create a new table in an ACE or Jet database, as follows:

```
CREATE TABLE tablename
(
    fieldspecification [, fieldspecification [, ...]]
    [, CONSTRAINT multifieldindex [, ...]]
)
```

You specify the name of the table, followed by a list of field specifications and constraint specifications inside parentheses. The field specification defines the name of the field, the data type and size, and optionally certain special attributes.

```
fieldname datatype[(size)] [NOT NULL | PRIMARY KEY]
```

The datatype is specified using a SQL data type keyword. If you are defining a char field, you may indicate how many characters the field can hold. If you omit the size, by default a char field will be created at the maximum size of 255 characters. The NOT NULL statement indicates that data is required in this field, and is equivalent to setting the Required property on the field to True (or Yes in the property sheet). The PRIMARY KEY statement indicates that this field comprises the PRIMARY KEY for the table.

The following query creates a new table named TestCases with five fields and a primary key:

```
CREATE TABLE TestCases
(
    idTestCase AUTOINCREMENT CONSTRAINT PrimaryKey PRIMARY KEY,
    Description char(255) NOT NULL,
    Owner char(8),
    [Date Created] datetime,
    Active bit
)
```

The first field is defined as AUTOINCREMENT. This creates what is referred to as an AutoNumber field in the Access table designer. It is a long integer with the AutoIncrement attribute set. The PRIMARY KEY statement causes a primary key index (named PrimaryKey) to be created on the field.

The NOT NULL statement is used to require a value in the Description field. This is equivalent to setting the Required property to True (or Yes in the property sheet).

Brackets are used to create a field name with embedded spaces for the fourth field.

Data Types Used

When you create a table using DDL, you must use one of the SQL Data Type names, which in many cases differ from the friendly names displayed in the table designer. The following table lists the most common data types you will use when working with DDL.

Part II: Data Manipulation

SQL Data type	Access designer
char, character	Text
Binary	Object
Bit	Yes/No
Byte	Byte
currency	Currency
datetime	DateTime
uniqueidentifier	ReplicationID
Real	Single
Float	Double
smallint	Integer
Integer	Long Integer
decimal	Decimal
Memo	Memo
autoincrement	AutoNumber Long Integer with dbAutoIncrement attribute set

For a complete description of each of these data types, search Access help for “SQL Data Types.”

The following query re-creates the `Categories` table from Northwind:

```
CREATE TABLE CategoriesNew
(
    CategoryID AUTOINCREMENT CONSTRAINT PrimaryKey PRIMARY KEY,
    CategoryName char(15) NOT NULL CONSTRAINT CategoryName UNIQUE,
    Description memo,
    Picture LONGBINARY
)
```

Working with Primary Keys

You can set a primary key on a table in a couple of different ways. First, you can add the `PRIMARY KEY` statement to any field definition to create a single-field primary key index on that field. The `PRIMARY KEY` statement may be added only to one field in the field list. An error (“Primary Key already exists”) is displayed if the statement appears on more than one field.

```
CustomerID AUTOINCREMENT PRIMARY KEY
```

Chapter 6: Using SQL to Retrieve Data and Manipulate Objects

The index will be created with an auto-generated name based on the field name. If you desire the name of the index with a specific value, you can add a CONSTRAINT clause to the field definition.

```
CustomerID AUTOINCREMENT CONSTRAINT pkIndexName PRIMARY KEY
```

This results in the same index as the first example, except it will use the name you specified (pkIndexName) instead of generating a random name.

Finally, you can create a multi-field primary key index by adding a multi-field CONSTRAINT clause after you have defined all the fields in the table:

```
LastName text(30),  
FirstName text(30),  
DateOfBirth datetime,  
CONSTRAINT [PrimaryKey] PRIMARY KEY (LastName, FirstName, DateOfBirth)
```

If you need to adjust the sort order (ascending or descending), you may create a primary key index using a CREATE INDEX query.

Alter Table

The ALTER TABLE statement allows you to change the design of an existing table. Using this statement, you can add, modify, or remove fields and constraints.

To add a field to an existing field, include an ADD COLUMN statement within your ALTER TABLE query, as follows:

```
ALTER TABLE Customers  
    ADD COLUMN Remarks char(255) NOT NULL,  
        [Most Recent Order] datetime,  
        [Is Active Customer] bit
```

You can add one or more columns. Each column is defined just as it would be in a CREATE TABLE statement.

To remove a field, include a DROP COLUMN statement in your query:

```
ALTER TABLE Customers  
    DROP COLUMN Remarks, [Is Active Customer]
```

You can remove one or more columns in a DROP COLUMN statement. There are no variations to this statement: You merely indicate the name of the table you wish to remove the column from, and specify the names of each column to remove.

While you can add or remove multiple columns in a single ALTER TABLE query, you cannot mix and match adding and removing columns. An ALTER TABLE query may contain an ADD COLUMN or a DROP COLUMN statement but not both. You will need to create two ALTER TABLE queries if you need to both add and remove columns from a table.

You can change the datatype of an existing field using the ALTER COLUMN statement. Specify the name of the field to alter, and the new datatype and size as appropriate.

Part II: Data Manipulation

This action may be destructive! If you make a field smaller, data may be truncated; if you change the datatype to an incompatible datatype, existing data will be deleted.

The following query will change the address field to a text field with a maximum of 30 characters:

```
ALTER TABLE Customers  
    ALTER COLUMN Address text(30)
```

Because Northwind is shipped with an address field of 60 characters, when this query is run, only the first 30 characters of the address data will be preserved.

Drop Table

To remove a table from your database, use the appropriately named `DROP TABLE` statement. As the name implies, this is a destructive command: *Your table will be deleted and it cannot be undeleted.*

```
DROP TABLE tablename
```

It's that simple! Use brackets around the `tablename` if it contains any embedded spaces or punctuation. To delete the `Customers` table, use the following query:

```
DROP TABLE Customers
```

Create Index

When you want to add an index to a table, you use the `CREATE INDEX` statement, as shown in the following code. This statement accepts the name of the index and the table to which it applies, and a list of fields to create the index on:

```
CREATE INDEX [Ship Address] ON Orders  
(  
    ShipCountry,  
    ShipPostalCode DESC,  
    ShipRegion ASC,  
    ShipCity,  
    ShipAddress  
)
```

If you need to adjust the sort direction, use the `ASC` (for ascending sort order: smallest to largest, earliest to latest) or `DESC` (for descending sort order: largest to smallest, latest to earliest) keyword after each field you want adjust. By default, indexes are created in ascending order.

You can create a unique index by adding the `UNIQUE` keyword, as follows:

```
CREATE UNIQUE INDEX [Ship Address Unique] ON Orders  
(  
    OrderID  
)
```

Drop Index

Just as we removed a table using the `DROP TABLE` statement, we can remove indexes from a table using the `DROP INDEX` statement. This statement is just as simple as the `DROP TABLE` statement: merely indicate the name of the index you want to remove, and the name of the table that contains the index.

```
DROP INDEX [Ship Address] ON Orders
```

No data is destroyed when you drop an index. If necessary, you can recreate the index using a `CREATE INDEX` query.

Alter Database Password

Use the `ALTER DATABASE PASSWORD` statement to change the password for a database. This statement takes two string arguments: the new password and the old password. If the database does not currently have a password, specify the keyword `NULL` for the `oldpassword` argument. If you wish to remove a password from a database, specify the keyword `NULL` for the `newpassword` argument.

A password consists of 1 to 20 string characters. Control characters (including the space character) are not allowed in a password and will result in an error:

```
ALTER DATABASE PASSWORD newpassword oldpassword
```

To add the password `Acc3ssPassw0rd` to a database:

```
ALTER DATABASE PASSWORD Acc3ssPassw0rd NULL
```

To change the password to `ChangedPassw0rd`:

```
ALTER DATABASE PASSWORD ChangedPassw0rd Acc3ssPassw0rd
```

To remove the password:

```
ALTER DATABASE PASSWORD NULL ChangedPassw0rd
```

SQL Pass-Through Queries

SQL Pass-Through queries are a special variety of query that are used when working with Microsoft SQL Server or other enterprise database products. These queries are authored using the native syntax of the enterprise database. As the name implies, these queries are passed directly to the database engine, and are executed entirely on the server. The Access query processor does not parse the query and does not attempt to run the query through the ACE or Jet database engine.

SQL Pass-Through queries can be extremely performant because all processing is handled on the server and only the final results are passed across the network to Access. Pass-Through queries also allow you to call stored procedures on the server.

Part II: Data Manipulation

You can create a Pass-Through query in the Access Query Designer. Create a new query, and choose the Pass-Through button in the Query Type group on the Query Design ribbon.

When you create a new SQL Pass-Through query, you must declare a connection string that informs Access which server it needs to pass the query to. Open the property sheet for the query, and enter the connection string into the ODBC Connection Str property (referred to as the Connect property if you are working with a QueryDef in VBA). You can also click the builder button in the property to launch the ODBC Data Source Browser. In the browser, you can locate data sources that are defined on your system, browse to file-based data sources, and create a new data source from scratch.

The ReturnsRecords property provides optimizations for Pass-Through queries that do not return records. If you execute an action query that updates or deletes rows, or execute a stored procedure that does not return records, setting this property to No improves performance.

The LogMessages property determines whether messages returned from the server are persisted. If LogMessages is set to Yes, messages from the server are stored in a table in the database. Setting this property to No results in faster performance, with the obvious side effect that you will not be aware of any messages that the server returns. Search online help for “LogMessages property” for details on the naming scheme used when creating.

Running Ad Hoc SQL Pass-Through Queries

When working with Pass-Through queries, it is not necessary to persist them in your database. You can create QueryDef objects in code, set the properties of the QueryDef, and Execute the query without ever persisting them in the database.

The following procedure accepts a connection string and a SQL statement. It builds a QueryDef object, and sets the Connect property.

```
Public Sub ExecSQLPT(stConnect As String, stSQL As String)
    Dim qd As DAO.QueryDef
    Set qd = CurrentDb.CreateQueryDef(vbNullString)
```

When an empty string is passed to the CreateQueryDef method, the QueryDef is created in memory and is not appended to the QueryDefs collection or saved in the database.

```
qd.Connect = stConnect
qd.ReturnsRecords = False
```

Setting the Connect property informs Access that this is a Pass-Through query. The property takes a fully formed connection string, which identifies the ODBC driver and server connection information. You should set the Connect property before you set any other properties of the QueryDef.

```
' Build the update query string
qd.SQL = stSQL
qd.Execute

Set qd = Nothing
End Sub
```

Store the SQL statement into the SQL property and call the Execute method to run the query.

ANSI Mode

By default, the Microsoft Jet database engine and the Microsoft Access database engine (Ace) use a SQL syntax based on the ANSI-89 standard that is slightly different from the ANSI-92 standard syntax used by many enterprise database products such as Microsoft SQL Server.

You can set an option to use ANSI-92 syntax on a specific database. This setting is stored with the database and will apply only to that database. Whenever you open the database, Access will know to use ANSI-92 syntax instead of the default.

Generally speaking, you will get the best experience in Access by sticking with the default behavior. You should switch a database to use ANSI-92 compatibility mode only if you have a specific need for this capability (for example, you have many linked tables to a SQL Server and are experienced creating queries and procedures in SQL Server).

To switch between standard Jet mode and ANSI mode, follow these steps:

1. From the Office menu, choose Access Options.
2. Choose the Object Designers tab and locate the Query Design section.
3. Check the “This database” box to enable ANSI-92 compatibility mode.

You can also change the default behavior for all new databases you create in Access, by checking the “Default for new databases” check box. Again, unless you have a specific need for ANSI-92 compatibility, we recommend that you do not check this box.

String Pattern Matching Using ANSI-92 Syntax

When you use ANSI-92 syntax, you must use the `ALIKE` (ANSI-Like) operator instead of `LIKE` to perform string matching in query `WHERE` clauses. The Access Query Designer makes this change for you automatically.

You must also use the ANSI wildcards for pattern matching instead of the Access defaults. Use the `%` (percent sign) to match on zero or more characters (equivalent to the `*` asterisk wildcard). Use the `_` (underscore) to match on any single character (instead of the `?` question mark).

Summary

In this chapter, we examined the following:

- How to create select queries to return data from a database
- How to create action queries to add, modify, and remove data from a database
- How to create and work with subqueries
- How to create and work with crosstab and union queries
- How to use Data Definition Language queries to create and modify schema
- The subtleties of working with ANSI-92 compatibility mode

In the next chapter, we will look at techniques you can use to locate and manage data in your database.

7

Managing Data

The primary function of a relational database application, let alone an Access application, is to manage data. Data is our bread and butter, and is probably a big reason behind the work that you're doing. Regardless of the amount of data that you have, it should be easy to work with. Finding it should be not only fast, but easy as well.

The Jet database engine that was used with Access from the beginning is now included with Access 2007 and known as the Microsoft Office Access database engine. This updated version of the database engine includes some pretty interesting new features, such as new multiple-value data types that make very simple many-to-many relationships easier to use, as well as introduce new features such as append-only memo fields and attachments.

In this chapter, we look at a wealth of different techniques for working with your data, including strategies and techniques to:

- Locate specific records that match criteria
- Position and manipulate the cursor in a recordset
- Implement custom sorting solutions
- Work with Append-Only Memo fields
- Work with Attachment fields

Finding Data

Access provides three different techniques for locating data within your database. The `Find` methods enables you to locate records that match one or more criteria you specify. This is analogous to a cursor in the SQL Server/Oracle world because it provides procedural access to data in a recordset. `Find` searches through each record in a recordset in order, and tests each row to see if it meets the search criteria. Data does not have to be indexed in order to use a `Find` method, although `Find` uses an index if one exists and if the search specification is compatible with the index.

Part II: Data Manipulation

The `Seek` method enables you to quickly locate indexed data in a recordset object. `Seek` requires that your data be indexed, and that your search criteria be compatible with the index. `Seek` works against only one index at a time, although it can use multi-field indexes to search on more than one field at a time.

Finally, the `Move` methods allow you to step through a recordset, one row at a time. No attempt is made to compare a row to a search criterion. You can evaluate the values in each row.

Find Methods

The `Find` methods, as shown in the table that follows, are used to locate data in a recordset, regardless of whether the data is indexed or not.

Method	Description
<code>FindFirst</code>	Locate the first occurrence of an item in a recordset.
<code>FindLast</code>	Locate the last occurrence of an item in a recordset.
<code>FindNext</code>	Starting at the current location in a recordset, locate the next occurrence of an item.
<code>FindPrevious</code>	Starting at the current location in a recordset, locate the previous occurrence of an item.

The `Find` methods take one argument, which is a string that defines the search criteria. The string is a `WHERE` clause from a SQL statement, but without the word `WHERE` (as we discussed in Chapter 5). For example, if you want to find the first record in the `Customers` table where the `Region` field contains `OR`, the SQL query might look like this:

```
SELECT * FROM [Customers] WHERE [Region] = "OR"
```

You would use the `FindFirst` method to locate the first matching customer with the following criteria:

```
rs.FindFirst "[Region] = 'OR'"
```

Because this is a SQL clause, we can use single quotes to delineate the string value. Alternatively, we could use escaped double-quote characters, like this:

```
rs.FindFirst "[Region] = ""OR"""
```

When you call the `FindFirst` method, the current record is set to the first row of the recordset, and then the database engine searches through each row in the recordset from the beginning to the end. `FindLast` sets the current record to the last row of the recordset, and the search proceeds backwards from the end to the beginning.

Each row is compared against the criteria; if the criteria are satisfied, the row is set as the current record and control is returned to the calling procedure. When control is returned, the `NoMatch` property is set to `False` if a row was found that satisfied the search criteria; otherwise, the `NoMatch` property is set to `True` and the current record is not changed.

The following function demonstrates the use of the `Find` methods:

```
Public Function FindHarness(tableName As String, _
    criteria As String, Optional bForward _
    as Boolean = True)

    Dim db As DAO.Database
    Dim rs As DAO.Recordset

    Set db = CurrentDb()
    Set rs = db.OpenRecordset(tableName, dbOpenDynaset)
```

The function accepts three arguments: the name of the table to search, the criteria string, and a Boolean variable that indicates whether you want to search in a forward or reverse direction. After declaring our variables, we open a dynaset-type recordset against the `tableName` argument:

```
'// Find the first customer in Washington
If bForward Then
    rs.FindFirst criteria
Else
    rs.FindLast criteria
End If
```

We check the value of the Boolean argument and call either `FindFirst` or `FindLast` on the recordset.

```
If rs.NoMatch Then
    Debug.Print "No match! Current record is now:", rs(0)
Else
    Debug.Print "Match(es) found!"
    Do Until rs.NoMatch
        '// Print information for each matching row
        Debug.Print rs(0).Value

        '// Find the next matching record
        If bForward Then
            rs.FindNext criteria
        Else
            rs.FindPrevious criteria
        End If
    Loop
    Debug.Print "No more matches found! "; _
    "Current record is now:", rs(0)
End If
```

Next, we check the `NoMatch` property to see if a match was found. If no match is found, we print out the first field of the current record. This demonstrates the behavior of resetting the `CurrentRecord` before commencing the search.

If a match is found, we print the value of the first field of the row, and then we call `FindNext` or `FindPrevious` in a loop to locate additional matching rows. Each time we find a match, we print the first field of the row, and continue until no more matches are found.

```
Cleanup:
On Error Resume Next
```

Part II: Data Manipulation

```
rs.Close  
Set rs = Nothing  
Set db = Nothing  
  
Terminate:  
On Error GoTo 0  
  
End Function
```

Our cleanup block closes the recordset and releases the database variables.

When this function is run in the Immediate window against the *Customers* table in the Northwind database, the following output is generated. Because the *bForward* argument was omitted, the argument value defaults to *True*, and the code will use *FindFirst*/*FindNext*, which results in an ascending order.

```
? FindHarness("Customers", "Region = 'OR'")  
Match(es) found!  
GREAL  
HUNGC  
LONEP  
THEBI  
No more matches found! Current record is now: THEBI
```

Adding the *False* flag causes the harness to use *FindLast*/*FindPrevious* instead, resulting in a descending sort order:

```
? FindHarness("Customers", "Region = 'OR'", False)  
Match(es) found!  
THEBI  
LONEP  
HUNGC  
GREAL  
No more matches found! Current record is now: GREAL
```

Seek Method

When you want to locate data that is indexed, the *Seek* method is the most efficient choice. To use the *Seek* method, you specify which index you want to search, a conditional operator, and one or more values to match.

The *Seek* method works only with table-type recordsets. If you need to query against dynaset or snapshot recordsets, you must use one of the *Find* methods. Linked tables in an Access database cannot be opened as table-type recordsets (*dbOpenTable*) and so you cannot use the *Seek* method directly against a linked table. However, you can use the *OpenDatabase* method to open the database that contains the linked tables, and then you can create table-type recordsets against the tables in that database object.

You can specify one of five conditional operators, shown in the table that follows, that control how the index is searched.

Operator	Result
=	Returns an exact match among all fields in the index. Search begins at the beginning of the index and searches in a forward direction.
>	Returns the first item in the index that is greater than the value(s) specified. Search begins at the beginning of the index and searches in a forward direction.
>=	Returns the first exact match among all fields in the index; or, if no exact match is found, the first match that is greater than the criteria(s). Search begins at the beginning of the index and searches in a forward direction.
<	Returns the first item in the index that is less than the value(s) specified. Search begins at the end of the index and proceeds in a backward direction.
<=	Returns the last exact match among all fields in the index; or, if no exact match is found, the first match that is less than the criteria(s). Search begins at the end of the index and proceeds in a backward direction.

The `Seek` method works with multi-field indexes. When working with a multi-field index, you must specify a criteria for each field in the index if you want an exact match (using the `=` operator). If you don't, `Null` is used to compare any field that doesn't have an explicit criteria defined, which will not result in an exact match if the index has a value defined for the field. This situation can be avoided by using the `>=` operator when working with multi-field indexes.

Seek Pattern

The following pattern is commonly used when working with the `Seek` method:

```
Dim db As DAO.Database
Dim rs As DAO.Recordset2
Dim vbm As Variant
Set db = CurrentDb()
Set rs = db.OpenRecordset("TableName", dbOpenTable)
rs.Index = "IndexName"
```

The recordset must be opened as a table-type recordset, using the `dbOpenTable` constant. Only table-type recordsets support the `Index` property.

```
'// Cache the current record so we can restore if no
'// match is found and the current record is not set
vbm = rs.Bookmark

'// Perform the seek
rs.Seek "=", "FieldValue"
'// if no match is found, restore the bookmark so
'// that we have a current record
If rs.NoMatch Then
    rs.Bookmark = vbm
Else
```

Part II: Data Manipulation

```
'// We found the desired record, do something useful  
Debug.Print rs(0)  
End If
```

When the `Seek` method successfully matches a record, the current record is set to the matched record. You can test the `NoMatch` property to determine if a match was made. If the `NoMatch` property returns `False`, the seek was successful, and the current record pointer is set to the matching row in the recordset.

However, if no matching record is located in the index, the `NoMatch` property is set to `True`, and the current record is undefined. If you attempt to reference any field in the recordset at this point, you will receive a 3021 "No Current Record" runtime error. This is why we store a bookmark before we perform the seek; if no match is found, we can restore the current record to the record that was current before we started the seek.

Move Methods

The `Find` methods and the `Seek` method use the `NoMatch` property to signal whether or not a match was found, but when you use the `Move` methods, there is no concept of finding data. `Move` simply advances or reverses the current record pointer by one or more records. The `Move` method is always successful unless you attempt to move before the first record or after the last record. The `BOF` and `EOF` properties are used to determine whether you are at either end of a recordset. The following table shows the various combinations of `BOF` and `EOF` for each record position in a recordset.

When the Current Record points to...	BOF is ...	EOF is ...	MovePrevious	MoveNext
Before first record	True	False	Results in a 3021 Error	Moves to first record
First Record	False	False	Successful; sets BOF to True	Moves to next record
Any record between the First Record and the Last Record	False	False	Moves to previous record	Moves to next record
Last Record	False	False	Moves to previous record	Successful; sets EOF to True
After last record	False	True	Moves to last record	Results in a 3021 Error

If either the `BOF` or the `EOF` property reports `True`, the current record pointer for the recordset is invalid, and any attempt to reference a field in the recordset will generate a 3021 "No current record" runtime error.

There are five methods in the `Move` family, as shown in the table that follows.

Method	Description
MoveFirst	Moves the current record pointer to the first row in the recordset. Not allowed on forward-only recordsets.
MoveLast	Moves the current record pointer to the last row in the recordset.
MoveNext	Moves the current record pointer to the next row in the recordset.
MovePrevious	Moves the current record pointer to the previous row in the recordset. Not allowed on forward-only recordsets.
Move <i>n</i>	Moves the current record pointer by the specified number of rows. If <i>n</i> is positive, moves the pointer forward <i>n</i> rows; if negative, moves the pointer backward <i>n</i> rows. Only positive values allowed for forward-only recordsets.

The following function demonstrates the interactions between the `BOF` and `EOF` properties and the `Move` methods. It creates a recordset against the `Shippers` table from Northwind. Recall that this table has three rows. We print two header rows to the Immediate Window and then report the value of the `BOF` and `EOF` properties, as well as the `ShipperID` and `Name` to identify which row we are on.

```
Public Sub FunWithBofEof()
    Dim db As DAO.Database
    Dim rs As DAO.Recordset

    Set db = CurrentDb()
    Set rs = db.OpenRecordset("Shippers")
    Debug.Print , "BOF", "EOF", "rs(0)", "rs(1)"
    Debug.Print String(80, "-")
    Debug.Print "After Recordset is created"
    Debug.Print , rs.BOF, rs.EOF, rs(0), rs(1)
```

From the first row, we call `MovePrevious`. This takes us before the first record. The `MovePrevious` method call is successful, but we cannot reference any fields in the recordset because the `BOF` property is `True`, and the current record is invalid.

We also cannot call `MovePrevious` again, because the `BOF` property is `True` and we are already positioned before the first record. If we try to call `MovePrevious`, we receive the same 3021 (No Current Record) runtime error that we receive if we try to reference a field in the recordset. We test this by setting `On Error Resume Next`, and then calling `MovePrevious` and testing for an error. We set `On Error GoTo 0` to resume normal error behavior when we are done with this test.

```
rs.MovePrevious
Debug.Print "After MovePrevious from first row, "; _
    "can't reference rs(0)"
Debug.Print , rs.BOF, rs.EOF, "{3021}", "{3021}"

On Error Resume Next
rs.MovePrevious '// Error, because BOF is True
If Err.Number <> 0 Then
```

Part II: Data Manipulation

```
Debug.Print "Error on MovePrevious; {ERR:"; _
    Err.Number & "}", "BOF: " & rs.BOF, _
    "EOF: " & rs.EOF
Err.Clear
Else
    Debug.Print "OOPS! Expected an error here!"
    Debug.Print , rs.BOF, rs.EOF, rs(0), rs(1)
End If
On Error GoTo 0
```

We can call MoveNext when BOF is True. This sets the current record back to the first row. Both BOF and EOF report False again.

```
rs.MoveNext
Debug.Print "Should be back on first row"
Debug.Print , rs.BOF, rs.EOF, rs(0), rs(1)
```

When we are on the first row, calling MoveNext moves us to the second row, as expected.

```
rs.MoveNext
Debug.Print "Should be on second row"
Debug.Print , rs.BOF, rs.EOF, rs(0), rs(1)
```

When we are on the second row, calling MoveNext moves us to the third row, as expected. Both BOF and EOF report False.

```
rs.MoveNext
Debug.Print "Should be on last row"
Debug.Print , rs.BOF, rs.EOF, rs(0), rs(1)
```

When we are on the last row, calling MoveNext is successful. EOF is now set to True because the current record pointer now points past the last row. Any attempt to reference a field in the recordset generates a 3021 No Current Record runtime error. If we call MoveNext a second time, we receive a 3021 error because we are already pointing past the last record. We again test this by setting On Error Resume Next and then calling MoveNext again and reporting the error we receive.

```
rs.MoveNext
Debug.Print "After MoveNext from last row, ";
    "can't reference rs(0) without error"
Debug.Print , rs.BOF, rs.EOF, "{3021}", "{3021}"

On Error Resume Next
rs.MoveNext      // Error, because EOF is True
If Err.Number <> 0 Then
    Debug.Print "Error on MoveNext; {ERR:" _
        Err.Number & "}", _
        "BOF: " & rs.BOF, "EOF: " & rs.EOF
    Err.Clear
Else
    Debug.Print "OOPS! Expected an error here!"
    Debug.Print , rs.BOF, rs.EOF, rs(0), rs(1)
End If
On Error GoTo 0
```

From the EOF position, we can ask to move back five rows. Because the table has only three rows, we are now pointing before the first row. BOF now reports True, and EOF should report False (in Access12, EOF incorrectly reports True in this case). Because BOF is True, we cannot reference any fields in the recordset.

```
rs.Move -5
Debug.Print "Attempt to Move -5 rows, beyond first row"
Debug.Print , rs.BOF, rs.EOF, "{3021}", "{3021}"
```

Finally, a MoveNext returns us to the first row in the recordset. BOF and EOF are now False.

```
rs.MoveNext
Debug.Print "Should be back on first row"
Debug.Print , rs.BOF, rs.EOF, rs(0), rs(1)
rs.Move 2
Debug.Print "Attempt to Move +2 rows, ";
"should be on last row"
Debug.Print , rs.BOF, rs.EOF, rs(0), rs(1)

Cleanup:
rs.Close
Set rs = Nothing
Set db = Nothing

End Sub
```

Search Optimization Tips

There are a couple of tips that will increase the efficiency of your search operations.

First, set operations (such as SELECT statements used to build a recordset) are almost always more efficient than cursor-based operations such as the FIND methods. Where possible, use queries with effective WHERE clauses to identify the specific records you are looking for. Instead of this code, which creates a large recordset that uses Find methods to isolate records:

```
criteria = "[EmployeeID]='3' AND [CustomerID] LIKE 'ALFKI'"
Set rs = db.OpenRecordset("SELECT * FROM [Orders]")
rs.FindFirst(criteria)
Do While Not rs.NoMatch
    // Do something useful
    rs.FindNext(criteria)
Loop
```

consider this code, which returns only the records of interest to begin with:

```
sql = "SELECT * FROM [Orders] WHERE [EmployeeID]='3' AND [CustomerID] LIKE 'ALFKI'"
Set rs = db.OpenRecordset(sql)
Do While Not rs.EOF
    // Do something useful
    rs.MoveNext
Loop
```

This is particularly important when you have linked tables or are querying data stored on a network.

Part II: Data Manipulation

The `Find` methods use indexes if you construct your criteria effectively. When possible, make sure that your criteria are defined against indexed fields. For the best performance, criteria should be simple equality expressions against an indexed field (`[IndexedField] = "some value"`). If you need to perform wildcard searches, use the `LIKE` operator with a prefix search string that does not begin with a wildcard character. This search expression can use an index:

```
rs.FindFirst "[LastName] LIKE 'TUC*' "
```

This search expression, however, cannot use an index, and is much slower to run:

```
rs.FindFirst "[LastName] LIKE '*KER'"
```

In general, all of the performance optimization tips in Chapter 5 apply to criteria used with the `Find` methods.

There are some tricks when using the `Find` methods on a system with customized regional settings. If you need to use one of the `Find` methods with a criteria that includes date values, you should use the default US date formats (`mm/dd/yy` or `mm/dd/yyyy`) — even when you are working with a localized version of the product or on a system with non-US regional settings — because the Microsoft Access database engine SQL language always expects date values to be US formatted. You can use the `Format` VBA function to convert a date value to the default US standard value in your criteria.

```
rs.FindFirst "[OrderDate] > #" & Format(dtValue, _ "mm/dd/yyyy") & "#"
```

If you will be doing this frequently, consider creating a helper function that you can use to simplify the SQL statement, such as the following:

```
Public Function SqlDate(item As Date) As String
    SqlDate = "#" & Format(item, "mm/dd/yyyy") & "#"
End Function
```

Now, you can create your `FindFirst` statement like this:

```
rs.FindFirst "[OrderDate] > " & SqlDate(dtValue)
```

Categorization and Sorting

When presenting data in user interface elements, it is often desirable to provide categorization hints to the user. Categorized data is easier for the user to parse, and makes it faster (and often less frustrating) to locate those bits of information that are most important to the user. The more information that you attempt to present to your user, the more important it becomes to provide a categorized view of your data.

Likewise, sorted data makes it easier to zoom in on the data your user is attempting to locate. We've already looked how to use `ORDER BY` clauses in queries and Record Source properties to provide a sorted view of data based on the data itself, but often, you will want to customize the sort process. Either the data doesn't provide enough hints to sort on, or it is not in the correct format to be sorted the way it needs to be. In this section, we look at ways to overcome these challenges.

Categorization Using the Choose Function

The Choose VBA function is a flexible way to return category information for an item when you have a limited number of items. The function accepts an Index and a parameter array of expressions. Based on the value of the one-based Index argument, the corresponding expression is evaluated and the result returned, as follows:

```
Public Function CategoryFromIndex(i As Long) As String
    CategoryFromIndex = Choose(i, _
        "Fruits", "Vegetables", "Meats", "Fruits", _
        "Desserts", "Doilies")
End Function
```

In this simple example, passing in an Index of one or four returns the category `Fruits`. Passing in two returns `Vegetables`, three returns `Meats`, and five returns `Desserts`. This function doesn't scale well; if you have more than a handful of categories, you are better off creating a lookup table.

The parameter array accepts expressions that are evaluated by the Expression Service. This would be enormously handy, except that each expression in the array gets evaluated every time the function is called, even if it is not a match. If your expressions update data, increment values, or display UI, you will get results you didn't intend (and probably don't want).

The following example demonstrates the problem. When the `Choose` function is called, even though you passed three, you expect only to get a message box that says `Goodbye` and then a value of 1 (the return value from the `MsgBox` function that indicates you pressed the OK button), but instead you see all three message boxes.

```
Dim strChoice As String
Dim i As Long
i = 3
strChoice = Choose(i, MsgBox("Hi"), MsgBox("Hello"), _
    MsgBox("Goodbye"))
```

Our recommendation is to use the `Choose` function only when you have a limited number of simple expressions.

Using a Custom Sort Order Field

By now you are probably comfortable using `ORDER BY` clauses in queries to sort your data based on values in fields in your tables and queries. But there are many situations where we need to categorize data in a seemingly arbitrary fashion. Other times, you may have all the information you need to perform your sort in fields in your tables, but you need to extract a subset of the data to perform the sort. In this section, we look at a few different scenarios and how we can easily solve custom sorting dilemmas.

Our first example illustrates a common problem with titles: what to do with leading articles. Our sample data is a small list of movies that were released in 2005. We were going to use an extract from IMDB, until we realized that over 21,000 movies were released in 2005! So instead, we've plucked about 30 titles and stuffed them into a table in the `CommonSort.accdb` supporting file. The table that follows lists some of the movies in the `Movies1` table.

Part II: Data Manipulation

Movies1 : Title
1001 vies de Lia Rodrigues, Les
10th Amendment Project, The
14 Days in America
About Face
Abrazo de la tierra, El
Absente, L'
Absolut Böse, Das
Abstract Expression, An
Accordion, The
Acera de enfrente, La
Aces Texas Hold 'Em: No Limit
Adventure Golf Guy, The
Adventures of Buster Smith, The
Alice in Wonderland
All Souls Day: Dia de los Muertos
Amants réguliers, Les
Amants, Les

Several of these movie titles begin with articles: The, Les, El, L', and so on. It is common practice not to sort titles based on the leading article, but to sort beginning with the first term following the article. This poses a dilemma for database designers. There are a few different ways to handle the situation, none superior to the other. Each technique has pros and cons, but each one relies on consistent data entry.

IMDB has chosen to solve the problem by removing the leading article and appending it to the end of the movie title following a comma delimiter. In the table below, you find the movie “The 10th Amendment Project” listed as “10th Amendment Project, The.” This solution solves the sorting problem quite nicely: just sort on the title field and everything shows up where you expect it too — assuming, of course, that you remembered to enter each title correctly, and move leading articles to the end of the title. If you are creating a new database and populating the data manually, this is a good and performant solution for building fast and efficient queries, but at the cost of much more complex and error prone data entry. You

may think we're kidding, but when you are heads down on data entry, it's hard to remember to strip off the leading articles!

```
SELECT Title FROM Movies1 ORDER BY Title
```

Now, let's consider the case of an import. You've been given a list of movie titles (shown in the following table2, and available as the table `Movies2` in the sample database), but without the convenient concatenated leading articles. Instead, your movie titles look like the table that follows.

Movies2: Title	SortOffset
14 Days in America	
About Face	
Aces Texas Hold 'Em: No Limit	
Alice in Wonderland	
All Souls Day: Dia de los Muertos	
An Abstract Expression	4
Das Absolut Böse	5
El Abrazo de la tierra	4
La Acera de enfrente	4
L'Absenté	3
Les 1001 vies de Lia Rodrigues	5
Les Amants	5
Les Amants réguliers	5
The 10th Amendment Project	5
The Accordion	5
The Adventure Golf Guy	5
The Adventures of Buster Smith	5

Data formatted this way is easier to enter: Just enter the title the way it is displayed/pronounced. However, when you attempt to sort this table, the records do not appear in the expected order because of the leading articles.

Part II: Data Manipulation

Rather than editing all of the items to move the articles to the end of the title, consider adding a new column to the table (perhaps labeled `SortOffset` as shown in the preceding table. Then, for those items with a leading article in the title, enter an offset value that indicates where you want the sorting to begin. For example, the movie “The 10th Amendment Project” has a three character long leading article and a space, so we really want our sorting to begin in the 5th position. The movie “El Abrazo de la tierra” has a two-character long-leading article, plus a space, so we want to begin sorting at position 4.

You can create an update query to quickly fill in the `SortOffset` field for the common cases (The, A, An, Le, La, Les, and so on). You want to do a quick visual scan of your data for other leading articles or other constructs that you wish to omit from sorting. If you have a lot of custom terms that you want to omit, consider creating a table of items that you want to exclude, which you can then link into the data table.

Once we have filled in the `SortIndex` field, we can create an expression that strips off the leading article. Then, we can perform the sort on this calculated expression. We use the `Mid$` VBA function, passing in the `Title` field, and the `SortOffset` value. We wrap the `SortOffset` value in the `NZ` VBA function, so that we do not have to populate the `SortOffset` field for those rows that do not need to be trimmed. By defaulting to 1 for rows without a `SortOffset` value, we receive the entire `Title` string unmodified.

The following query presents the movie titles sorted in the expected order:

```
SELECT Title, Mid$(Title,NZ(SortOffset,1)) AS SortTitle  
FROM Movies2  
ORDER BY Mid$(Title,NZ(SortOffset,1))
```

The following expression converts the movie title into the format that IMDB uses (with the trailing articles):

```
Mid$([Title],NZ([SortOffset],1)) &  
IIf(Not IsNull([SortOffset]),", " & Left$([title],[SortOffset]-1))
```

The following query combines all three expressions into one data grid:

```
SELECT Title,  
      Mid$([Title],NZ([SortOffset],1)) AS SortExpression,  
      Mid$([Title],NZ([SortOffset],1)) &  
      IIf(Not IsNull([SortOffset]),", " &  
           Left$([title],[SortOffset]-1))  
           AS ConvertedExpression  
FROM Movies2  
ORDER BY Mid$(Title,NZ(SortOffset,1))
```

Figure 7-1 shows movie titles sorted and converted.

Backup and Restore

Access provides a simple backup and restore mechanism for the traditional file-based backup system. To back up your entire database file, first ensure that all users are logged out of the file. Then, from the Office menu, choose Manage, and select Back-Up database from the flyout.

Figure 7-2 shows initiating a backup from Microsoft Access.

Title	SortExpression	ConvertedExpression
Les 1001 vies de Lia Rodrigues	1001 vies de Lia Rodrigues	1001 vies de Lia Rodrigues, Les
The 10th Amendment Project	10th Amendment Project	10th Amendment Project, The
The 11th Day	11th Day	11th Day, The
14 Days in America	14 Days in America	14 Days in America
About Face	About Face	About Face
About Freedom	About Freedom	About Freedom
About Love	About Love	About Love
About Sofia	About Sofia	About Sofia
Above and Beyond	Above and Beyond	Above and Beyond
El Abrazo de la tierra	Abrazo de la tierra	Abrazo de la tierra, El
L'Absenté	Absente	Absente, L'
Das Absolut Böse	Absolut Böse	Absolut Böse, Das
An Abstract Expression	Abstract Expression	Abstract Expression, An
L'Accord	Accord	Accord, L'
The Accordion	Accordion	Accordion, The
La Acera de enfrente	Acera de enfrente	Acera de enfrente, La
Aces Texas Hold 'Em: No Limit	Aces Texas Hold 'Em: No Limit	Aces Texas Hold 'Em: No Limit
The Adventure Golf Guy	Adventure Golf Guy	Adventure Golf Guy, The
The Adventures of Buster Smith	Adventures of Buster Smith	Adventures of Buster Smith, The
After the End	After the End	After the End
After the Rain	After the Rain	After the Rain
Alice in Wonderland	Alice in Wonderland	Alice in Wonderland
All About Darfur	All About Darfur	All About Darfur
All Souls Day: Dia de los Muertos	All Souls Day: Dia de los Muertos	All Souls Day: Dia de los Muertos
The Alma Drawings	Alma Drawings	Alma Drawings, The
Les Amants réguliers	Amants réguliers	Amants réguliers, Les
Les Amants	Amants	Amants, Les
*		

Figure 7-1



Figure 7-2

When you ask Access to back up your database, it will make a copy of the database file and store it, by default, in the same directory, with the date of the backup appended to the filename. You may choose to

Part II: Data Manipulation

store the backup in another folder, and to rename the backup file. (We often append the time to the filename as well as the date.) Access will remember the location you choose to store backups to and will automatically increment the suggested filename.

This is an extremely easy way to perform backups, and is particularly useful during application development. We cannot stress enough the importance of maintaining backups of your work as you develop your application.

Using Checksum Functions

We have included a Microsoft .NET solution in the sample files for this book that provides the source code in C# and VB.NET for creating a library of checksum functions that can be consumed by Microsoft Access. In this section, we look at how you can use the functions in this library to create hash values and checksum from VBA code and queries in Microsoft Access.

We provide the complete source code so that you can examine how the library is constructed, and so that you can extend the library to support your favorite hash algorithm. Details for how to construct a Microsoft .NET Framework assembly that can be consumed from VBA is beyond the scope of this work. For detailed information on how to accomplish this task, check out the Wrox publication “Access 2007 VBA Programmer’s Reference.”

To use this library from Microsoft Access:

- ❑ Copy the SimpleMD5.dll and tlb files to a folder on your system. You can install them in the same directory as your application, or place them in a shared location.
- ❑ Add a reference to the SimpleMD5 Type Library. Browse to the directory where you installed the files and choose the SimpleMD5.tlb type library.
- ❑ Create a new code module named Checksums. Add a global variable that instantiates a SimpleMD5.MD5Checksum class using the New keyword.

```
Option Compare Database
Option Explicit
Dim MD5Class As New SimpleMD5.MD5Checksum
```

- ❑ Add a function that wraps a call to the Calculate method.

```
Public Function MD5ChecksumWrapper(item As String) _
As String
MD5ChecksumWrapper = MD5Class.Calculate(item)
End Function
```

Try out this function by adding an MD5 hash of the movies titles to the Movies table we worked with earlier. Add a text field of length 32 to the Movies table named MD5Title. Then, create and run the following query:

```
UPDATE [Movies]
SET [MD5Title] = MD5ChecksumWrapper([Title])
WHERE [MD5Title] Is Null
```

You can call the wrapper function from your VBA code, inside a query, or anywhere an expression can be evaluated.

MD5 Algorithm

MD5 (Message-Digest algorithm 5) is a hash function that is used to generate a 128-bit hash value. Given any arbitrary input, it generates a 128-bit hash value that is displayed as 32 hexadecimal digits. Our simple library implementation enables you to pass in any arbitrary string value to the function. The function returns the calculated 128-bit hash value. The algorithm is deterministic: every time you pass the same content into the function, you receive the same 128-bit hash value back.

Wikipedia has detailed information on a wide variety of hash functions. Browse to www.wikipedia.org and search on "Hash Function." Detailed information on the MD5 hash function is located at <http://en.wikipedia.org/wiki/MD5>.

MD5 hash values are useful as shorter keys for lengthy data. When you import data, oftentimes there are no primary key values, or the primary key values are lengthy strings. You can compute an MD5 hash value on the lengthy string, and store the MD5 hash value as your primary key.

One real-world example of doing this is when we import the Internet Movie Database (IMDB) text files into an Access database. IMDB uses the name of a movie (often combined with symbols and abbreviations) as the key value that links together the 30-odd files that comprise the complete IMDB dataset. We could assign an AutoNumber to each movie and use that as a key, but then we would have to perform a lookup into the movies table (which has nearly a million rows!) each time we needed to resolve a movie name in one of the other tables. For example, when we're parsing the `actors.list` table (which has one entry for each movie that each actor has appeared in), we need to identify the key for each movie. Instead of doing a lookup into the movies table to identify the autonumber that was assigned to the movie, we calculate the MD5 hash value for the string in the `actors.list` table, and then use the MD5 hash value as the foreign key that we store in the `MoviesActors` association table.

If you are a movie enthusiast, you'll find a treasure trove of data just waiting to be parsed at www.imdb.com/interfaces. Most of the content on the IMDB site is available for free download for non-commercial purposes. The data is not Access friendly; at the very least you'll need to write some complex VBA code to parse the files into Access tables. The Web site for this book also has some sample code written in C# and VB.NET to parse some of the IMDB data files.

Mod-10 Checksum

Mod-10 (also known as the modulus 10 algorithm, and more formally known as the Luhn Algorithm) is the checksum formula that is used to validate a wide variety of account numbers, including most credit cards issued world wide.

Given a base account number, a single digit Mod-10 checksum — in the range of 0 to 9 — is generated. This checksum number is deterministic, and is concatenated to the base account number to create the full account number. In a sixteen-digit credit card number, the first 15 digits represent the base identification number, and the sixteenth digit represents the Mod-10 check digit. The checksum is not designed to provide any sort of cryptographic protection or validation. It is designed to catch simple transposition and other data entry errors.

Part II: Data Manipulation

The following paragraphs describe the process of calculating a Mod-10 checksum in potentially gruesome detail. If you don't care about the details of the implementation, you can skip to the discussion of how to use the Mod-10 checksum functions presented below.

How the Mod-10 Checksum Algorithm Works

The checksum is calculated against the base account number by walking the digits in the base number from right to left. In brief, each digit in the base number is multiplied by a factor. Digits in odd-numbered positions (first, third, fifth, and so on, counting from right to left) are multiplied by 2. Digits in even-numbered positions (second, third, and so on) are multiplied by 1. The results of this multiplication are further processed by summing the digits of the result. Finally, the sum of the digits is accumulated, and then passed into an equation that results in a single digit checksum result.

Let's walk through an example of calculating the Mod-10 checksum for an account number. We will calculate a checksum for the base account number 456789. Refer to the data in the following table as we discuss the steps.

Digits in base account number	4	5	6	7	8	9
Factor (R to L, begin at 2)	1	2	1	2	1	2
Result (Digit * Factor)	4	10	6	14	8	18
Sum of Digits in Result	4	1+0=1	6	1+4=5	8	1+8=9

We assign the alternating factor to digits from right to left. The right-most digit is always assigned a factor of 2. We alternate the factor between 2 and 1 for each digit as we move right to left.

Each digit in the base account number is then multiplied by the factor. This value is shown in the table in the results row. Next, we sum the digits of the result. If the result of the multiplication is less than 10, there is only one digit in the result, and so the sum of the digits is the same as the result. But if the result is greater than 10, we add together the value of the digits in the result. This is displayed in the Sum of Digits row in the previous table.

The result of the Sum of Digits calculation is accumulated for each digit in our base account number. In the example, the accumulated value of the sum of the digits is $4 + 1 + 6 + 5 + 8 + 9 = 33$.

Finally, we run the accumulated value through this equation to arrive at the check digit:

```
(10 - (accumulatedValue Mod 10)) Mod 10
```

This results in a check digit of 7. So base account number 456789 results in a final account number of 4567897.

Wikipedia has a wealth of information a large number of check digit algorithms, including the Luhn Algorithm. Browse to www.wikipedia.org, and search on "Checksum Algorithms." A detailed description of the Luhn algorithm, complete with implementations in several programming languages can be found at http://en.wikipedia.org/wiki/Luhn_algorithm.

Using VBA to Calculate Mod-10 Checksums

Over the next several pages, we create a VBA solution to calculate Mod-10 checksums, as well as a helper function to easily validate credit card numbers and other numbers that contain Mod-10 check digits.

Our solution uses a multi-dimensional array to store pre-calculated Sum of Digits values for each possible value in a Mod-10 number. Rather than making our VBA code parse each digit in the multiplied result, we take advantage of the fact that for any given digit in a base account number, there are only two possible Sum of Digits results, depending upon whether the factor is 1 or 2.

The lookup array has two ranks of nine items. The first rank (rank 0) contains the non-doubled values that are accumulated for digits in the even-numbered positions (remember that we count from right to left). The second rank (rank 1) contains the sum of digits for the doubled values that are accumulated for digits in the odd-numbered positions. The following table shows the values that are stored in the Sum of Digits multi-dimensional array.

Rank/Digit	0	1	2	3	4	5	6	7	8	9
0 (Factor=1)	0	1	2	3	4	5	6	7	8	9
1 (Factor=2)	0	2	4	6	8	1	3	5	7	9

We use a module level multi-dimensional array to store the sum of the digits lookup, and a module level variable to store an initialization flag.

```
Private m_rgSums(0 To 1, 0 To 9) As Variant
Private m_bInitialized As Boolean
```

Next, we create a function to load the multi-dimensional array with the sums.

```
Private Sub PrepareLookupArray()
    Dim rgRank As Variant
    rgRank = Array(0, 2, 4, 6, 8, 1, 3, 5, 7, 9)
```

This single-dimension array is created merely to simplify the loading of the multi-dimensional array. This single-dimensional array contains the “sum of digits” for the doubled digits that are loaded into the second rank of the lookup array. It would be very nice if there were a way to use the very convenient `Array` function to load a multi-dimensional array; unfortunately, this is not an option. We could avoid the need for this array by making 20 separate calls to load each of the array values directly, but that makes for messy code. Our solution is to loop from 0 to 9 and fill each rank inside the loop.

```
Dim i As Long
For i = 0 To 9
    '// load the multi-dimensional array
    m_rgSums(0, i) = i
```

The first rank of the array contains the non-doubled values, 0 to 9, and so we can use the loop variable to load the value.

```
m_rgSums(1, i) = rgRank(i)
```

Part II: Data Manipulation

The second rank contains the sum-of-digits of the doubled values. We load the second rank by referencing the values in the temporary array we just created using the `Array` VBA function. There is a definite performance hit to creating this extra array; however, this function is called only once when we initialize the lookup array, and the hit is small.

```
    Next i
    ' // report the array initialized
    m_bInitialized = True
End Sub
```

Once the multi-dimensional array is loaded, we set the `m_bInitialized` flag to `True` and return.

Now, let's look at the `CalculateChecksum` function, which actually generates the check digit.

```
Public Function Checksum(item As String) As Integer
    Dim sum As Integer      ' // accumulates sums
    Dim odd As Integer      ' // is this an odd or even character position
    Dim i As Integer
    Dim dwChar As Integer

    ' // Validate argument
    If Not IsNumeric(item) Then
        MsgBox "item argument must consist of digits only!", vbCritical
        Checksum = -1
        Exit Function
    End If
```

First, we validate the argument, using the `IsNumeric` VBA function. `IsNumeric` returns true if every character in a string is a valid numeric digit (the characters +, -, and E are also considered valid digits by the `IsNumeric` function, to support scientific notation). If `IsNumeric` returns `False`, we exit the function with an error message.

```
' // initialize the lookup array
If Not m_bInitialized Then PrepareLookupArray
```

Next, we check the `m_bInitialized` flag to see if we need to prepare the lookup array, and call `PrepareLookupArray` if necessary.

```
sum = 0
odd = 1
```

We zero out the `sum` accumulator variable. The `odd` variable keeps track of whether the current digit of the account number that we are processing is an odd or even digit. This variable serves as the rank index into the lookup array.

```
For i = Len(item) To 1 Step -1
    dwChar = Val(Mid$(item, i, 1))
```

We use a `For` loop to walk through the account number string one character at a time, from the back of the string to the front of the string. The current character is isolated and stored in the `dwChar` variable.

```
    sum = sum + rgFactors(odd, dwChar)
```

The `odd` and `dwChar` variables serve as indexes into the lookup array. When the current character occupies an odd-numbered position in the string, the value of the `odd` variable will be 1 and the `dwChar` variable contains the value of the digit at the current character. When we are in an odd-numbered position, the value of the character needs to be doubled and the digits summed. The summed values are stored in rank 1 of our lookup array, and so we can use the `odd` variable directly as our rank index, and the `dwChar` variable as our position index.

Conversely, when the current character occupies an even-numbered position in the string, the value of `odd` will be 0 and `dwChar` will still contain the value of the digit at the current character. Rank 0 contains the summed values we need for even numbered positions (simply the value of the character).

Using these variables, we look up the sum from the lookup array and add it to the accumulated value stored in the `sum` variable.

```
    odd = (odd + 1) Mod 2
    Next i
```

Next, we toggle the value of the `odd` variable, by incrementing the variable by 1, and then performing a `Mod 2` on the result. Recall that `Mod` — modulus division — returns the remainder of a division operation. So if the value of `odd` is 0, `(0+1) Mod 2` returns 1 (because 1 divided by 2 is 0 with a remainder of 1). If the value of `odd` is 1, then `(1+1) Mod 2` returns 0 (because 2 divided by 2 is 1 with a remainder of 0). In this way, by consistently incrementing `odd` by 1 each time through the loop, we effectively toggle the value of `odd` between 0 and 1.

The `For` loop repeats until we reach the first character in the input string.

```
Checksum = (10 - (sum Mod 10)) Mod 10
End Function
```

Once we have accumulated the sums for each character in our input string, we calculate the checksum using the `Mod-10` checksum equation. We assign the value of this calculation to the return value for our function, and then terminate.

To validate an account number, you must remove the final digit of the number, and pass the remaining digits into the checksum function. The checksum function will calculate the check digit, which you can then compare to the final digit you removed. If the value returned by the function matches the final digit you removed, the account number is a valid number.

To make things easier, create a helper function to tie everything together. `IsValidAccountNumber` takes a complete account number (for example, a complete credit card number as supplied by a customer) and returns a Boolean value indicating whether or not the account number is a valid number according to the Luhn algorithm.

The default return value is set to `False`, so that if any errors occur, we don't incorrectly report a valid account number.

```
Public Function IsValidAccountNumber(ccnum As String) _
    As Boolean

    Dim checkDigit As Integer
    IsValidAccountNumber = False
```

Part II: Data Manipulation

Just as with the `Checksum` function, we need to validate that the string passed into this function consists of digits. However, there are a few extra conditions we need to test for. If there are non-numeric characters in the string, we terminate the function and report an error.

```
If Not IsNumeric(ccnum) Then
    MsgBox "ccnum must consist of digits only!", _
        vbCritical
    Exit Function
```

If an empty string is passed, pass the function. This supports the scenario where `IsValidAccountNumber` is called in a `BeforeUpdate` event handler (we'll look at this scenario shortly).

```
ElseIf vbNullString = _
    Trim$(Nz(ccnum, vbNullString)) Then
    IsValidAccountNumber = True
```

Finally, a valid account number must have at least two digits: at least one base digit, and the calculated check digit.

```
ElseIf Len(ccnum) < 2 Then
    MsgBox "A valid account number must have " & _
        "at least two digits.", vbCritical
    Exit Function
End If
```

Next, we locate the last number of the string and cache it into a variable. This is the digit that we compare to the check digit that we calculate.

```
// store the checksum
checkDigit = Mid$(ccnum, Len(ccnum), 1)
```

We pass the account number string (minus the last character) to the `Checksum` function to calculate the Luhn check digit. Then we compare this to the value of the final character in the string that we cached in the `checkDigit` variable. If these two variables match, then the account number string that was passed into this function is a valid account number and we return `True`; otherwise, the account number is invalid and we return `False`.

```
IsValidAccountNumber = _
    (checkDigit = Checksum(Left$(ccnum, Len(ccnum) - 1)))
End Function
```

Pull out your wallet or purse and try it with your credit cards. Enter the account number from your credit cards and you should see `True` returned from the `IsValidAccountNumber` function. Try changing some of the digits and see how the algorithm correctly detects the alteration. Finally, try entering some other account numbers (bank accounts, electric and gas company accounts, and so on). Luhn checksums are used for many other purposes than just credit card account numbers.

You can hook this function up to the `AfterUpdate` event handler of a text box on a form, to validate data entry of credit card and other account numbers. Try it out! Create a blank form. Add a text box

named `Text0` and a button to the form. Add the following code to the `AfterUpdate` event handler of the text box:

```
Private Sub Text0_BeforeUpdate(Cancel As Integer)
    If Not IsValidAccountNumber(Me.Text0) Then
        Beep
        MsgBox "Invalid account number!", vbOK, _
            "Luhn Checksum Validation Failure"
        Cancel = True
    End If
End Sub
```

This code traps keyboard focus in the text box until a valid account number is entered. This is why it is important to treat an empty string as a valid account number. If we don't allow empty strings, the user could be trapped forever in the text box (or at least until they guessed a valid account number, and you don't want data entry personnel guessing at data)! By accepting an empty string as valid, you allow the user to simply clear the text box to release focus.

Access 2007 Specifics

Access 2007 introduced many new features. In this section, we look at two of the most useful: Append Only memo fields, and Attachment fields.

An Append Only memo field is a variant of the Memo data type. The Memo data type now has an `AppendOnly` property. If this property is set to `True`, users can add data to the field, but existing entries are preserved, along with a time stamp. Only the most recent entry is displayed in the field in a data grid or a form, but you can access the historical entries preserved in this data type using code.

Attachment fields build on the Multi-valued lookup field type added in Access 2007. This data type enables you to store zero or more files in a field. The files are even automatically compressed to save space. A thumbnail of the first file in the field (alphabetically speaking) is displayed in an attachment control on a form, making an attachment field an ideal replacement for the Image control, especially if you need to be able to store multiple images, but really only need to be able to display one.

Locale-Agnostic Parser for ColumnHistory Function

Although adding data to an `AppendOnly` memo field is straightforward (just click into the memo field and start typing!), retrieving the history can be challenging. The `Application.ColumnHistory` function provides access to the items stored in the column. Unfortunately, this function returns the data in a single, formatted string blob, which must be parsed in order to separate the individual entries and the time-stamps. Additionally, although an `AppendOnly` Memo field can set the `TextFormat` property to `Rich Text`, thus allowing users to apply rich text formatting to data stored in the field, the `ColumnHistory` function strips out all user-applied rich formatting.

The `ColumnHistory` function contains three arguments, as shown in the table that follows.

Part II: Data Manipulation

Argument	Description
TableName	Name of the table that contains the AppendOnly Memo field you want to query.
ColumnName	Name of the AppendOnly Memo field you want to query.
queryString	A string used to locate the specific record you want for the Column History. This string is a SQL WHERE clause without the word WHERE. It must return one and only one row. If the function is called from code on a form bound to the table listed in table name, this argument may be passed as an empty string and the function will use the current row for the form.

Output from the ColumnHistory function looks like this:

```
[Version: 22/05/2007 13:07:25 ] My first brief comment  
[Version: 22/05/2007 13:07:34 ] My second brief comment  
[Version: 22/05/2007 13:08:51 ] Bold Red Underlined Blue Italic HugeCourierNew
```

The date and time is formatted according to the regional settings you have defined for "Short Date" and "Long Time".

The following function demonstrates how to work with output from the ColumnHistory function. This function is designed to be called from a form that contains a list box. We parse the output into three columns in the list box.

Our function accesses the three arguments needed by the Application.ColumnHistory function, and also accepts a reference to the Access.ListBox control that we will use to display the parsed Column History content.

```
Public Sub ShowColumnHistory(strTableName As String, _ strFieldName As String, _  
    queryString As String, _  
    ctlList As Access.ListBox)  
    'History data is in this format:  
    '[Version: Date Time ] History Data  
    Const VERSION_PREFIX As String = "[Version: "  
  
    Dim strHistory      As String  
    Dim astrHistory()   As String  
    Dim strHistoryItem As String  
    Dim pchEndDelim    As Long  
    Dim pchCRLF        As Long  
    Dim lngCounter     As Long  
    Dim strDateTime    As String  
    Dim datDate        As Date  
    Dim datTime        As Date  
  
    'Get the column history  
    On Error Resume Next
```

Here, we call `Application.ColumnHistory` and store the output into a string variable. We test for a specific error case because if the `queryString` passed into this function doesn't locate a row, the `ColumnHistory` function throws this error (later in this section, we look at reasons why the `queryString` might not locate a row).

```
strHistory = Application.ColumnHistory( _
    strTableName, strFieldName, queryString)
If Err.Number <> 0 Then
    If Err.Number = -2147467259 Then
        'probable cause is an invalid query string
        'resulting from the new row
        Err.Clear
        ctlList.RowSource = vbNullString
        Exit Sub
    End If
End If
```

We split the `ColumnHistory` output into the individual records, and then we clear the contents of the list box. We choose to split on the `VERSION_PREFIX` rather than `vbCrLf` because it is perfectly legal to have CRLF combinations inside the data string. By splitting on the `VERSION_PREFIX`, we ensure that we get a complete history entry in each element of the split array.

```
'Make sure there is history data
If Len(strHistory) > 0 Then
    astrHistory = Split(strHistory, VERSION_PREFIX)
    ctlList.RowSource = vbNullString
```

The array is enumerated in reverse order (so that the most recent entry appears at the top of the list; if you wish to see the column history in chronological order, loop through from `LBound` to `UBound` and change the step value to `+1`). Each entry is saved to the `strHistoryItem` variable, which is modified extensively through this loop. Finally, if there is a trailing CRLF on this history item, it is removed, so that we do not write empty lines between each history item.

```
For lngCounter = UBound(astrHistory) To LBound(astrHistory) Step -1
    strHistoryItem = astrHistory(lngCounter)
    If Right$(strHistoryItem, Len(vbCrLf)) = _
        vbCrLf Then
        strHistoryItem = _
            Left$(strHistoryItem, _
                Len(strHistoryItem) - Len(vbCrLf))
    End If
```

Next, we parse the individual history item into three separate pieces: the date the item was created, the time it was created, and the content of the item. The date and time pieces are parsed using the `Format` VBA function. The entire date/time string is passed to the `Format` function, and a named format string is used to create the output that is stored.

```
pchEndDelim = InStr(strHistoryItem, "])"
If Len(strHistoryItem) > 0 Then
    strDateTime = Trim$( _
        Left$(strHistoryItem, pchEndDelim - 1))
```

Part II: Data Manipulation

```
datDate = CDate( _
    Format(strDateTime, "Short Date"))
datTime = CDate( _
    Format(strDateTime, "Long Time"))
```

Finally, we locate the "]" character, which closes the date/time block, and stores all of the data that comes after that character back into the strHistoryItem variable.

```
'Separate the item data
strHistoryItem = Mid(strHistoryItem, _
    InStr(strHistoryItem, " ] ") + 3)
```

We write the contents of strHistoryItem to the list box. Because of how the List Box works, if strHistoryItem contains any embedded CRLF characters, only the content that appears before the first CRLF combination is actually displayed in the list box. Because it is quite possible that we will have embedded CRLFs in our history data, we now proceed to check for these characters, and extract any extra lines and add them to the list box, too.

```
ctlList.AddItem datDate & ";" & _
    datTime & ";" & strHistoryItem
    pchCRLF = InStr(strHistoryItem, vbCrLf)
```

This loop is a standard pattern used to identify a character sequence within a string. First, we check to see if the sequence (in this case, vbCrLf) exists within the string. If it does, we enter a loop. Inside the loop, we trim off the content that came before the sequence (because we have already written that to the list box). Then we add everything remaining to the list box. We include two semicolons because we do not want to write anything to the date and time columns in the list box, because this line continues the line we previously wrote.

```
Do While pchCRLF > 0
    ' strip off the content that has
    ' already been written
    strHistoryItem = Mid$(strHistoryItem, _
        pchCRLF + Len(vbCrLf))
    ctlList.AddItem ";" & strHistoryItem
    pchCRLF = InStr(strHistoryItem, vbCrLf)
Loop
End If
Next
End If
End Sub
```

Just as before, if there are any additional CRLF combinations remaining in the string, only the content that appears before the first CRLF shows up in this row of the list box. Finally, we repeat the check for a vbCrLf. If there are more CRLFs in this string, we reenter the loop; otherwise, we are finished writing this item and can exit the loop, finish this individual history item, and check the next one in the array.

To use this function, you want to create a table named AppendOnlyTable that contains an AutoNumber field named ID and a memo field named Comments with the AppendOnly property set to True. Then create a form based on this table. Add a text box named txtID bound to the ID field. Add a ListBox named lstColumnHistory to the form. Set the properties of the ListBox, as shown in the table that follows.

List Box Property	Value
Name	lstColumnHistory
ColumnCount	3
ColumnHeads	True (or Yes in the property sheet)
RowSourceType	Value List
RowSource	Date;Time;History

Finally, add a button to the form. In the `onClick` event handler, add the following code:

```
ShowColumnHistory "AppendOnlyTable", "Comments", _
    "ID=" & Me.txtID, Me.lstColumnHistory
```

Run the form and add a few items to the table. For each item, add one or more comments to the `Comments` field. Press the `Save` button on the toolbar to save the form, and then reenter the `Comments` field and enter another comment.

Once you have a few comments entered, press the button you created to run the code. Step through the code and watch the variables as we parse the `ColumnHistory` list.

You can also call this function from the form's `OnCurrent` event handler. When you do this, every time you move to a new record on the form, the list box will be updated with the `ColumnHistory` data for this row. If you move the form to the New row (where you can create a new item in the table), there is no ID number assigned. This is the case we mentioned earlier, where the `queryString` can be invalid. When you move to the new row, the `queryString` argument is passed as

```
ID=
```

with no value following the ID. This will result in an "Automation Error" when you call the `ColumnHistory` function. We trap for this error in the `ShowColumnHistory` function so that no error is displayed when you move to the new record.

Getting a List of Attachments in an Attachment Field

The Attachment field data type is a new data type introduced with Access 12. The attachments field enables you to store zero or more file attachments in a field in a table.

An Attachment field is a multi-valued lookup (MVF) field (also commonly referred to as a complex field). An MVF field can store multiple items in a single field; in this case, it stores multiple files.

There are some special considerations when working with an Attachment data type in DAO. Because there are multiple items stored in the field, you cannot just reference the `.Value` property of the field. Instead, the `.Value` property for a complex field returns a `Recordset2` object that holds one row for each item in the field. If there are three attachments, there will be three rows in the recordset; if there are nine

Part II: Data Manipulation

attachments, there will be nine rows in the recordset. If there are no attachments, then the recordset will be empty, but there will be a recordset. The `.Value` property will not be `Null`, nor will it be `Nothing`. Both the `BOF` and `EOF` properties of the recordset will be `True` if there are no attachments, and the `RecordCount` property will return 0.

The following code is the typical pattern when working with MVF fields:

```
Dim db As DAO.Database
Dim rsOuter As DAO.Recordset2
Dim rsComplex As DAO.Recordset2
Set db = CurrentDb()
Set rsOuter = db.OpenRecordset("TableWithMvfData")
```

You will need two recordsets when working with MVF fields. The first recordset will contain the data in the table or query. The second recordset is used to access the individual items stored in the multi-valued lookup field. If your data source contains more than one MVF field, you may need additional recordsets (for example, an Issues table with an MVF field storing who the issue is assigned to, and an MVF field storing file attachments related to the issue).

```
'// Enumerate each record in the table
Do Until rsOuter.EOF
    '// Get the complex data recordset
    Set rsComplex = rsOuter("MVFField").Value
```

You loop through the outer recordset just as you would expect. Within the loop, you set the MVF recordset (`rsComplex`) to the `Value` property of the MVF field.

```
'// Enumerate the items in the MVF field
Do Until rsComplex.EOF
    '// Do something with the complex item
    Debug.Print rsComplex(0)
    rsComplex.MoveNext
Loop
rsComplex.Close
Set rsComplex = Nothing
rsOuter.MoveNext
Loop
rsOuter.Close
Set rsOuter = Nothing
...
```

The following function returns a semicolon delimited string of names of each attachment in an attachment field. It accepts the name of a table, the name of an attachment field, and the record number we want to interrogate.

```
Public Function GetAttachmentList( _
    tableName As String, fieldName As String, _
    rowid As Long) As String
    Const ATT_FIELDNAME_FILENAME As String = "FileName"
    Const CH_DELIM As String = ";"
    Dim db As DAO.Database
    Dim rs As DAO.Recordset2
```

```
Dim rsAtt As DAO.Recordset2
Dim strItems As String

Set db = CurrentDb
Set rs = db.OpenRecordset(tableName, dbOpenDynaset)
```

Following the pattern we just discussed, this function sets up two recordset variables, one (rs) to enumerate the dataset, and one (rsAtt) to hold the list of attachments in the attachments field. Here, we declare the variables, and open the dataset, using the `tableName` passed in the function argument list.

```
If Not rs(fieldName).IsComplex Then
    MsgBox fieldName & _
        " is not a Multi-valued lookup field!"
    Goto Cleanup:
End If
If Not rs(fieldName).Type = _
    DataTypeEnum.dbAttachment Then
    MsgBox fieldName & " is not an Attachment field!"
    Goto Cleanup:
End If
```

Here, we perform two tests to validate the arguments passed to the function. We want to make sure that the field passed is actually an Attachment field. The first test uses the `.IsComplex` property of a `Field2` object. This property will be true if the field is an MVF field. We include this test to demonstrate the use of the `.IsComplex` property. By itself, testing `IsComplex` is not sufficient because it does not determine whether or not the field is an Attachment field. A better test (and one that does not need the `IsComplex` test) is to check the `Type` property of the file. If the data type of the field is not `dbAttachment`, then we display an error message and proceed to the cleanup block.

```
rs.FindFirst "ID=" & rowId

If Not rs.NoMatch Then
    Set rsAtt = rs(fieldName).Value
```

We use the `FindFirst` method to locate the requested row. The `NoMatch` property will be `True` if no row is located. Because we want to enter the `If` loop only if a match *is* found, we use the `Not` operator to reverse the logic. If a match is found, we enter the loop and assign the `Value` property of the attachment field to the `rsAtt` recordset variable.

```
Do Until rsAtt.EOF
    strItems = strItems & _
        rsAtt(ATT_FIELDNAME_FILENAME) & CH_DELIM
    rsAtt.MoveNext
Loop
rsAtt.Close
Set rsAtt = Nothing
End If
```

If there are no attachments in the attachment field, there will still be a recordset in the `Value` property, but the recordset will be empty, and the `EOF` property will be true, so the loop will be skipped. If there are items in the field, there will be one row in the `rsAtt` recordset for each attachment in the field. We

Part II: Data Manipulation

loop through each row in the recordset, concatenating the value of the `FileName` field of the recordset to our `strItems` string. Once we have reached the end of the recordset, we close and release it.

```
Cleanup:  
    On Error Resume Next  
    rs.Close  
    Set rs = Nothing  
    If Right$(strItems, Len(CH_DELIM)) = CH_DELIM Then  
        strItems = Left$(strItems, Len(strItems) _  
            - Len(CH_DELIM))  
    End If
```

Our cleanup block closes the data recordset and then releases it. Finally, we trim the delimiter that exists on the end of the string.

```
Terminate:  
    GetAttachmentList = strItems  
    On Error GoTo 0  
    Exit Function  
  
End Function
```

The terminate block sets the return value of the function, and sets error handling to the default behavior.

You can test this function using the ch07-Attachments.accdb file in the sample files. Open this database, and then bring up the Immediate Window by pressing **Ctrl+G**. In the Immediate window, enter the following:

```
? GetAttachmentList("tblAttachments", "Attachment", 1)
```

The first row of the `tblAttachments` table contains five attachments. The function returns the following string:

```
candidate.png;empty.png;file1.txt;file2.txt;file3.txt
```

If you open the table and double-click on the `Attachment` field in the first row, you will see the same five files listed.

The second row of the `tblAttachments` table does not contain any attachments. Try stepping through the function again, this time passing 2 for the `rowId` argument. Notice that `rsAtt` is a valid recordset, although it contains no records.

Searching for Records with Attachments

To identify records that have attachments, you can test the number of rows in the complex data recordset. You merely need to test whether the `RecordCount` property of the complex recordset is greater than zero. The following function returns true if the `MVF` field that is passed as an argument contains any items.

```
Public Function HasMVFItems(mvfField As Field2) As Boolean  
    If mvfField IsComplex Then  
        HasMVFItems = CLng((mvfField.Value.RecordCount) _  
            > 0)
```

```
    Else
        '// Not a complex field so it can't have any items
        HasMVFItems = False
    End If
End Function
```

Because we don't need to actually work with the data in the complex recordset, we don't need to assign the `.Value` property of the MVF field to a recordset variable. We can directly query the `RecordCount` property of the `.Value` property.

We test the field passed as an argument to determine whether or not it is a Complex field. If we don't do this and we accidentally pass a non-complex field, the attempt to reference the `RecordCount` property will fail. Because a non-complex field cannot contain multiple items, we return `False` if the field isn't a complex field.

We can create a second helper function to return the number of items in an MVF field. To determine the number of items in an MVF field, move to the last record in the complex item recordset to ensure the `RecordCount` property is accurate, and then return the value of the `RecordCount` property. We move the recordset back to the first record before returning, so that the current record pointer sits at the top of the recordset.

```
Public Function MVFItemCount (mvfField As Field2) As Long

    If mvfField.IsComplex Then
        With mvfField.Value
            If .RecordCount > 0 Then
                .MoveLast
                .MoveFirst
            End If
            MVFItemCount = .RecordCount
        End With
    Else
        MVFItemCount = 0
    End If

End Function
```

Again, we test the field passed in the argument to ensure it is a complex field, and again, we do not need to assign the `Value` property to a recordset variable because we don't need to work with the actual data in the recordset and can directly reference the properties from the `Value` property. If there are any records in the `Value` property's recordset, we do the `MoveLast/MoveFirst` dance to ensure the `RecordCount` property reports the correct number of records, and then we return the value of the `RecordCount` property. We need an `Else` block so that we report 0 if we pass a non-MVF field.

The following function demonstrates how to call the `HasMVFItems` and the `MVFItemCount` functions. It prints the record number and whether or not the record has any attachments, and how many.

```
Public Sub HasItemsHarness(tableName As String, _
    fieldName As String)

    Dim db As DAO.Database
```

Part II: Data Manipulation

```
Dim rs As DAO.Recordset2

Set db = CurrentDb()
Set rs = db.OpenRecordset(tableName, dbOpenDynaset)

Do Until rs.EOF
    Debug.Print rs(0), _
        HasMVFItems(rs(fieldName)), _
        MVFItemCount(rs(fieldName))
    rs.MoveNext
Loop

Cleanup:
rs.Close
Set rs = Nothing
Set db = Nothing

End Sub
```

Searching for Specific Attachments

We can query an Attachment field in a specific row to find out whether it contains a specific attachment. We do this by looping through each entry in the complex recordset, comparing each `FileName` property to the name of the specific attachment we are interested in.

```
Public Function HasSpecificAttachment(mvfField As Field2, _
    fileName As String) As Boolean
Const ATT_FIELDNAME_FILENAME As String = "FileName"
Dim bResult As Boolean
Dim rsAtt As DAO.Recordset2

If DataTypeEnum.dbAttachment <> mvfField.Type Then
    MsgBox mvfField.Name & _
        " is not an Attachment field!"
    GoTo Terminate
End If
```

Because we will be attempting to read the `FileName` property of the complex recordset, we must first ensure the complex field is an attachment field. Other types of complex fields do not have a `FileName` property, and we would receive an error if we tried to use the property.

```
Set rsAtt = mvfField.Value
Do While Not rsAtt.EOF
    If rsAtt(ATT_FIELDNAME_FILENAME) = fileName Then
        bResult = True
        Exit Do
    End If
    rsAtt.MoveNext
Loop
```

We loop through each record in the complex recordset, comparing the actual filename to the specified filename. If they match, we set the result to True and break out of the loop. Otherwise, we move on to the next record.

```
Terminate:  
    HasSpecificAttachment = bResult  
  
End Function
```

Our terminate block simply returns a Boolean value indicating whether or not the specified attachment exists in the field.

We can use this function to search all records in a data table or query to find the record or records that have an attachment we are interested in. The following function searches through a specific attachment field in each record of a table or query. It returns a semicolon-delimited list of records that contains the specified attachment.

```
Public Function FindAttachment( _  
    tableName As String, attachmentFieldName As String, _  
    attachmentName As String) As String  
Const CH_DELIM As String = ";"  
  
Dim db As DAO.Database  
Dim rsOuter As DAO.Recordset2  
Dim rsComplex As DAO.Recordset2  
Dim fldComplex As DAO.Field2  
  
Dim strReturn As String  
  
Set db = CurrentDb()  
Set rsOuter = _  
    db.OpenRecordset(tableName, dbOpenDynaset)  
  
'// Test for Attachment field  
If DataTypeEnum.dbAttachment <> rsOuter(attachmentFieldName).Type Then  
    MsgBox attachmentFieldName & _  
        " is not an Attachment field!"  
    GoTo Cleanup  
End If
```

All of the code should be familiar up to this point. We create a recordset, and validate the attachment field.

```
Set fldComplex = rsOuter(attachmentFieldName)
```

Here, we set a field variable to point to the attachment field in our data source. This is a performance optimization. We could also simply pass the expression `rsOuter(attachmentFieldName)` to the `HasSpecificAttachment` function, but by assigning the field variable, we need to look up the field information only once per call, instead of once per row.

```
Do While Not rsOuter.EOF  
    If HasSpecificAttachment(fldComplex, attachmentName) Then  
        strReturn = strReturn & rsOuter("ID").Value & CH_DELIM
```

Part II: Data Manipulation

```
    End If  
    rsOuter.MoveNext  
Loop
```

We loop through each row in the data source, calling the `HasSpecificAttachment` function for each row. If the row does contain the attachment we are interested in, `HasSpecificAttachment` will return `True`, and we will add the `ID` field of that row to our `strReturn` variable (this function assumes your data source contains a field named `ID`; you can modify this code to concatenate another field, or add an argument so that you can dynamically specify which field contains the information you want to show in the list).

```
Cleanup:  
    Set fldComplex = Nothing  
    rsOuter.Close  
    Set rsOuter = Nothing  
Terminate:  
    If Right$(strReturn, Len(CH_DELIM)) = CH_DELIM Then  
        strReturn = Left$(strReturn, Len(strReturn) - Len(CH_DELIM))  
    End If  
    FindAttachment = strReturn  
  
End Function
```

Our cleanup block closes and releases the recordset and field objects, and our terminate block cleans up and then returns the list of record IDs.

Summary

This chapter introduced several VBA features that allow you to manage your data quickly and easily. We discussed the following:

- ❑ How to use the `Seek` and `Find` methods to find data
- ❑ How to use the `Move` methods and related properties to position the cursor in a recordset
- ❑ How to use several techniques for categorizing and sorting data
- ❑ How to use MD5 and Mod-10 checksum functions in an external library to validate data
- ❑ How to work with `AppendOnly` and `Attachment` fields in Access 2007

In the next chapter, we put some of these techniques to work when we look at using VBA to add advanced features to Access forms.

Part III: Interacting with the Application

Chapter 8: Using Code to Add Advanced Functionality to Forms

Chapter 9: Using Code to Add Advanced Functionality to Reports

Chapter 10: Using Automation to Add Functionality

Chapter 11: Creating Dynamic Ribbon Customizations

8

Using Code to Add Advanced Functionality to Forms

Aside from the data itself, forms are another key part of an application. Many Access developers tend to prevent users interacting with the tables directly, which makes forms the primary mechanism for presentation for data in the application. Unlike reports, forms also provide users with the ability to update data, making them the primary means for interacting with the data as well.

Because forms represent the user interface of the application, what you can do with them is virtually unlimited (or at least within the amount of hours you can bill if you're a consultant). Regardless of the layout and presentation, however, the ultimate goal for creating forms is to create something for the user that is both useful and enjoyable. Because the requirements for users and the application itself can greatly vary from one application to the next, this is an open statement. Keeping issues of style aside, however, there are several form-related features found throughout database applications and this chapter focuses on those. More specifically, in this chapter you will:

- Work with events on forms to perform tasks such as validating data and moving a borderless form
- Work with events on controls to perform tasks such as working with list boxes to present usable interfaces, creating custom progress bars, and to validate data
- Learn how to create common types of forms such as menus for navigation, splash screens, and dashboards
- See different ways you can create navigation experiences for your users

Working with Form Events

Events are one of the key components of Windows programming, and Access forms take full advantage of them. Access forms provide events that enable you, as the developer, to manipulate both the data and the user interface (UI). This section discusses some of the different events you can use at the form level in Access 2007.

Part III: Interacting with the Application

How to Determine When Data Is Added

The two events on a form that are related to inserting data are `BeforeInsert` and `AfterInsert`. The `BeforeInsert` event fires when the user types the first character in a new record, and the `AfterInsert` event fires after a new record has been added to the form. As with the other `Before` events in Access, the `BeforeInsert` event handler includes a `Cancel` argument to cancel the event if needed.

Let's say that you are creating a form that tracks orders and to receive credit for placing the order, you want to know who placed the order. The `BeforeInsert` event can be used to fill a control with the logon name of the user as shown here:

```
Private Sub Form_BeforeInsert(Cancel As Integer)
    Me.txtPlacedBy = Environ("USERNAME")
End Sub
```

The `AfterInsert` event can be used to take an action after a new record has been added. For example, let's say that you sell books online and want to send an e-mail to customers who place orders with you as confirmation. Here's how you could do this using the `AfterInsert` event:

```
Private Sub Form_AfterInsert()
    ' send mail after an order is placed
    Dim strTo      As String
    Dim strSubject As String
    Dim strBody     As String

    strTo = Me.EmailAddress
    strSubject = "Order Confirmation - Order Number: " & Me.[Order ID]
    strBody = "Thank you for your order!%0a%0d" & _
              "Order Date: " & Me.[Order Date] & "%0a%0d" & _
              "Shipping Date: " & Me.[Shipped Date] & "%0a%0d" & _
              "Ship To: " & "%0a%0d" & _
              Me.[Ship Name] & "%0a%0d" & _
              Me.[Ship Address] & "%0a%0d" & _
              Me.[Ship City] & ", " & Me.[Ship State/Province] & " " & _
              Me.[Ship ZIP/Postal Code]

    SendEmail strTo, strSubject, strBody

End Sub
```

We're using the `SendEmail` routine defined in Chapter 2 to create the e-mail. Here is that code as a reminder:

```
Private Declare Function ShellExecute Lib "shell32.dll" Alias "ShellExecuteA" ( _
    ByVal hWnd As Long, _
    ByVal lpOperation As String, _
    ByVal lpFile As String, _
    ByVal lpParameters As String, _
    ByVal lpDirectory As String, _
    ByVal nShowCmd As Long) As Long
Private Const SW_SHOW As Long = 5
Sub SendEmail(strTo As String, strSubject As String, strBody As String)
    Dim rc As Long
    Dim strFile As String
```

```
' build the lpFile argument
strFile = "mailto:" & strTo
strFile = strFile & "?subject=" & strSubject
strFile = strFile & "&body=" & strBody

rc = ShellExecute(0, "open", strFile, "", "", SW_SHOW)

End Sub
```

How to Determine When Data Is Changed

The AfterUpdate event of a form fires when the value of any control in a record is changed. There are many scenarios in which you might use this event. Here are some of them:

- Creating a change history of data in a table
- Updating some other information on screen such as status or calculations
- Showing or hiding controls
- Synchronizing record values between databases
- Synchronizing records in a subform
- Writing the results of a query to a file

Let's say that by default you provide a read-only view of data in a form by setting the AllowAdditions and AllowEdits properties of the form to False. On the form, you might include a button that sets these properties to True to allow users to edit a record. Use the AfterUpdate event of the form as follows to lock the form after the user commits a change to the data.

```
Private Sub Form_AfterUpdate()

    Me.AllowEdits = False
    Me.AllowAdditions = True
End Sub
```

How to Determine When Data Is Deleted

Access fires one of three different events when records are deleted. The Delete event fires for each record that is being deleted. The BeforeDelConfirm event fires when Access is about to delete records and prompts you to do so. And the AfterDelConfirm event fires after the records have been deleted (or the deletion was canceled).

Of the three, the Delete event is perhaps the most useful. This event includes a Cancel argument so that it can be canceled based on some criteria. In addition, because it fires for each record that is being deleted, you can also use this event to mark records for deletion instead of actually deleting them. Many types of applications have requirements about how data is handled and deletions are an important aspect of such applications. Say that you have a table with a Yes/No field called MarkedForDeletion. The following code in the Delete event of a form shows you how to update this field and cancel the actual deletion:

```
Private Sub Form_Delete(Cancel As Integer)
    ' mark records for deletion
```

Part III: Interacting with the Application

```
Dim strSQL As String
strSQL = "UPDATE tblCustomersDeleteEvents SET MarkedForDeletion=True " & _
          "WHERE ID=" & Me.ID

        ' run the sql
CurrentDb.Execute strSQL

        ' cancel the delete
Cancel = True

        ' refresh
Me.Refresh
End Sub
```

Validating Form Data

One of the more important tasks you're likely to do in a form is to validate data. For this you need the `BeforeUpdate` event of the form. If you want to validate data at the control level use the `BeforeUpdate` event of the control instead.

Because validation may fail for multiple controls when using the `BeforeUpdate` event of the form, it might be helpful to notify the user of all places where validation has failed. This prevents them from seeing an error message on individual controls until validation succeeds. As with most things in Access, there are a number of ways you can do this. We like to give visual cues for the controls where validation has failed rather than a list in a message box so that the user doesn't have to write down the controls. We sometimes include error text in the control itself or in another text box that displays the errors.

Using our examples of Orders from earlier, create a new form based on the `Orders` table in the Northwind 2007 sample database. In order for an order entry to be valid in this form, it must meet the following criteria:

- The order date cannot be in the past.
- The shipped date must be greater than or equal to the order date.
- When the payment type is specified, the paid date must be specified.
- When the paid date is specified, the payment type must be specified.

To validate these criteria, use a function called `IsValidData`, as shown in the code that follows. This function returns a Boolean value that determines whether the data on the form is valid. You start out by assuming that all data is valid and initialize a flag to `True`. When you hit an invalid case, set it to `False`.

```
Private Function IsValidData() As Boolean
    Dim fValid As Boolean
    Dim iRule As Integer

    ' assume all is good
    fValid = True

    ' contains the validation data for the form
    ResetControlFormatting
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

```
' if the payment type has been set, make sure the paid date is set
If (Not IsNull(Me![Payment Type]) And IsNull(Me![Paid Date])) Then
    Me.[Paid Date].BackColor = vbYellow
    fValid = False
End If

' if the paid date is set, make sure the payment type is set
If (Not IsNull(Me![Paid Date]) And IsNull(Me![Payment Type])) Then
    Me.[Payment Type].BackColor = vbYellow
    fValid = False
End If

' make sure shipped date is > order date
If ([Shipped Date] < [Order Date]) Then
    Me.[Shipped Date].BackColor = vbYellow
    fValid = False
End If

' make sure order date is >= today's date
If ([Order Date] < Date) Then
    Me.[Order Date].BackColor = vbYellow
    fValid = False
End If

' return
IsValidData = fValid
End Function
```

If you encounter invalid criteria, set the `BackColor` property of the control to yellow using the `vbYellow` constant. Because controls may have this property set before validation, you need some way to reset the back color. Add the following routine to the form to reset this property for combo boxes and text boxes.

```
Private Sub ResetControlFormatting()
    Dim c As Control

    ' reset control backcolor
    For Each c In Me.Controls
        If (c.ControlType = acTextBox Or c.ControlType = acComboBox) Then
            c.BackColor = vbWhite
        End If
    Next
End Sub
```

Now, to prevent an update to the database with invalid data, add the following code to the `BeforeUpdate` event of the form. This code sets the `Cancel` parameter value of the event handler based on the `return` value of the `IsValidData` function.

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
    ' validate data in the form
    Cancel = Not IsValidData()
End Sub
```

Part III: Interacting with the Application

There may still be controls that have the yellow back color set if a user presses the escape key to undo their changes. Add the following code to the Undo event of the form to reset the BackColor property.

```
Private Sub Form_Undo(Cancel As Integer)
    ResetControlFormatting
End Sub
```

The Undo event fires when changes are undone on the form.

Suppressing Access Error Messages

When you're working with data in a form, several data-related errors can occur. For example, errors can occur if the user enters a duplicate primary key, or violates a validation rule, or hits a write conflict. An error may also occur if the user tries to delete records in a parent table and records cannot be deleted from a child table because cascade delete is not enforced. In many cases, the error messages provided by Access are long and difficult to read. Use the Error event of the form to provide your own error message and take action based on a particular error, such as setting focus back to a control.

Using the Northwind 2007 sample database, create a new form based on the Customers table. This table is related to the Orders table and the cascade delete option is not set for the relationship. If there are related records in the foreign table, you get an error if you try to delete records in the parent table. The following code shows you how to use the Error event of the form to provide your own error message for this scenario:

```
Private Sub Form_Error(DataErr As Integer, Response As Integer)
    Const ERR RELATED RECORDS As Long = 3200      ' Cannot delete related records
    Const ERR PRIMARYKEY As Long = 3022            ' Primary key violation
    Const ERR_VALIDATION_RULE As Long = 3317        ' Validation rule violation
    Const ERR_INVALID_INPUTMASK As Long = 2113      ' Invalid entry for input mask

    Dim stMsg As String
    Dim fHandled As Boolean

    ' assume we're handling the error
    fHandled = True

    Select Case DataErr
        Case ERR_INVALID_INPUTMASK
            stMsg = "Invalid input mask"
        Case ERR_PRIMARYKEY
            stMsg = "You have entered a duplicate primary key"

            ' put focus back in the field
            Me.ID.SetFocus
        Case ERR RELATED RECORDS
            stMsg = "The company [" & Me.Company & "] has related orders. " & _
                    "Please delete these orders prior to deleting the company."
        Case ERR_VALIDATION_RULE
            stMsg = "Invalid data has been entered"
        Case Else
            ' unknown - let Access handle it
            fHandled = False
    End Select
End Sub
```

```
End Select
' hide Access' error message and show our own
If (fHandled) Then
    Response = acDataErrContinue
    MsgBox stMsg, vbExclamation
End If
End Sub
```

Notice that we're handling several errors here as defined by the constants in the code. We're also using a flag called `fHandled` to indicate whether we are handling the error or if Access should handle it. When we handle a given error, we've set the `Response` argument of the event to `acDataErrContinue`. This tells Access not to display its error message.

How to Determine If a Modifier Key Is Pressed

There might be times when you want to process keystrokes on a form. For example, you might want to provide custom navigation using arrow keys or handle keystrokes that the `autokeys` macro does not handle. Because the `autokeys` macro enables you to specify modifier keys (Alt, Shift, and Control), you need some way to do this with a form as well.

Forms include an event called `KeyDown` that includes an argument called `Shift`. This argument is a bitmask that includes the different modifier keys. Because this is a bitmask, the following code tests for the different modifiers:

```
fIsShiftSet = ((Shift And acShiftMask) = acShiftMask)
fIsAltSet = ((Shift And acAltMask) = acAltMask)
fIsCtrlSet = ((Shift And acCtrlMask) = acCtrlMask)
```

Let's say that you want to enhance the selection semantics for a form in datasheet view to make it easier to copy data to the clipboard. The semantics to add are: Shift+Ctrl+Down Arrow to select an entire column, Shift+Right Arrow to select a row, Ctrl+Right Arrow to move to the last field, and Ctrl+Left Arrow to move to the first field. Add the following code to the `KeyDown` event of a form in datasheet view to enable these semantics:

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    If (KeyCode = vbKeyRight And ((Shift And acShiftMask) = acShiftMask)) Then
        ' select an entire row: Shift+Right
        Me.SelLeft = 1
        Me.SelWidth = Me.Recordset.Fields.Count ' # of fields in the datasheet
    ElseIf (KeyCode = vbKeyDown And ((Shift And acShiftMask) = acShiftMask) And _
            ((Shift And acCtrlMask) = acCtrlMask)) Then
        ' select an entire column: Shift+Control+Down
        Me.SelTop = 1
        Me.SelHeight = Me.Recordset.RecordCount ' Records in the datasheet
    ElseIf (KeyCode = vbKeyRight And ((Shift And acCtrlMask) = acCtrlMask)) Then
        ' move to the last field: Control+Right
        Me.SelLeft = 10
    ElseIf (KeyCode = vbKeyLeft And ((Shift And acCtrlMask) = acCtrlMask)) Then
        ' move to the first field: Control+Left
        Me.SelLeft = 1
    End If
End Sub
```

Part III: Interacting with the Application

Be sure to set the `KeyPreview` property of the form to Yes or this event will not fire.

Windows defines key codes for a given keyboard using what is called a virtual key code. These codes are defined in VBA in the `KeyCodeConstants` module. This module in VBA defines the `vbKeyDown` and `vbKeyRight` constants in the previous code.

The Difference Between KeyDown and KeyPress

You've probably noticed that there are two events in Access that appear to be similar in nature: `KeyDown` and `KeyPress`. While it's true that both events fire when the user presses a key on the keyboard, the similarity ends there. The `KeyPress` event only responds when ASCII keys are pressed on the keyboard, hence the name of the argument for its event handler: `KeyAscii`. The `KeyDown` event on the other hand is considerably more granular — it fires for any key pressed on the keyboard, including control keys, such as the arrow keys.

If you need to handle arrow keys or other control keys, use the `KeyDown` event. If you need to determine whether a modifier is pressed, use the `KeyDown` event. If you only need ASCII keys, use the `KeyPress` event.

The other difference between these two events is that the `KeyDown` event also enables you to suppress keystrokes by setting the `KeyCode` argument to 0.

Periodic Backups Using the Timer Event

Here's a little confession. When we first discovered the `Timer` event of a form, we just used it to provide cute little animations and to flash labels on the form. While we still use it to provide animations (more for progress bars these days), we've since discovered that it's far more useful than we originally realized.

It goes without saying that backing up data is pretty important. There may even be requirements for backing up data, depending upon the industry in which you're working. To make sure that data is always backed up, you might consider implementing a backup strategy into your application if you haven't already. The `Timer` event can be used here to create a backup of the data on a particular form to a file on a periodic basis. This is only a partial solution — it only works when the form is open, but over the course of a day it may help ensure that there is always a backup available.

Add the following code to the `Timer` event of a form. In this example, the data is backed up to the same file every time the event fires. Depending on your requirements, you might extend this to write data to a different file each time. This code creates a backup of data in a table called `Customers`. To run this code, you need to have a table with this name or replace the name of the table in the code. You also need to set the `TimerInterval` property of the form to an acceptable value in milliseconds. To back up data every hour, set the `TimerInterval` property to 60000.

```
Private Sub Form_Timer()
    ' do a periodic backup of records
    Dim stFile As String
    Dim rs      As DAO.Recordset2
    Dim fld     As DAO.Field2
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

```
Dim stData As String

' show status
Me.lblWait.Visible = True
DoEvents

' open the file name - latest data always available
stFile = CurrentProject.Path & "\CustomerBackup.txt"
Open stFile For Output As #1

' get the data
Set rs = CurrentDb().OpenRecordset("Customers")

While (Not rs.EOF)
    ' loop through fields and build the data string
    For Each fld In rs.Fields
        If (Not fld.IsComplex) Then
            stData = stData & fld.Value & "|"
        End If
    Next

    ' print
    Print #1, Left(stData, Len(stData) - 1)
    rs.MoveNext
    DoEvents
Wend

' cleanup
rs.Close
Set rs = Nothing
Set fld = Nothing

Close #1

' hide status
Me.lblWait.Visible = False
End Sub
```

When we first started using the `Timer` event, we ran into a rather interesting problem. Every time we were writing code, IntelliSense would appear and then suddenly go away. Each line of code we typed would turn red as if there were a syntax error. It turned out that there was a `Timer` event running on an open form. The running timer would close any open windows such as the IntelliSense window in the code module.

Moving a Form Without a Border

Depending on the type of look that you're after for your application, you may choose to create borderless forms. We use borderless forms for About dialog boxes described later in this chapter, but they are also useful for popup types of items, such as calendars and calculators. Without a border, however, it can be difficult to move the form! Fortunately for us, however, Access provides mouse events for forms and controls that can be used to move a form on the screen.

Part III: Interacting with the Application

Start by creating a new form and set the `Border Style` property of the form to `None`. Next, add the following line of code to the declarations section of the form's module.

```
Private fMoving As Boolean
```

Then, add the following code to the `MouseDown` event of the `Detail` section to set the `fMoving` flag. This begins the move operation.

```
Private Sub Detail_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    fMoving = True
End Sub
```

Most of the work happens on the `MouseMove` event of the `Detail` section. Here, you're using the window dimensions of the form to determine where to move it. The calculation shown here is designed to prevent the mouse from jumping to the top-left hand corner of the form. Once you've calculated the new position, move the form by calling its `Move` method.

```
Private Sub Detail_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    Dim sngNewX As Single
    Dim sngNewY As Single

    If (fMoving) Then
        ' calculate using window dimensions so the mouse doesn't jump
        sngNewX = (Me.WindowLeft - Me.WindowWidth / 2) + X
        sngNewY = (Me.WindowTop - Me.WindowHeight / 2) + Y

        ' move the form
        Me.Move sngNewX, sngNewY
    End If
End Sub
```

Last, you need to stop the move operation when the mouse is released. To do this, add the following code to the `MouseUp` event of the `Detail` section. This resets the flag so that when the `MouseMove` event fires again, you don't move the form unless you're in a move operation.

```
Private Sub Detail_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    fMoving = False
End Sub
```

Run the form and click in the `Detail` section to begin moving the form.

Customizing ControlTipText

Control tips, or tooltips are used to provide informational text when you hover over an object, such as a control. In many cases, this property contains a static value, but wouldn't it be cool if it could contain actual data? We think it would, so let's take a look.

Chapter 8: Using Code to Add Advanced Functionality to Forms

Because the Current event of the form fires for every record, use this event to bind the ControlTipText property of a control based on values in the form:

```
Private Sub Form_Current()
    ' add controltips
    If (Not Me.NewRecord) Then
        Me.[Last Name].ControlTipText = Me.[First Name] & " " & _
            Me.[Last Name] & vbCrLf & _
            "Business: " & Nz(Me.[Business Phone]) & vbCrLf & _
            "Home: " & Nz(Me.[Home Phone]) & vbCrLf & _
            "Mobile: " & Nz(Me.[Mobile Phone])
    End If
End Sub
```

This might look something like the form shown in Figure 8-1.

A screenshot of a Microsoft Access form titled "frmEmployeesControlTips". The form has a title bar "frmEmployeesControlTips" and a main title "Employees". It contains several text input fields: "Last Name" (containing "Neipper"), "First Name" (containing "Michael"), "E-mail Address" (containing "michael@north"), "Job Title" (containing "Sales Representative"), "Address" (containing "123 6th Avenue"), "City" (containing "Redmond"), and "State/Province" (containing "WA"). A tooltip is displayed over the "Last Name" field, containing the text "Michael Neipper" followed by three lines of phone numbers: "Business: (123)555-0100", "Home: (123)555-0102", and "Mobile: ".

Last Name:	Neipper	Michael Neipper Business: (123)555-0100 Home: (123)555-0102 Mobile:	Address:	123 6th Avenue
First Name:	Michael		City:	Redmond
E-mail Address:	michael@north		State/Province:	WA
Job Title:	Sales Representative			

Figure 8-1

Working with Controls

We've taken a look at scenarios using form events, so it's time to move our attention to controls. In this section, we take a look at ways to validate data at the control level, as well as how to use list boxes in advanced scenarios that provide user interfaces that are intuitive. We also discuss creating custom progress bars and the new attachment control in Access 2007.

Validating Control Data

Early in this chapter, we looked at how you could use the BeforeUpdate event of a form to validate all the controls on a form. For some forms, this technique is good but it can make complex forms more difficult for the user. If you want to notify the user that there's a problem in a control without having to validate everything, you use the BeforeUpdate event of a control instead.

The following code shows you how to cancel the BeforeUpdate event based on some criteria:

```
Private Sub Taxes_BeforeUpdate(Cancel As Integer)
    ' no taxes for the following states:
    Const LIST_NO_SALESTAX As String = "AK|DE|MT|NH|OR"
    ' get the state for the customer
```

Part III: Interacting with the Application

```
Dim stState As String
stState = DLookup("State/Province", "Customers", "ID=" & Me.[Customer ID])

' verify tax is not applied for the defined states
If (Me.Taxes < 0) Then
    Cancel = True
    MsgBox "Taxes cannot be negative", vbExclamation
    Exit Sub
ElseIf (Me.Taxes > 0 And InStr(LIST_NO_SALESTAX, stState) > 0) Then
    Cancel = True
    MsgBox "Cannot apply tax for customer located in: " & stState, _
        vbExclamation
    Exit Sub
End If
End Sub
```

In this case, we're checking two things. First, we check that the amount of the tax is greater than 0 (this should probably be a validation rule in the table). Second, we're retrieving the state for the customer placing the order. If the state where the customer resides does not have a sales tax, then we cancel the event and alert the user.

Disabling all Controls

Rather frequently, we perform an action on all controls on a form. Typically, this is to apply common formatting, such as the BackColor property that you saw earlier, or to disable or enable controls on a form.

The following routine locks or unlocks all bound controls on a form. Notice that this routine includes error handling for the control types that don't support the ControlSource property.

```
Private Sub LockControls(fLock As Boolean)
    Dim c As Control

    For Each c In Me.Controls
        On Error Resume Next

        ' lock all bound controls
        If (Len(c.ControlSource) > 0) Then
            c.Locked = fLock
        End If

        On Error GoTo 0
    Next
End Sub
```

Reusing a Subform Control

Subform controls are really powerful. They are used to show related data on a form or report, but can take up a fair amount of space depending on what you put in them. For that reason, as well as for performance reasons, we like to reuse subform controls to display one subform at a time. Using subforms also enables you to create what we call a frame-based application where the main form is a frame used for navigation or dashboard and the content for the application is switched in a subform. An example of such an application is shown in Figure 8-2.

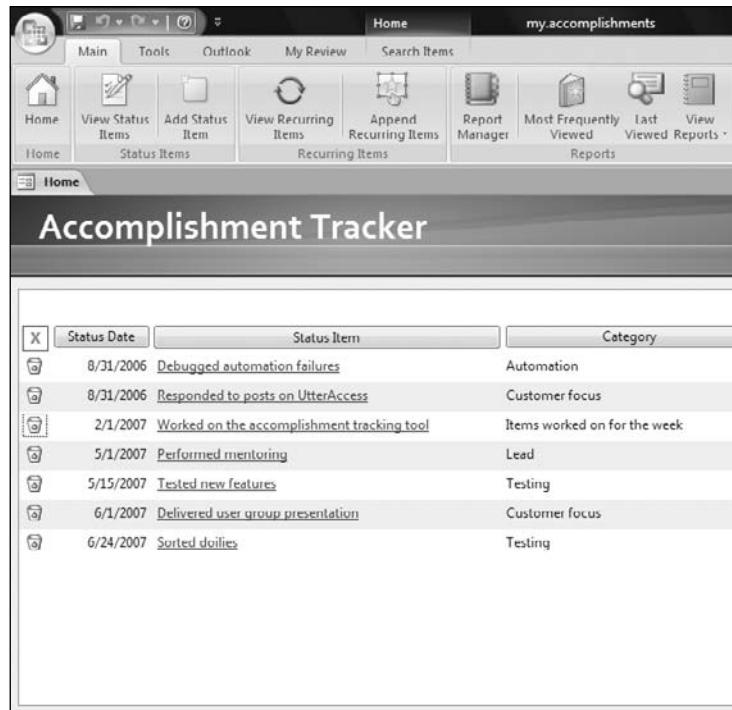


Figure 8-2

In this application, selecting one of the buttons in the Ribbon changes the subform shown in the bottom of the screen. To reuse a subform control, simply set the `SourceObject` property of the subform control to the name of a form in the database.

Extended List Box Functionality

List boxes are a staple of Access forms. As with combo boxes, they display data from a table or query, a list of values, or the field list from a table or query. In this section, we take a look at how you can extend them to create highly functional user interfaces.

Moving Items Up and Down in a List Box

In Chapter 7, we discuss using a field that defines a custom sort order for items in a list. A list box is a natural approach for displaying these items, but editing the sort order might be difficult. To help make this easier, we'll add up/down buttons to a form, enabling users to move items up and down using these buttons.

Create the Table

Start with a simple categories table, as shown in Figure 8-3. Notice that this table contains a field called `SortOrder` that defines the order in which items should appear in the list. Save the table as `tblCategories`.

Part III: Interacting with the Application

CategoryID	Category	SortOrder
1	Items worked on for the week	4
2	Mentoring	5
3	Customer focus	3
4	Automation	1
5	Lead	2
6	Testing	6
*	(New)	

Figure 8-3

Create the Form

Next, create a new form named `frmCategories` and set the `Pop Up` property of the form to `Yes`. Add a list box, a text box, and two command buttons to the form. Set the properties of the controls, as shown in the table that follows.

Control Name	Property Name	Property Value
List0	Column Count	3
	Column Widths	0" ; 1" ; 0"
	Row Source	SELECT CategoryID, Category, SortOrder FROM tblCategories ORDER BY SortOrder;
	Name	lstCategories
	ControlTip Text	Press {delete} to delete a category
Text2	Name	txtNewCategory
Command4	Caption	Up
	Name	cmdMoveUp
Command5	Caption	Down
	Name	cmdMoveDown

Arrange the form so that it resembles the one shown in Figure 8-4.

Let's start by adding code to the `Click` events of the two buttons. These event handlers call other routines that we define in a moment.

```
Private Sub cmdMoveDown_Click()
    MoveCategoryDown
End Sub
Private Sub cmdMoveUp_Click()
    MoveCategoryUp
End Sub
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

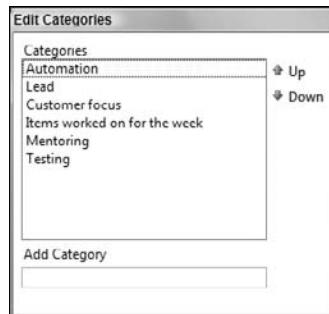


Figure 8-4

Because you're handling up and down behavior in the list, you should enable or disable the buttons depending on where you are in the list. Add the following code to the AfterUpdate event of the list box. This code disables the Up button when you're at the top of the list or the Down button when you're at the bottom of the list:

```
Private Sub lstCategories_AfterUpdate()
    DoEvents

    ' enable the move buttons if the first or last item has been set
    If (Me.lstCategories.ListIndex = Me.lstCategories.ListCount - 1) Then
        Me.cmdMoveUp.Enabled = True
        Me.cmdMoveUp.SetFocus
        Me.cmdMoveDown.Enabled = False
    Else
        Me.cmdMoveDown.Enabled = True
    End If

    If (Me.lstCategories.ListIndex = 0) Then
        Me.cmdMoveDown.Enabled = True
        Me.cmdMoveDown.SetFocus
        Me.cmdMoveUp.Enabled = False
    Else
        Me.cmdMoveUp.Enabled = True
    End If

    Me.lstCategories.SetFocus
End Sub
```

When you click the Up button, items move up in the list. This is accomplished by swapping the sort order for the current item and the item above it. Start by adding the `MoveCategoryUp` routine that is called from the `cmdMoveUp` button.

```
Private Sub MoveCategoryUp()
    Dim lOldOrder As Long
    Dim lNewOrder As Long
    Dim lSelRow As Long
```

Part III: Interacting with the Application

If there is a category selected, first determine the selected row:

```
' move the selected item up
If (Not (IsNull(Me.lstCategories))) Then
    ' Get the current selected row
    lSelRow = Me.lstCategories.ListIndex
```

Remember that the sort order is part of the row source for the list box so use the `Column` property to get the old sort order. The new sort order is stored in the same column but for the previous row.

```
If (Not IsNull(Me.lstCategories.Column(2)) And _
    Not IsNull(Me.lstCategories.Column(2, lSelRow + 1))) Then

    ' get the sortorder values for the current row and the next row
    lOldOrder = Me.lstCategories.Column(2)
    lNewOrder = Me.lstCategories.Column(2, lSelRow - 1)
```

To swap the two sort orders, call a routine called `SwapSortOrders` that you'll define in a moment.

```
' swap the sort order for the selected item and the one above it
SwapSortOrders lOldOrder, lNewOrder
```

Last, requery the list box so it has the current sort orders and refresh the button states by calling the `AfterUpdate` event handler for the list box.

```
' requery the listbox
Me.lstCategories.Requery

' update the buttons enabled state
lstCategories_AfterUpdate
End If
End If
End Sub
```

The `MoveCategoryDown` routine is very similar to `MoveCategoryUp` except that you want to get the sort order for the next item instead of the previous item.

```
Private Sub MoveCategoryDown()
    Dim lOldOrder As Long
    Dim lNewOrder As Long
    Dim lSelRow As Long

    ' move the selected item down
    If (Not (IsNull(Me.lstCategories))) Then
        ' Get the current selected row
        lSelRow = Me.lstCategories.ListIndex

        If (Not IsNull(Me.lstCategories.Column(2)) And _
            Not IsNull(Me.lstCategories.Column(2, lSelRow + 1))) Then

            ' get the sortorder values for the current row and the next row
            lOldOrder = Me.lstCategories.Column(2)
            lNewOrder = Me.lstCategories.Column(2, lSelRow + 1)
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

```
' swap the sort order for the selected item and the one above it
SwapSortOrders lOldOrder, lNewOrder

' requery the listbox
Me.lstCategories.Requery

' update the buttons enabled state
lstCategories_AfterUpdate
End If
End If
End Sub
```

To swap the sort orders, you need to update the `tblCategories` table. Start by defining the `SwapSortOrders` routine as follows:

```
Private Sub SwapSortOrders(lOldOrder As Long, lNewOrder As Long)
    Dim stSQL As String
    Dim lTemp As Long
```

This is a pretty straightforward swap operation. You first need to save the old sort order:

```
' cache item1
lTemp = lOldOrder
```

Next, set the old sort order to a dummy value:

```
' set the item1 sort order to -1 temporarily
stSQL = "UPDATE tblCategories SET SortOrder = -1 " & _
        "WHERE SortOrder = " & lNewOrder
CurrentProject.Connection.Execute stSQL
```

Then update the old sort order to the new sort order:

```
' set the new sort order to the old sort order
stSQL = "UPDATE tblCategories SET SortOrder = " & lNewOrder & _
        " WHERE SortOrder = " & lOldOrder
CurrentProject.Connection.Execute stSQL
```

And last, update the dummy value of -1 to the old sort order:

```
' final swap
stSQL = "UPDATE tblCategories SET SortOrder = " & lTemp & _
        " WHERE SortOrder = -1"
CurrentProject.Connection.Execute stSQL
End Sub
```

The last thing you want to do is to add a category to the list box using the `txtNewCategory` text box. If a category already exists, you don't want to add it again so use the `BeforeUpdate` event of the text box to make this determination:

```
Private Sub txtNewCategory_BeforeUpdate(Cancel As Integer)
    ' make sure the new category does not exist
    Dim i As Integer
```

Part III: Interacting with the Application

```
Dim stCat As String

' replace text
If (Not IsNull(Me.txtNewCategory)) Then
    stCat = Replace(Replace(Me.txtNewCategory, "''", "'''"), _
        Chr(34), Chr(34) & Chr(34))

    i = Nz(DCount("*", "tblCategories", "Category=''' & stCat & ''''), 0)

    If (i > 0) Then
        MsgBox "The category '" & Me.txtNewCategory & "' already exists. " & _
            "Please specify a new category name.", vbCritical
        Cancel = True
    End If
End If
End Sub
```

Notice that we've used the `DCount` function to determine whether the specified category already exists. If it does, cancel the event and alert the user.

If the category does not exist, add it to the underlying table with a sort order. To do this, use the `AfterUpdate` event of the control. Start by adding the event handler:

```
Private Sub txtNewCategory_AfterUpdate()
    ' add a new category
    Dim stSQL As String
    Dim stCat As String
    Dim lNewSortOrder As Long
```

Next, do some cleanup of the text:

```
If (Not IsNull(Me.txtNewCategory)) Then
    ' replace text
    stCat = Replace(Replace(Me.txtNewCategory, "''", "'''"), _
        Chr(34), Chr(34) & Chr(34))
```

We need to determine the new sort order based on the maximum sort order in the table and add one to that.

```
' get the new sort order (put the new item at the end)
lNewSortOrder = Nz(DMax("SortOrder", "tblCategories"), 0) + 1
```

Next, run a SQL statement to add the new category and refresh the controls.

```
' build and run the SQL
stSQL = "INSERT INTO tblCategories (Category, SortOrder) VALUES ('" & _
    stCat & "','" & lNewSortOrder & ")"
CurrentDb.Execute stSQL

' update the controls
Me.lstCategories.Requery
Me.txtNewCategory = Null
Me.txtNewCategory.SetFocus
End If
End Sub
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

Deleting Items in a List Box with the Delete Key

In the list box created in the previous example, we defined a tooltip that told the user you would handle deleting items in the list using the delete key, so let's do that now. Add the following code to the KeyUp event of the lstCategories list box to handle the delete key:

```
Private Sub lstCategories_KeyUp(KeyCode As Integer, Shift As Integer)
    ' trap deletes
    Select Case KeyCode
        Case vbKeyDelete
            DeleteCategory
    End Select
End Sub
```

Next, define the DeleteCategory routine:

```
Private Sub DeleteCategory()
    ' handle deletes
    Dim stCat As String
    Dim stSQL As String
    Dim stMsg As String
    Dim stSelected As String
    Dim lID As Long
    Dim lSelOrder As Long
```

Start by retrieving the category ID and name from the list box:

```
' get the id and selected item
lID = CLng(Me.lstCategories.ItemData(Me.lstCategories.ListIndex))
stSelected = Me.lstCategories.Column(1, Me.lstCategories.ListIndex)

' message
stMsg = "Are you sure you want to delete the category: " & stSelected
```

Deleting a category affects sort order. Get the sort order from the list box.

```
' existing sort order
lSelOrder = Me.lstCategories.ListIndex + 1
```

Prompt to delete, and if the user selects yes, then delete the category from the tblCategories table.

```
If (MsgBox(stMsg, vbExclamation + vbYesNo, "Delete Category") = vbYes) Then
    ' delete the category
    stSQL = "DELETE * FROM tblCategories WHERE CategoryID = " & lID
    CurrentDb.Execute stSQL
```

Then update all of the subsequent sort orders to the current order minus one.

```
' update the sort orders that follow
stSQL = "UPDATE tblCategories SET SortOrder = SortOrder - 1 " & _
        "WHERE SortOrder > " & lSelOrder
CurrentDb.Execute stSQL

' requery
```

Part III: Interacting with the Application

```
Me.lstCategories.Requery
    lstCategories_AfterUpdate
End If
End Sub
```

Using a Multi-Select List Box for Filter Criteria

Because list boxes provide multiple values, they make a pretty nice choice for a query form. However, the Value property for a multi-select list box is Null so you need to build the criteria dynamically.

Using the Northwind 2007 sample database, start by creating a new form and adding a list box and a subform control. The list box stores the customers in the Customers table and the subform stores the results of the filter. Set the properties of the list box as shown in the table that follows.

Property Name	Property Value
Column Count	2
Column Widths	0"
Row Source	SELECT ID, Company FROM Customers ORDER BY Company;
Name	lstCustomers
Multi Select	Extended

Set the properties of the subform control as follows:

Property Name	Property Value
Source Object	Table.Orders
Name	sfrmResults

Next, add two command buttons to the form named cmdFilter and cmdClearFilter. Add the following code to the cmdFilter button to create the filter for the subform.

```
Private Sub cmdFilter_Click()
    Dim stFilter As String
    Dim vItem      As Variant

    ' Build the filter string
    For Each vItem In Me.lstCustomers.ItemsSelected
        stFilter = stFilter & "[Customer ID] = " & _
                   Me.lstCustomers.ItemData(vItem) & " OR "
    Next

    ' remove the last ' OR '
    stFilter = Left(stFilter, Len(stFilter) - 4)
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

```
Me.sfrmResults.Form.Filter = stFilter  
Me.sfrmResults.Form.FilterOn = True  
End Sub
```

Add the following code to the cmdClearFilter button to remove the filter:

```
Private Sub cmdClearFilter_Click()  
    ' clear the filter  
    Me.sfrmResults.Form.Filter = ""  
    Me.sfrmResults.Form.FilterOn = False  
End Sub
```

Selecting All Items in a Multi-Select List Box Using the Keyboard

If we continue with the multi-select list box from the previous example, let's say that you want to use the CTRL+A keyboard shortcut to select all items in a list box. Add the following code to the KeyDown event of the lstCustomers list box from the previous example to do this.

```
Private Sub lstCustomers_KeyDown(KeyCode As Integer, Shift As Integer)  
    Dim lItem As Long  
  
    If (KeyCode = vbKeyA And ((Shift And acCtrlMask) = acCtrlMask)) Then  
        ' select all items in the list  
        For lItem = 0 To Me.lstCustomers.ListCount - 1  
            Me.lstCustomers.Selected(lItem) = True  
        Next  
    End If  
End Sub
```

This code walks through each item in the list box and sets its Selected property to True. When you select the list box, you should be able to press CTRL+A to select all items in the list box.

Custom Progress Bars

Way back when, the Office Developer Edition and Microsoft Office Developer released ActiveX versions of a progress bar control that you could use to provide feedback on a long running progress. ActiveX controls can simplify programming tasks but have deployment issues. As an alternative to an ActiveX control, the SysCmd function in Access can be used to display a progress bar in the status bar space at the bottom of the screen. This is pretty cool but what if the user chooses not to show his status bar or if you want to use that real estate? Unfortunately, the progress bar at the bottom of the screen sometimes goes unnoticed by users as well.

To solve these problems, we create lightweight progress bars using rectangle controls in Access. Visually we tend to prefer these controls for popup or modal forms that don't take up the entire screen and lean toward using the SysCmd function for full screen forms.

Creating this type of progress bar is pretty straight forward. It requires two rectangle controls that we call boxOutside and boxInside. As you can imagine from the names, boxInside is placed inside boxOutside. To update progress, the width of boxInside is adjusted as a percentage of the width of boxOutside. As with the progress bar created by the SysCmd function, this technique also requires that you know the number of items that you're measuring. Let's take a look at this in action.

Part III: Interacting with the Application

Create the Form

The form will be fairly simplistic as we want to focus on the progress bar itself. Create a new form with two rectangles and a command button. Name the rectangles `boxInside` and `boxOutside` as mentioned earlier. Set the width of `boxOutside` to 4" so you can see the progress. Set the width of `boxInside` to 0" to start. Make sure that `boxInside` is not transparent or you won't be able to see anything! Name the command button `cmdGo`.

Run the Process

Because file operations may take some time, we write to a file by hand as a long running process. The file only contains the values from the loop as an example. Add the following code to the `Click` event of the button:

```
Private Sub cmdGo_Click()
    ' create a file in the current directory
    Dim i          As Integer
    Dim stFile     As String
    Dim sngOffset  As Single
    Dim n          As Long

    stFile = CurrentProject.Path & "\NewFile.txt"

    ' open the file
    Open stFile For Output As #1

    ' number of items
    n = 5000

    For i = 1 To n
        ' print to the file
        Print #1, i

        ' update the progress - includes an offset calculation for aesthetics
        sngOffset = Me.boxInside.Left - Me.boxOutside.Left
        Me.boxInside.Width = i * ((Me.boxOutside.Width - (sngOffset * 2)) / n)
        Me.Repaint

        DoEvents
    Next

    ' close
    Close #1

    MsgBox "Done!"
End Sub
```

The algorithm shown here is one we use quite a bit and it includes an offset that bears some digging into. The rectangle `boxOutside` is formatted to have a border, whereas `boxInside` is not. To prevent `boxInside` from completely covering `boxOutside`, we tend to position the `Left` property of `boxInside` just barely to the right of `boxOutside` as shown in Figure 8-5, which has been zoomed to show the offset.

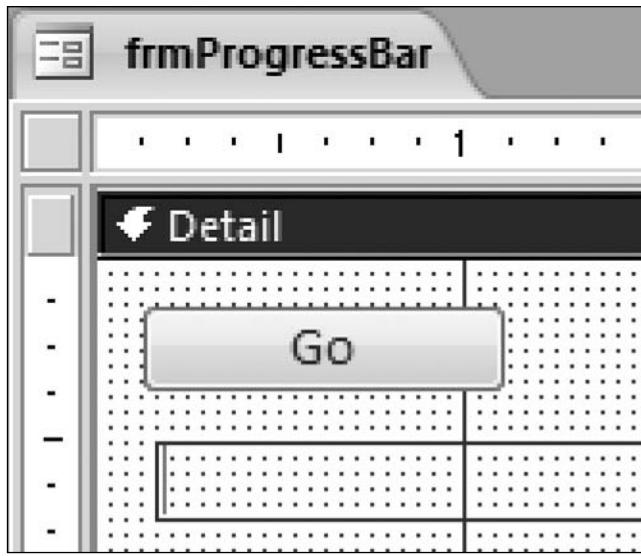


Figure 8-5

The offset calculation then is simply the difference in the `Left` property between `boxInside` and `boxOutside`. To keep this offset on both the left and right inside edges of `boxOutside`, multiply this by 2 when subtracting it from the width of `boxOutside`. Once the aesthetics are accounted for, divide the width of `boxOutside` by the number of items (`n`), and multiply by the current location in the loop (`i`). This dynamically determines the width of `boxInside`.

Custom Up/Down Buttons

Speaking of ActiveX controls, the toolkits mentioned in the previous section also had a pretty cool up/down control that users can click and hold a button to set values. This is pretty useful but it turns out that command buttons in Access let you do this without an ActiveX control.

Create a form with a text box and two command buttons named `cmdDown` and `cmdUp`, and a text box named `txtInterval`. Set the `Auto Repeat` property of the two command buttons to Yes. When this property is set, the `Click` event of a button fires repeatedly while the user is holding down the button. Add the following code to the two buttons. This code adjusts the value of the text box in increments of 1000. When scrolling up, it scrolls without a limit. When scrolling down, it scrolls until it reaches 0.

```
Private Sub cmdDown_Click()
    ' init
    Me.txtInterval = Nz(Me.txtInterval, 1000)

    ' decrement to 0
    If (CLng(Me.txtInterval) >= 1000) Then
        Me.txtInterval = CLng(Me.txtInterval) - 1000
    End If
End Sub
```

Part III: Interacting with the Application

```
End If  
  
DoEvents  
End Sub  
Private Sub cmdUp_Click()  
    ' init  
    Me.txtInterval = Nz(Me.txtInterval, 1000)  
  
    ' increment (unbounded)  
    Me.txtInterval = CLng(Me.txtInterval) + 1000  
    DoEvents  
End Sub
```

Displaying Multiple Attachments Onscreen

The Attachment control in Access 2007 is used to display data in the Attachment data type. This is a multi-valued field where multiple attachments can be stored in a single record. This data type is extremely useful and a big improvement over the OLE Object data type. Because it stores multiple files, however, navigation of attachments tends to happen within a single attachment control. For this example, let's say that you're developing a real estate application and want to display multiple attachments for a given property at once using a thumbnail. This is common in Web applications, such as www.realtor.com.

For this example, create a table called `tblRealEstate` with the fields shown in the following table.

Field Name	Data Type (Size)
ID	AutoNumber
Address	Text (255)
City	Text (255)
State	Text (255)
HousePictures	Attachment

Sample images that you can use for the attachment field are included with the sample available with this book on www.wrox.com.

Fill the table with address information and images in the attachment field, as shown in Figure 8-6.

Next, create a form bound to this table. Set the `Visible` property of the `HousePictures` attachment control to `No` to hide the control. Add three new unbound attachment controls named `Attachment1`, `Attachment2`, and `Attachment3` respectively. Set their height and width to `1.5"` each to give them the appearance of being thumbnails.

Chapter 8: Using Code to Add Advanced Functionality to Forms

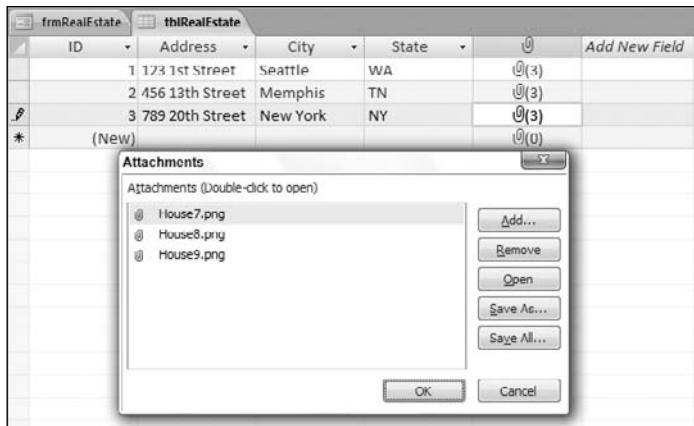


Figure 8-6

Time for the code, which is surprisingly simple. You'll remember that the `HousePictures` field stores the actual attachment files for each property in the `tblRealEstate` table. We know that we want to update the unbound attachment controls for each record so we need to use the `Current` event of the form. The `AttachmentCount` property of the attachment control returns the number of attachments in the control. The control also includes a property called `DefaultPicture` that is used to display an image if there are no attachments in the control. In our case, because we are using unbound controls, we can simply loop through the attachments in the `HousePictures` control to set the `DefaultPicture` property of the unbound controls, as shown in the following code. Navigation through the `HousePictures` field is done using the `Forward` method of the control.

```
Private Sub Form_Current()
    Dim i As Integer
    Dim att As Attachment

    ' bind the attachments
    For i = 0 To Me.HousePictures.AttachmentCount - 1
        Set att = Me.Controls("Attachment" & i + 1)
        att.DefaultPicture = Me.HousePictures.FileName(i)

        Me.HousePictures.Forward
    Next
End Sub
```

The graphics used in this sample are also used in the section “Kiosk Forms” later in this chapter. We originally created the sample kiosk form with these same images using unbound OLE Object controls as the result of a copy/paste from another application. After pasting all nine images, the size of the database was approximately 28MB. After adding them to the database as PNG files in an attachment field, the database shrunk to 1.75MB after compact!

Common Forms for Your Applications

Many applications have forms, ranging from the simplest to the complex. For example, a menu form may be used to provide navigation to other parts of the application, or a dashboard form might contain information to provide a summary view of the data in the database. In this section, we look at several types of forms that can be added to applications to provide a professional touch.

Dynamic Menu Forms and Dashboards

Menu forms and dashboards can be combined into a single form that provides both navigation and rich summary views. The main menu form can also be used for core functionality that you want to highlight in the application, such as a search feature. With so many pieces of data that can be presented on the form, how can you prevent information overload? One way is to create several different configurations for a dashboard and then allow users to select the one that they want. Different components that plug into the dashboard form can be selected to display in a particular place on the form. This enables users to configure the application to meet their needs and to see the pieces of data they are most interested in.

You can find out more about configuring applications in Chapter 12.

Defining the Dashboard Layouts

The first thing we do is define the different form layouts for the dashboard forms. Our dashboards are simple with no more than four or five components on the form. Each dashboard is a separate form that hosts the components. Components on the form are displayed using subforms, which provide flexibility and allow movement of objects without affecting the main form.

Each dashboard should have the same properties set so they have a consistent appearance. Start by creating four dashboard forms with the common properties shown in the table that follows.

Property Name	Property Value
Caption	Dashboard
Allow Datasheet View	No
Allow PivotTable View	No
Allow PivotChart View	No
Record Selectors	No
Navigation Buttons	No
Scroll Bars	Neither

Save the forms with the name USysFrmDashboard1, USysFrmDashboard2, USysFrmDashboard3, and USysFrmDashboard4.

Chapter 8: Using Code to Add Advanced Functionality to Forms

These forms are included with the sample code available for download with this book from www.wrox.com. The sample file for this section is called Dashboards.accdb. The data used in the example is from the Northwind 2007 sample database.

We're also going to take a screen shot of each form once the controls have been added to act as a preview for users to choose. More on this later in this section.

USysFrmDashboard1

The design of dashboard 1 is similar to that of a Web page with a large navigation frame on the left side of the page and a top and bottom frame on the right. Add three subforms to the form and delete their associated labels. Set properties on the subforms as shown in the table that follows.

Control Name	Property Name	Property Value
Child0	Width	1.9167"
	Height	4.7083"
	Top	0.0417"
	Left	0.0833"
	Border Style	Solid
	Horizontal Anchor	Left
	Vertical Anchor	Both
	Name	sfrmLeft
	Width	5.375"
	Height	2.4583"
Child1	Top	0.0417"
	Left	2.0833"
	Border Style	Solid
	Horizontal Anchor	Both
	Vertical Anchor	Top
	Name	sfrmTop
	Width	5.375"
	Height	2.2083"
Child2		

Continued on next page

Part III: Interacting with the Application

Control Name	Property Name	Property Value
	Top	2.5417"
	Left	2.0833"
	Border Style	Solid
	Horizontal Anchor	Both
	Vertical Anchor	Both
	Name	sfrmBottom

After setting the properties, you should have a form that resembles the one shown in Figure 8-7.

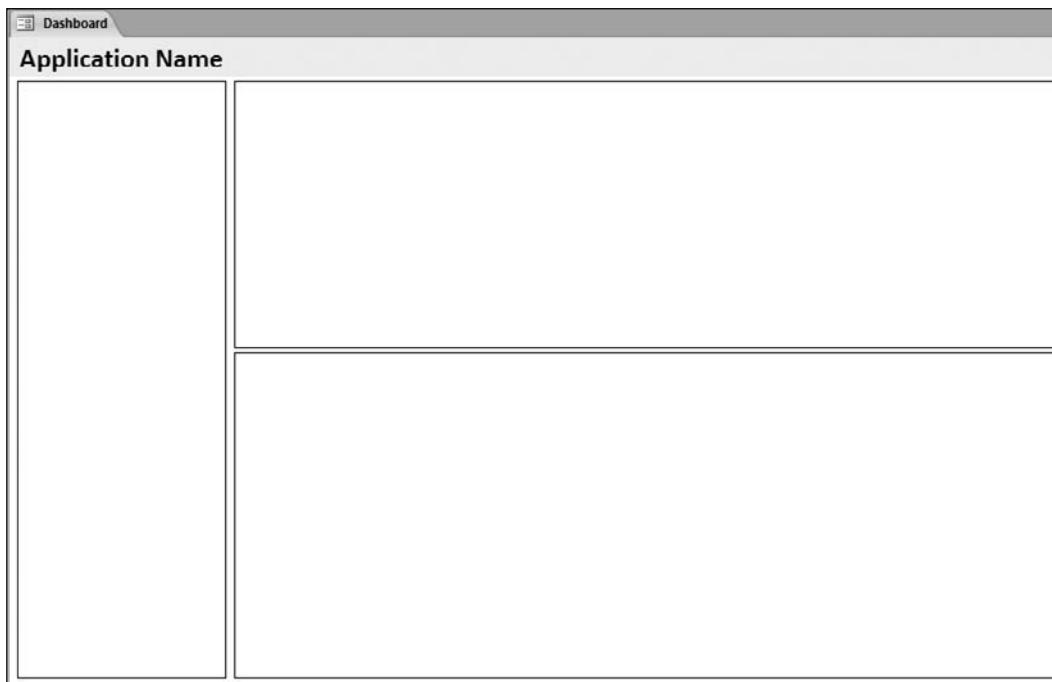


Figure 8-7

Open the form in form view and take a screenshot of the form. Using a graphics program such as Microsoft Paint, save the screenshot to the directory where the database resides in the PNG format as `dashboard1.png`.

Chapter 8: Using Code to Add Advanced Functionality to Forms

USysFrmDashboard2

The design of dashboard 2 is similar to that of dashboard 1 with the exception that it includes a subform on the far right. The subforms on the top and bottom right of dashboard 1 appear in the middle of dashboard 2.

Add four subforms to the form and delete their associated labels. Set properties on the subforms as shown in the table that follows.

Control Name	Property Name	Property Value
Child0	Width	1.9167"
	Height	4.7083"
	Top	0.0417"
	Left	0.0833"
	Border Style	Solid
	Horizontal Anchor	Left
	Vertical Anchor	Both
	Name	sfrmLeft
	Width	3.9167"
	Height	2.4583"
Child1	Top	0.0417"
	Left	2.0833"
	Border Style	Solid
	Horizontal Anchor	Both
	Vertical Anchor	Top
	Name	sfrmTop
	Width	3.9167"
	Height	2.2083"
	Top	2.5417"
	Left	2.0833"
Child2	Width	3.9167"
	Height	2.2083"
	Top	2.5417"
	Left	2.0833"

Continued on next page

Part III: Interacting with the Application

Control Name	Property Name	Property Value
Child3	Border Style	Solid
	Horizontal Anchor	Both
	Vertical Anchor	Both
	Name	sfrmBottom
	Width	1.9167"
	Height	4.7083"
	Top	0.0417"
	Left	6.0833"
	Border Style	Solid
	Horizontal Anchor	Right
Child4	Vertical Anchor	Both
	Name	sfrmRight

After setting the properties, you should have a form that looks something like the one shown in Figure 8-8.

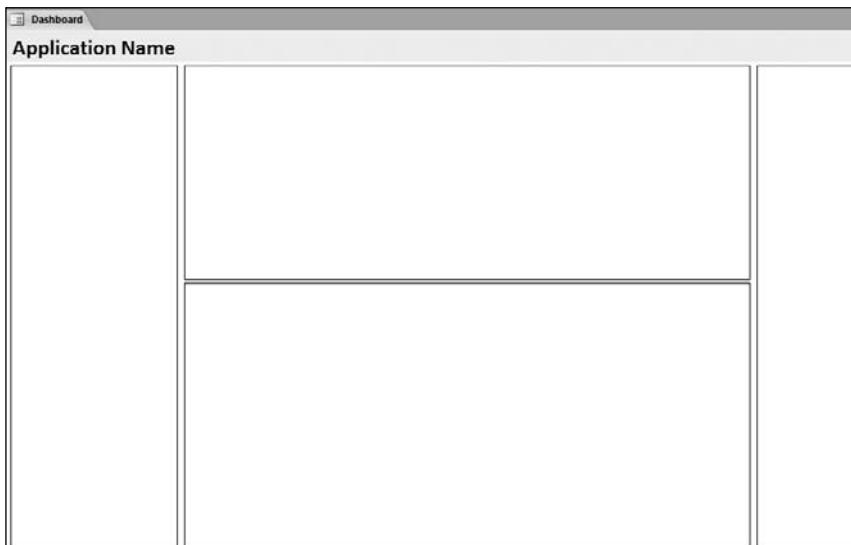


Figure 8-8

Chapter 8: Using Code to Add Advanced Functionality to Forms

Open the form in form view and take a screenshot of the form. Using a graphics program such as Microsoft Paint, save the screenshot to the directory where the database resides in the PNG format as dashboard2.png.

USysFrmDashboard3

The design of dashboard 3 consists of a navigation subform on the left side and then four equally sized subforms on the right. This is a classic dashboard form that you might use to display multiple charts on screen.

Add five subforms to the form and delete their associated labels. Set properties on the subforms as shown in the following table.

Control Name	Property Name	Property Value
Child0	Width	1.9167"
	Height	4.7083"
	Top	0.0417"
	Left	0.0833"
	Border Style	Solid
	Horizontal Anchor	Left
	Vertical Anchor	Both
	Name	sfrmLeft
	Width	3.5"
	Height	2.5"
Child1	Top	0.0417"
	Left	2.0833"
	Border Style	Solid
	Horizontal Anchor	Left
	Vertical Anchor	Top
	Name	sfrmTopLeft
	Width	3.5"
Child2		

Continued on next page

Part III: Interacting with the Application

Control Name	Property Name	Property Value
Child3	Height	2.5"
	Top	0.0417"
	Left	5.625"
	Border Style	Solid
	Horizontal Anchor	Left
	Vertical Anchor	Top
	Name	sfrmTopRight
	Width	3.5"
	Height	2.5"
	Top	2.5833"
Child4	Left	2.0833"
	Border Style	Solid
	Horizontal Anchor	Left
	Vertical Anchor	Top
	Name	sfrmBottomLeft
	Width	3.5"
	Height	2.5"
	Top	2.5833"
	Left	5.625"
	Border Style	Solid
Child5	Horizontal Anchor	Left
	Vertical Anchor	Top
Child6	Name	sfrmBottomRight
	Width	3.5"

After setting the properties, you should have a form that resembles the one shown in Figure 8-9.

Chapter 8: Using Code to Add Advanced Functionality to Forms

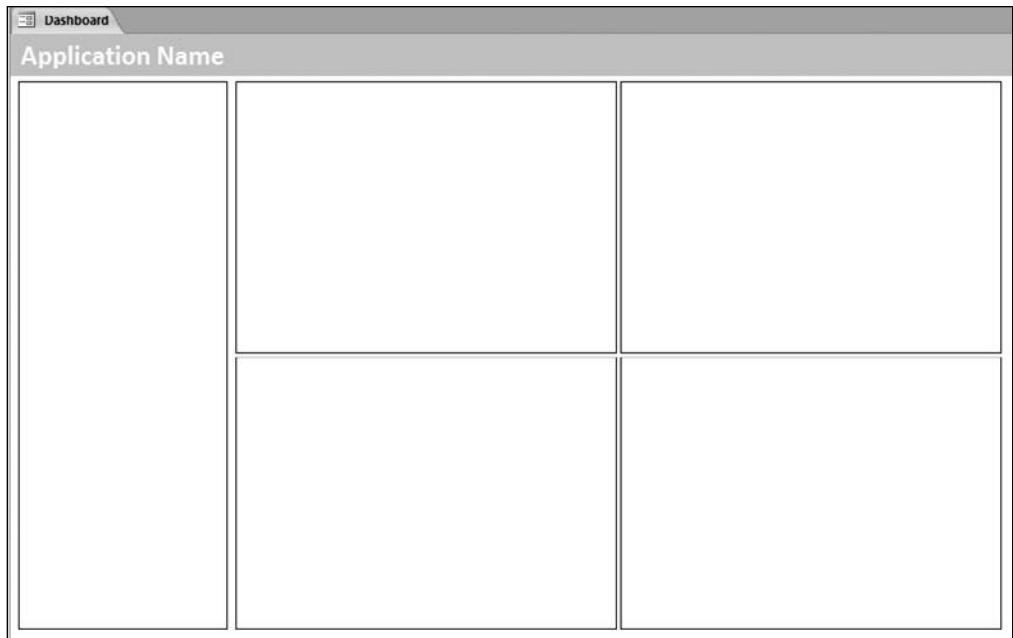


Figure 8-9

Open the form in form view and take a screenshot of the form. Using a graphics program such as Microsoft Paint, save the screenshot to the directory where the database resides in the PNG format as dashboard3.png.

USysFrmDashboard4

The design of dashboard 4 consists of a subform on the top similar to a banner on a Web page and two subforms on the bottom.

Add three subforms to the form and delete their associated labels. Set properties on the subforms as shown in the following table.

Control Name	Property Name	Property Value
Child0	Width	8.9167"
	Height	0.9583"
	Top	0.0417"
	Left	0.0833"
	Border Style	Solid

Continued on next page

Part III: Interacting with the Application

Control Name	Property Name	Property Value
Child1	Horizontal Anchor	Both
	Vertical Anchor	Top
	Name	sfrmTop
	Width	2.916"
	Height	2.5"
	Top	1.0417"
	Left	0.0833"
	Border Style	Solid
	Horizontal Anchor	Left
	Vertical Anchor	Both
Child2	Name	sfrmLeft
	Width	5.916"
	Height	2.5"
	Top	1.0417"
	Left	3.0833"
	Border Style	Solid
	Horizontal Anchor	Both
	Vertical Anchor	Both
	Name	sfrmRight

After setting the properties, you should have a form that resembles the one shown in Figure 8-10.

Open the form in form view and take a screenshot of the form. Using a graphics program such as Microsoft Paint, save the screenshot to the directory where the database resides in the PNG format as dashboard4.png.

Creating the Tables

To store information about the different components that can be used in a dashboard, we need some tables. The tables store the names of objects that can be used as components, the different dashboard layouts with a preview of the dashboard, and then a junction table that determines where the components are placed in a dashboard.

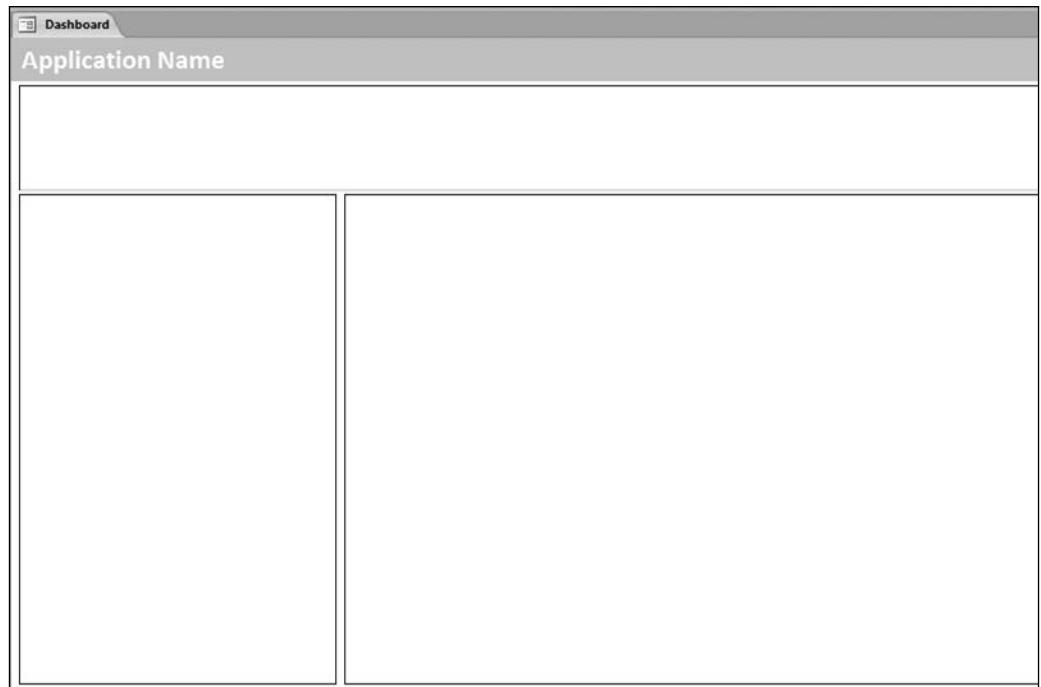


Figure 8-10

Create the Dashboard Table

The dashboard table stores information about the dashboards in the application. Create a new table with the following fields. Save the table with the name USysDashboards after you've created the fields. We're using the USys prefix to hide our tables in the Access navigation pane by default.

Field Name	Data Type (Size)
DashboardID	AutoNumber (Primary Key)
DashboardName	Text (255)
DashboardForm	Text (64)
DashboardDescription	Text (255)
DashboardPreview	Attachment
IsSelected	Yes/No

Part III: Interacting with the Application

After you save the table, fill it with data as shown in Figure 8-11. Use the screenshots created in the section “Defining the Dashboard Layouts” as the attachment in the DashboardPreview field.

USysDashboards						
DashboardID	DashboardName	DashboardForm	DashboardDescription	①	②(1)	IsSelected
1	Simple	USysFrmDashboard1	Simple - navigation on left + 2 components	③(1)	<input type="checkbox"/>	
2	Extended	USysFrmDashboard2	Extended - navigation on left and right + two compo	④(1)	<input checked="" type="checkbox"/>	
3	Classic	USysFrmDashboard3	Classic - navigation on left + 4 components	⑤(1)	<input type="checkbox"/>	
4	Banner	USysFrmDashboard4	Simple - top banner + 2 components	⑥(1)	<input type="checkbox"/>	
*	(New)			⑦(0)	<input type="checkbox"/>	

Figure 8-11

Create the Component Table

The component table stores information about the components that can be displayed in a dashboard. Create a new table with the following fields. Save the table with the name USysComponents after you've created the fields.

Field Name	Data Type (Size)
ComponentID	AutoNumber (Primary Key)
ComponentName	Text (255)
ComponentObject	Text (64)

Fill the table with the names of forms in your application in the ComponentObject field.

Create the Dashboard Components Table

The dashboard components table stores data about the components that appear in a given dashboard. The maximum number of components that can appear in the dashboards is fixed so we're actually going to pre-populate this table with the names of the subforms that appear on the dashboard forms. Begin by creating a new table with the following fields. Save the table with the name USysDashboardComponents when you're done.

Field Name	Data Type (Size)
DashboardComponentID	AutoNumber (Primary Key)
ComponentID	Number (Long Integer)
DashboardID	Number (Long Integer)
DashboardControl	Text

Chapter 8: Using Code to Add Advanced Functionality to Forms

In most cases, you would probably store only the components for the currently selected dashboard, which at most would be five in the case of USysFrmDashboard3. However, we're going to pre-populate this table for several reasons:

- ❑ The total number of subforms is fixed at 15.
- ❑ Switching dashboards would require deleting records. Because the Access database engine doesn't reclaim space until the database is compacted — the potential is there for unnecessary growth even though it may be minimal.
- ❑ Pre-populating the table allows the user to switch between dashboards and preserve their layout each time.

Add the name of each subform control in the DashboardControl field along with the appropriate DashboardID value from the USysDashboards table. The table should look something like the one shown in Figure 8-12 when you're done.

DashboardComponentID	ComponentID	DashboardID	DashboardControl
1			1 sfrmLeft
2			1 sfrmTop
3			1 sfrmBottom
4			2 sfrmLeft
5			2 sfrmTop
6			2 sfrmBottom
7			2 sfrmRight
8			3 sfrmLeft
9			3 sfrmTopLeft
10			3 sfrmTopRight
11			3 sfrmBottomLeft
12			3 sfrmBottomRight
13			4 sfrmTop
14			4 sfrmLeft
15			4 sfrmRight
*	(New)		

Figure 8-12

Creating Relationships for the Tables

With the tables created and filled, we need to create some relationships. View the Relationships diagram by selecting the Database Tools tab in the Ribbon, and then click the Relationships button. Add the USysDashboards, USysComponents, and USysDashboardComponents tables to the relationships window. Join the fields as shown in Figure 8-13. You'll notice that referential integrity is set on the relationships, as shown in Figure 8-13.

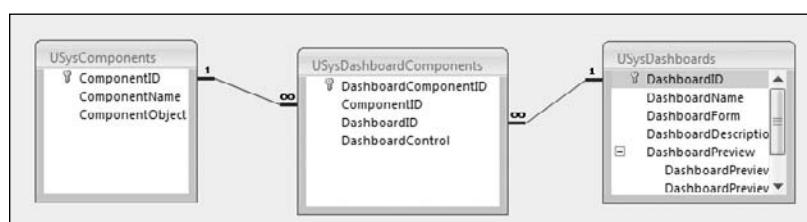


Figure 8-13

Part III: Interacting with the Application

Choosing a Dashboard

The currently selected dashboard is stored in the `IsSelected` field in the `USysDashboards` table. To set value and give the user a choice of dashboards, create a new form that is bound to the `USysDashboards` table called `USysFrmDashboards` as follows.

1. Create a new form in design view. Bind the form to the `USysDashboards` table. Save the form as `USysFrmDashboards`. Set the properties shown in the table that follows on the form.

Property Name	Property Value
Caption	Choose Dashboard
Allow Datasheet View	No
Allow PivotTable View	No
Allow PivotChart View	No
Allow Layout View	No
Width	4 .5 "
Border Style	Dialog
Record Selectors	No
Navigation Buttons	No
Record Source	<code>USysDashboards</code>
Allow Additions	No
Allow Deletions	No
Pop Up	Yes
Modal	Yes

2. Add a list box to the form with the properties shown in the table that follows.

Property Name	Property Value
Column Count	2
Column Widths	0 " ; 1 "
Width	1 "
Height	2 .9167 "

Chapter 8: Using Code to Add Advanced Functionality to Forms

Property Name	Property Value
Top	0.25 "
Left	0.0417 "
Row Source	SELECT DashboardID, DashboardName FROM USysDashboards ORDER BY DashboardName;
Name	1stDashboards

- 3.** Add an attachment control to the form with the properties shown in the table that follows.

Property Name	Property Value
Picture Size Mode	Stretch
Width	3.3333 "
Height	2.6667 "
Top	0.25 "
Left	1.0833 "
Control Source	DashboardPreview
Enabled	No
Locked	Yes
Name	DashboardPreview

- 4.** Add a text box to the form with the properties listed in the table that follows. Delete the associated label.

Property Name	Property Value
Width	3.3333 "
Height	0.2188 "
Top	2.9583 "
Left	1.0833 "
Control Source	DashboardDescription

Continued on next page

Part III: Interacting with the Application

Property Name	Property Value
Enabled	No
Locked	Yes
Name	DashboardDescription

5. Add a command button to the form with the properties listed in the table that follows.

Property Name	Property Value
Width	1 "
Height	0.25 "
Top	3.2083 "
Left	2.375 "
Caption	OK
Name	cmdOK
Default	Yes

6. Add a command button to the form with the properties listed in the table that follows.

Property Name	Property Value
Width	1 "
Height	0.25 "
Top	3.2083 "
Left	3.4167 "
Caption	Cancel
Name	cmdCancel
Cancel	Yes

The form should look something like the one shown in Figure 8-14.

Chapter 8: Using Code to Add Advanced Functionality to Forms

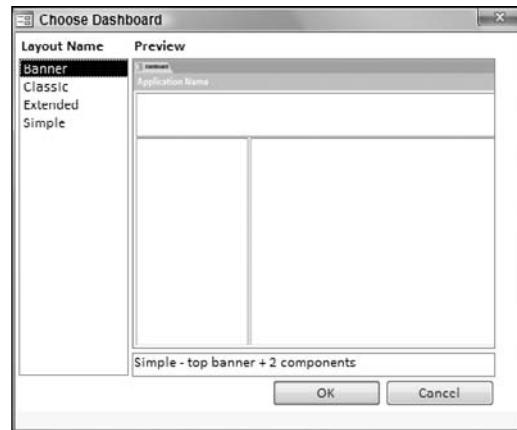


Figure 8-14

With the controls created, it's time to add some code. We're going to use the list box to filter the form to show the selected dashboard. Before we do however, we need to retrieve the selected dashboard when the form loads to apply a filter. Add the following code to the Load event of the form.

```
Private Sub Form_Load()
    ' select the currently selected id
    Dim id As Long
    Dim i As Long
```

We're using the DLookup function to get the selected DashboardID value. If we retrieve a value, then we create the filter string.

```
id = Nz(DLookup("DashboardID", "USysDashboards", "IsSelected=True"), -1)

If (id <> -1) Then
    Me.Filter = "DashboardID = " & id
    Me.FilterOn = True
```

Remember that we didn't bind the list box so in order to make the list box display the currently selected dashboard we need to add some additional code. The following code loops through the items in the list box and finds the matching ID. Once found, select it using the Selected property of the list box.

```
' select the item in the listbox
For i = 0 To Me.lstDashboards.ListCount - 1
    If (Me.lstDashboards.ItemData(i) = id) Then
        Me.lstDashboards.Selected(i) = True
        Exit For
    End If
Next
End If
End Sub
```

Part III: Interacting with the Application

When the user selects a dashboard in the list, we create a filter to display the selected dashboard. Add the following code to the AfterUpdate event of the list box:

```
Private Sub lstDashboards_AfterUpdate()
    ' filter
    If (Not IsNull(Me.lstDashboards)) Then
        Me.Filter = "DashboardID = " & Me.lstDashboards
        Me.FilterOn = True
    Else
        Me.Filter = ""
        Me.FilterOn = False
    End If
End Sub
```

When the user clicks OK we want to update the USysDashboards table to set the IsSelected field for the dashboard. This is a two step process. We can only have one selected dashboard so we need to first clear the field for all records. Next, we update the IsSelected field for the selected dashboard.

```
Private Sub cmdOK_Click()
    Dim stSQL As String

    ' update the selected menu
    If (Not IsNull(Me.lstDashboards)) Then
        stSQL = "UPDATE USysDashboards SET IsSelected = False"
        CurrentDb.Execute stSQL

        stSQL = "UPDATE USysDashboards SET IsSelected = True "
        stSQL = stSQL & "WHERE DashboardID = " & Me.lstDashboards
        CurrentDb.Execute stSQL
    End If

    ' close
    DoCmd.Close acForm, Me.Name
End Sub
```

The last bit of code we need is to close the form when the user clicks Cancel:

```
Private Sub cmdCancel_Click()
    DoCmd.Close acForm, Me.Name
End Sub
```

Opening the Selected Dashboard

Once the user has selected the dashboard form they'd like to see, we need to load the correct one. Create a new module called basDashboard and add the following code. This code opens the selected dashboard and returns it as an open Form object. This function will be used in other places in the application.

```
Public Function GetCurrentDashboard() As Form
    ' returns the current dashboard form as an open form
    Dim stDashboardForm As String

    ' get the selected dashboard
    stDashboardForm = Nz(DLookup("DashboardForm", "USysDashboards", _
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

```
"IsSelected=True"), "")  
  
' if no dashboard is selected default to dashboard 1  
If (Len(stDashboardForm) = 0) Then  
    stDashboardForm = "USysFrmDashboard1"  
End If  
  
' open the dashboard form if it is not already open  
If (Not CurrentProject.AllForms(stDashboardForm).IsLoaded) Then  
    DoCmd.OpenForm stDashboardForm  
End If  
  
' return the dashboard form  
Set GetCurrentDashboard = Forms(stDashboardForm)  
End Function
```

Next, add an initialization routine that you would typically call when the application loads. This can be called from a startup form or autoexec macro.

```
Public Function Init() As Long  
    GetCurrentDashboard  
End Sub
```

Choosing Components

So far so good. We've created a few dashboards and a mechanism for selecting a dashboard. We think the dashboards are pretty cool, but without any content there isn't much to them. So, let's spice them up with components. Start by creating a new form called USysFrmComponents with the properties listed in the table that follows.

Property Name	Property Value
Caption	Choose Components
Border Style	Dialog
Record Selectors	No
Navigation Buttons	No
Scroll Bars	Neither
Width	2.0417"
Pop Up	Yes

Next, add one text box, two combo boxes, and three command buttons to the form. Set their properties as shown in the table that follows.

Part III: Interacting with the Application

Control Name	Property Name	Property Value
Text0	Width	1.9167"
	Top	0.2917"
	Left	0.0833"
	Enabled	No
	Locked	Yes
	Name	txtCurrentDashboard
Label1	Caption	Current Dashboard
	Top	0.0833"
	Left	0.0833"
Combo2	Width	1.9167"
	Top	0.8021"
	Left	0.0833"
	Row Source Type	Value List
	Allow Value List Edits	No
	Name	cboDashboardFrames
Label3	Caption	Select frame
	Top	0.5938"
	Left	0.0833"
Combo4	Column Count	2
	Column Widths	0"
	Width	1.9167"
	Top	1.2813"
	Left	0.0833"
	Row Source	SELECT ComponentID, ComponentName FROM USysComponents ORDER BY ComponentName;

Chapter 8: Using Code to Add Advanced Functionality to Forms

Control Name	Property Name	Property Value
Label5	Name	cboComponents
	Caption	Select component
	Top	1.0729 "
Command6	Left	0.0833 "
	Caption	set
	Back Style	Transparent
	Name	cmdSet
	Cursor on Hover	Hyperlink hand
	Width	0.2708 "
Command7	Left	0.0833 "
	Caption	clear selected
	Back Style	Transparent
	Name	cmdClearSelected
	Cursor on Hover	Hyperlink hand
	Left	0.375 "
Command8	Width	0.8958 "
	Caption	clear all
	Back Style	Transparent
	Name	cmdClearAll
	Cursor on Hover	Hyperlink hand
	Left	1.2917 "
	Width	0.5833 "

After setting the properties, you should have a form that looks something like the one shown in Figure 8-15.

Part III: Interacting with the Application

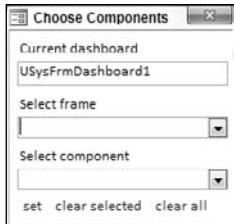


Figure 8-15

Let's add some code, beginning with the Load event:

```
Private Sub Form_Load()
    Dim id      As Long
    Dim rs      As DAO.Recordset
    Dim stSQL As String
    Dim stItem As String
```

Get the current dashboard form by calling the GetCurrentDashboard function. This also ensures that the form is open.

```
' get the current dashboard
Me.txtCurrentDashboard = GetCurrentDashboard().Name
id = CLng(Right(Me.txtCurrentDashboard, 1))
```

We also need to get the list of the subform controls from the USysDashboardComponents table. Notice that we're removing the sfrm prefix for presentation and adding them to the cboDashboardFrames combo box.

```
' get the frames for the dashboard
stSQL = "SELECT * FROM USysDashboardComponents "
stSQL = stSQL & "WHERE DashboardID = " & id
Set rs = CurrentDb().OpenRecordset(stSQL)

' fill the list of frames in the dashboard
Me.cboDashboardFrames.RowSource = ""
While (Not rs.EOF)
    stItem = Replace(rs("DashboardControl"), "sfrm", "")
    Me.cboDashboardFrames.AddItem stItem
    rs.MoveNext
Wend

' cleanup
rs.Close
Set rs = Nothing
End Sub
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

For aesthetics, we're going to add a visual cue to the dashboard form when the user selects a frame from the list of subforms. Add the following code to the `AfterUpdate` event of the `cboDashboardFrames` combo box.

```
Private Sub cboDashboardFrames_AfterUpdate()
    Dim stControl As String
    Dim frm As Form

    ' get the dashboard form
    Set frm = GetCurrentDashboard()

    ' get the control name
    stControl = "sfrm" & Me.cboDashboardFrames

    ' highlight the frame and reset the others
    HighlightFrame frm, stControl
End Sub
```

This event handler calls a routine called `HighlightFrame`, which does the actual work of setting the visual cue. The cue we're using is to thicken the border of the selected subform control and change its border color to red. Add the `HighlightFrame` routine to the form as follows:

```
Private Sub HighlightFrame(frm As Form, stControl As String)
    Const FRAME_BORDER_COLOR As Long = &HC0C0C0
    Const FRAME_BORDER_WIDTH As Long = 0           ' hairline

    Dim ctl As Control
    For Each ctl In frm.Controls
        If (ctl.ControlType = acSubform) Then
            If (ctl.Name = stControl) Then
                ctl.BorderColor = vbRed
                ctl.BorderWidth = 2
            Else
                ctl.BorderColor = FRAME_BORDER_COLOR
                ctl.BorderWidth = FRAME_BORDER_WIDTH
            End If
        End If
    Next

End Sub
```

This code loops through the controls on the specified form and changes the border color and width. After a selection is made, the dashboard form should resemble the one shown in Figure 8-16.

The last thing to do now is to set the components. For our purposes, we're going to set them one at a time using the `cmdSet` button. Add the following code to the `Click` event of the button:

```
Private Sub cmdSet_Click()
    ' set
    Dim stSQL As String
    Dim stCtl As String
    Dim lComponentID As Long
    Dim lDashboardId As Long
```

Part III: Interacting with the Application

```
Dim frm As Form
If (IsNull(Me.cboComponents) Or IsNull(Me.cboDashboardFrames)) Then
    MsgBox "Please select a component and a frame to continue", vbExclamation
    Exit Sub
End If
```

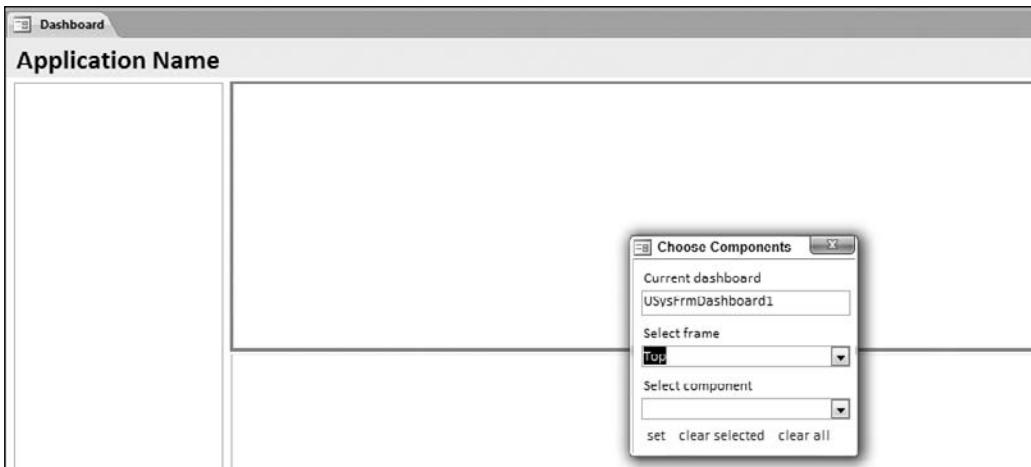


Figure 8-16

Again, we're getting the current dashboard form by calling the GetCurrentDashboard function:

```
' get the dashboard form
Set frm = GetCurrentDashboard()
```

Next, retrieve the component and dashboard ID values from the form:

```
' names and values
stCtl = "sfrm" & Me.cboDashboardFrames
lComponentID = Me.cboComponents
lDashboardId = CLng(Right(frm.Name, 1))
```

Build the SQL statement to update the USysDashboardComponents table and run it:

```
' build the SQL
stSQL = "UPDATE USysDashboardComponents SET ComponentID = " & lComponentID
stSQL = stSQL & " WHERE DashboardID = " & lDashboardId
stSQL = stSQL & " AND DashboardControl = '" & stCtl & "'"
' run the sql
CurrentDb().Execute stSQL
```

Call a routine named FillDashboard to fill components in the selected dashboard. We'll implement this routine in a moment. Call this routine to refresh the currently open dashboard:

```
' refresh
FillDashboard
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

Last, remove highlighting from the selected subform by calling the `HighlightFrame` routine and pass an empty string for the `stControl` argument:

```
' remove highlighting
HighlightFrame frm, ""
End Sub
```

Before we test the component form, we need to add code for the `cmdClearSelected` and `cmdClearAll` buttons. The `cmdClearSelected` button clears the selected component, and the `cmdClearAll` button clears the selected dashboard. Add the following code to the form:

```
Private Sub cmdClearSelected_Click()
    ' clear selected
    Dim stSQL      As String
    Dim stCtl       As String
    Dim lDashboardId As Long
    Dim frm         As Form

    ' get the dashboard form
    Set frm = GetCurrentDashboard()

    ' names and values
    stCtl = "sfrm" & Me.cboDashboardFrames
    lDashboardId = CLng(Right(frm.Name, 1))

    ' build the SQL
    stSQL = "UPDATE USysDashboardComponents SET ComponentID = NULL"
    stSQL = stSQL & " WHERE DashboardID = " & lDashboardId
    stSQL = stSQL & " AND DashboardControl = '' & stCtl & ''"

    ' run the sql
    CurrentDb().Execute stSQL

    ' clear the selected subform
    frm.Controls(stCtl).SourceObject = ""
End Sub

Private Sub cmdClearAll_Click()
    ' clear all
    Dim frm         As Form
    Dim lDashboardId As Long
    Dim stSQL      As String
    Dim ctl          As Control

    Set frm = GetCurrentDashboard()

    ' names and values
    lDashboardId = CLng(Right(frm.Name, 1))

    ' build the SQL
    stSQL = "UPDATE USysDashboardComponents SET ComponentID = NULL"
    stSQL = stSQL & " WHERE DashboardID = " & lDashboardId

    ' run the sql
    CurrentDb().Execute stSQL
```

Part III: Interacting with the Application

```
' clear the subforms
For Each ctl In frm.Controls
    If (ctl.ControlType = acSubform) Then
        ctl.SourceObject = ""
    End If
Next

End Sub
```

To test the components, open the USysFrmComponents form and select frames and components.

If you haven't already, you'll need components defined in the USysComponents table for this to work.

Filling the Dashboard

Once components have been selected, filling the dashboard is pretty straightforward. The first thing to do is retrieve the components for the currently selected dashboard. We'll do this using a query. Create a new query with the following SQL. Save the query as USysQrySelectedDashboard.

```
SELECT
    ComponentName,
    ComponentObject,
    DashboardControl
FROM USysDashboards
INNER JOIN (USysComponents INNER JOIN USysDashboardComponents ON
USysComponents.ComponentID = USysDashboardComponents.ComponentID) ON
USysDashboards.DashboardID = USysDashboardComponents.DashboardID
WHERE ((IsSelected)=True);
```

Next, open the basDashboard module created earlier and add the following routine:

```
Public Function FillDashboard() As Long
    Dim rs      As DAO.Recordset
    Dim frm     As Form
```

Get the dashboard form object by calling the GetCurrentDashboard function:

```
' get the dashboard form
Set frm = GetCurrentDashboard()
```

Get the components using the query you just created:

```
' get the components for the current dashboard
Set rs = CurrentDb().OpenRecordset("USysQrySelectedDashboard")
```

Last, fill the dashboard by setting the SourceObject property of each subform in the dashboard to the name of the component in the ComponentObject field in the query.

```
' fill the dashboard
While (Not rs.EOF)
    ' Set the SourceObject property of the dashboard subform
```

```
' to the specified component
frm.Controls(rs("DashboardControl")).SourceObject = rs("ComponentObject")
rs.MoveNext
Wend

' cleanup
rs.Close
Set rs = Nothing
End Function
```

You also need to call this function in the `Load` event for each dashboard. To do this, set the `OnLoad` property of each of the four dashboard forms to:

```
=FillDashboard()
```

When you open the currently selected dashboard, any selected components should appear in their respective subforms.

Splash Screens

The splash screen is often the first thing that a user sees when launching an application. They are an optional component of many applications, but can be useful if you need to configure portions of the application during startup. If you have a long-running process such as refreshing linked tables at startup, a splash screen lets the user know what's going on.

Quite often, a splash screen has no border or even buttons because if the splash screen is doing work behind the scenes, allowing the user to interact with the form could pose problems. We like to use a rectangle control around the detail section of the form to give the appearance of a border. An example of such a splash screen is shown in Figure 8-17.



Figure 8-17

Part III: Interacting with the Application

The buttons shown in the figure are for demo purposes and run either the Timer event or long running code that follows.

If the splash screen is doing work, a progress bar or other indication of status is useful. Consider closing the splash screen after a very short period of time if it is only being displayed for aesthetics or advertising. The following code shows you how to use the Timer event of a form to close a splash screen. Set the TimerInterval property of a form to 1000 to start the timer one second after the form opens.

```
Private Static Sub Form_Timer()
    ' countdown
    Const COUNTDOWN = 5

    Dim iCounter As Integer

    ' init or decrement
    If (iCounter = 0) Then
        iCounter = COUNTDOWN
    Else
        iCounter = iCounter - 1
    End If

    If (iCounter > 0) Then
        If (iCounter = 1) Then
            Me.txtClosing = "Closing in " & iCounter & " second"
        Else
            Me.txtClosing = "Closing in " & iCounter & " seconds"
        End If

        DoEvents
        Me.Repaint
    Else
        ' close
        Me.TimerInterval = 0
        DoCmd.Close acForm, Me.Name
    End If
End Sub
```

This code updates a text box to give the user an indication of how long the form will be open.

You can encapsulate long running processes by calling routines that contain these processes using the Run method of the application object. In this example, we have five sample routines, which represent long running processes.

```
' represents code that would run at startup
Public Function Step1() As Long
    MsgBox "This is step1"
End Function
Public Function Step2() As Long
    MsgBox "This is step2"
End Function
Public Function Step3() As Long
    MsgBox "This is step3"
End Function
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

```
Public Function Step4() As Long
    MsgBox "This is step4"
End Function
Public Function Step5() As Long
    MsgBox "This is step5"
End Function
```

The following code shows you how to run these routines in a loop and update a progress bar. It's shown here as being launched from a command button, but would likely be started from the Load event of a form.

```
Private Sub cmdDoWork_Click()
    ' runs 5 simulated routines at startup and updates a progress bar
    Dim iStep As Integer
    Dim sOffset As Single

    ' calculate the offset
    sOffset = Me.boxInside.Left - Me.boxOutside.Left

    ' reset the progress bar
    Me.boxInside.Width = 0

    For iStep = 1 To 5
        ' run the routine
        Application.Run "Step" & iStep

        ' update the progress bar
        Me.boxInside.Width = (iStep * (Me.boxOutside.Width - (2 * sOffset)) / 5)
    Next

    ' close
    DoCmd.Close acForm, Me.Name
End Sub
```

About Dialog Boxes

The About dialog box is used to display information about your application such as the version number, application name, and company name. In addition to providing users with information about the application itself, it can also be used to display other information, such as system information or credit to other developers or companies who have contributed to an application. As both the About dialog box and splash screen are important for branding an application, the About dialog box is sometimes a scaled-down version of the splash screen.

Figure 8-18 shows an example of an About dialog box.

Kiosk Forms

This one is simple but fun. A kiosk form is one you might find in your local mall or shopping center. Consider a real estate firm who has space set up in the mall and has a monitor to display current listings. These types of forms typically consume the entire screen real estate (no pun intended) to display full screen. It turns out that Access forms can easily be configured this way by simply maximizing a popup form.

Part III: Interacting with the Application



Figure 8-18

Create a new form in design view and set the `Pop Up` property to Yes. Then, add the following code to the `Load` event of the form. We're turning off `Echo` in this case so you don't see the form being maximized when it opens.

```
Private Sub Form_Load()
    With DoCmd
        .Echo False
        .Maximize
        .Echo True
    End With
End Sub
```

An example of a kiosk form is shown in Figure 8-19.

Custom Form Navigation

There are many ways to navigate through an application. You might provide buttons or hyperlinks, or give users a list of places to choose from. Here are some other ways to accomplish navigation that you might consider.

Navigation Bars

Web sites often contain navigation in a single collection of links or buttons organized into a bar at the top of the page. We can do the same thing with buttons on forms. Say that you have an application with tasks for customers, employees, and orders. Each area has a top-level button named `cmdCustomers`, `cmdEmployees`, and `cmdOrders` respectively. Underneath these buttons are groups of buttons that let the user drill into the selected area. Using the `Tag` property of the buttons, we can associate them with the top-level button so that when the user clicks on a top-level button, we show or hide the lower-level buttons as shown in the following code:

```
Private Sub ShowButtons(stTag As String)
    Dim c As Control
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

```
For Each c In Me.Controls
    If (Len(c.Tag) > 0) Then
        c.Visible = (c.Tag = stTag)
    End If
Next
End Sub
Private Sub cmdCustomers_Click()
    ShowButtons Screen.ActiveControl.Name
End Sub
Private Sub cmdEmployees_Click()
    ShowButtons Screen.ActiveControl.Name
End Sub
Private Sub cmdOptions_Click()
    ShowButtons Screen.ActiveControl.Name
End Sub
Private Sub cmdOrders_Click()
    ShowButtons Screen.ActiveControl.Name
End Sub
```

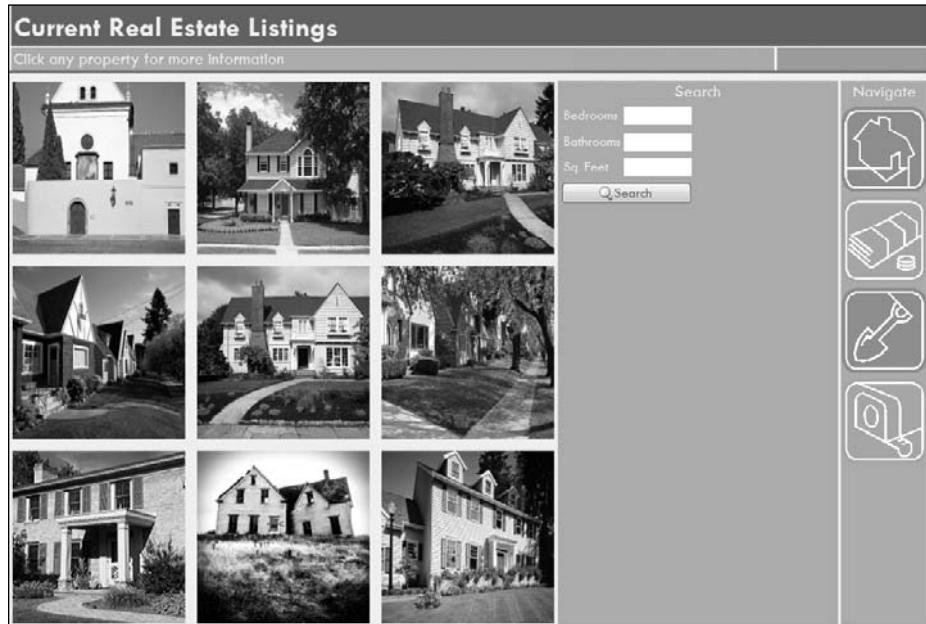


Figure 8-19

An example of the button layout is shown in Figure 8-20.

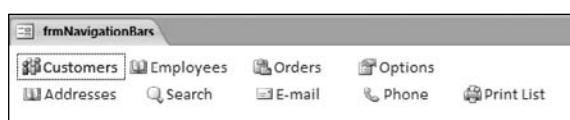


Figure 8-20

Part III: Interacting with the Application

“I Need To” Drop-Down

The “I Need To” drop-down gets its name from Web pages that we’ve seen that lists common tasks in a drop-down list. The label for the drop-down usually says “I Need To” followed by the choice of task. This is a nice way to allow for navigation in the application while preserving screen real estate.

From the navigation perspective, this drop-down is very easy to implement — simply store the friendly text and the name of a form to open in a table and then use the AfterUpdate event of a combo box control to open the form. Because the items in the list are geared toward tasks, however, you can really do much more with them. For example, you might include an item to preview a particular report or run a custom function that exports data to a file. Because of this, we like to store the name of a function (complete with its arguments) to run in a table and call it using the Eval function.

Create a new table with the fields shown in the table that follows.

Field Name	Data Type (Size)
ID	AutoNumber
DisplayText	Text (255)
Action	Text (255)

Save the table with the name USysActions and fill it with the data shown in Figure 8-21.

ID	DisplayText	Action
1	View current listings	ViewReport("rptRealEstate")
2	Export data	MyExportFunction()
3	Move list box items up and down	ViewForm("frmCategories")
4	Mark records for deletion	ViewForm("frmMarkForDeletion")
5	Handle keyboard events	ViewForm("frmKeyboardEvents")
*	(New)	

Figure 8-21

Create a new module and add the following code that will be used by the combo box.

```
' main function
Public Function RunAction(stAction As String) As Long
    Eval stAction
End Function
' functions in the table
Public Function ViewReport(stReportName As String) As Long
    DoCmd.OpenReport stReportName, acViewReport
End Function
Public Function ViewForm(stFormName As String) As Long
    DoCmd.OpenForm stFormName
End Function
Public Function MyExportFunction() As Long
    ' custom export function
```

Chapter 8: Using Code to Add Advanced Functionality to Forms

```
    MsgBox "This is the custom export function", vbInformation
End Function
```

Create a form with a combo box with the following Row Source:

```
SELECT * FROM USysActions ORDER BY DisplayText
```

Set the Column Count property of the combo box to 3 and the Column Widths property of the combo box to: 0";1";0". Then, add the following code to the AfterUpdate event of the combo box to call the RunAction routine defined earlier:

```
Private Sub cboActions_AfterUpdate()
    RunAction Me.cboActions.Column(2)
End Sub
```

When the RunAction routine is called, it passes the name of the function with its arguments stored in the combo box to the Eval function in the Access object model. The Eval function then calls the code specified in the table.

Keyboard-Driven Navigation

Point-of-sale or warehouse applications may not necessarily have a mouse attached to the computer where the application is running. In order to interact with the application, it might be necessary to use the keyboard for navigation. One popular way to do this is to use the function keys defined at the top of most keyboards. The KeyDown event shown earlier in this chapter can be used to listen for these key-strokes and take the appropriate action. The following code shows an example of what this might look like. Remember to set the Key Preview property of the form to Yes before running this code.

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    ' handle keyboard navigation (or tasks)
    Dim stCmd As String

    Select Case KeyCode
        Case vbKeyF1
            MsgBox "You pressed help!", vbInformation
            ' suppress Access help
            KeyCode = 0
        Case vbKeyF2
            ViewReport "rptRealEstate"
        Case vbKeyF3
            stCmd = "calc.exe"
        Case vbKeyF4
            stCmd = "notepad.exe"
        Case vbKeyF5
            MyExportFunction
    End Select

    ' run the command
    If (Len(stCmd) > 0) Then
        Shell stCmd, vbNormalFocus
    End If
End Sub
```

Summary

While the data is important to a database application, so is the design. A good design can go a long way in terms of enhancing the usability of an application and keeping users coming back, instead of making them go looking for other solutions. In several cases in this chapter we were able to extend functionality of an application by adding a little code that can also be made reusable by refactoring.

This chapter highlighted (not just controls), but really how code, when used with forms and controls, can also help to add the wow factor to an application. And it's not all about the wow factor — in many cases, code can be used to create user interfaces that users find intuitive and just make sense. In this chapter you saw:

- ❑ How to track form activity when records were added, updated, or deleted
- ❑ That you have a pretty good amount of control over which error messages are displayed to users
- ❑ How to use visual cues that draw the user's eye toward issues on the form that need their attention
- ❑ How to create dashboard forms to create applications that keep users engaged
- ❑ How to create standard types of forms such as splash screens and About dialog boxes that help brand your applications

In the next chapter, we look at how you can use code to create exciting, interactive reports using the new Report view of Access 2007, as well as a type of report that is not included with Access — the Calendar report.

9

Using Code to Add Advanced Functionality to Reports

Reports are an important part of many Access applications. They provide the ability to view information in the database in a manner suitable for printing. We use them for items such as invoices, calendars, and financial reporting. Access 2007 includes many new features for reports, such as improved sorting and grouping, Layout view, and Report view. In this chapter, you:

- Learn how to create interactive reports using Access 2007
- Navigate to a Web site from a report
- Simulate a new conditional formatting feature found in Excel 2007
- Utilize code to enhance reports and create different types of reports
- Create a report manager to manage reports in an application

Interactive Reports Using Access 2007

There is a new view for reports in Access 2007 called *Report view*. Report view provides interactivity to reports that are typically only available to forms. This new view allows you to search and copy data like you can on a form, but also allows for a nice printed view. Using code in Access 2007, you'll make these reports pop by adding rich interactive features such as sorting, filtering, and drill down.

Unlike forms, the data in a report is read-only. With Report view in Access 2007, you can create reports that contain similar functionality to forms but with a nice printed view. As a result, reports in Access 2007 can be used like a read-only form in many cases.

Part III: Interacting with the Application

Sorting a Report Using Controls on a Report

For this next example, you use the Northwind 2007 template database included with Access 2007. Create a new instance of the Northwind 2007 database and create a new report in Design View. Save the report as rptInteractiveSort. Set the Record Source property of the report to the following SQL statement to retrieve values from the Customers, Orders, and Employees tables.

```
SELECT Customers.Company,
       [Employees].[First Name] & " " & [Employees].[Last Name] AS EmployeeName,
       [Order ID],
       [Order Date],
       [Shipped Date],
       [Ship Name],
       [Ship City]
  FROM Employees
 INNER JOIN (Customers RIGHT JOIN Orders ON Customers.ID = Orders.[Customer ID])
    ON Employees.ID = Orders.[Employee ID];
```

After setting the record source, add the fields to the report so that it looks something like the report shown in Figure 9-1.

rptInteractiveSort						
Company	EmployeeName	Order ID	Order Date	Shipped Date	Ship Name	Ship City
Company AA	Anne Hellung-Lars	30	1/15/2006	1/22/2006	Karen Toh	Las Vegas
Company D	Jan Kotas	31	1/20/2006	1/22/2006	Christina Lee	New York
Company L	Mariya Sergienko	32	1/22/2006	1/22/2006	John Edwards	Las Vegas
Company H	Michael Nepper	33	1/30/2006	1/31/2006	Elizabeth Andersen	Portland
Company D	Anne Hellung Lars	34	2/6/2006	2/7/2006	Christina Lee	New York
Company CC	Jan Kotas	35	2/10/2006	2/12/2006	Soo Jung Lee	Denver
Company C	Mariya Sergienko	36	2/23/2006	2/25/2006	Thomas Axen	Los Angeles

Figure 9-1

After you create the report, set the following properties of the header labels in the report. Note that you're using the Tag property of the label to store the name of the field being sorted.

Label Caption	Name	Tag
Company	lblCompany	Company
EmployeeName	lblEmployeeName	EmployeeName
Order ID	lblOrderID	Order ID
Order Date	lblOrderDate	Order Date
Shipped Date	lblShippedDate	Shipped Date

Chapter 9: Using Code to Add Advanced Functionality to Reports

Label Caption	Name	Tag
Ship Name	lblShipName	Ship Name
Ship City	lblShipCity	Ship City

To make this report interactive, you'll add code to the `Click` event of each label. This code will call a helper routine called `OnApplySort`, which is defined as follows:

```
Private Sub OnApplySort(objLabel As Label)
    If (InStr(Me.OrderBy, " DESC") > 0 Or Len(Me.OrderBy) = 0) Then
        Me.OrderBy = "[" & objLabel.Tag & "]"
    Else
        Me.OrderBy = "[" & objLabel.Tag & "] DESC"
    End If

    Me.OrderByOn = True
End Sub
```

In this routine you are checking the `OrderBy` property of the report to see if the report is sorted in descending order. If it is not, then you flip the sort order by adding the `DESC` keyword. You use the `Tag` property of the control to determine the field being sorted.

Last, add code for the `Click` event of each label as shown here:

```
Private Sub lblCompany_Click()
    OnApplySort Me.lblCompany
End Sub
Private Sub lblEmployeeName_Click()
    OnApplySort Me.lblEmployeeName
End Sub
Private Sub lblOrderDate_Click()
    OnApplySort Me.lblOrderDate
End Sub
Private Sub lblOrderID_Click()
    OnApplySort Me.lblOrderID
End Sub
Private Sub lblShipCity_Click()
    OnApplySort Me.lblShipCity
End Sub
Private Sub lblShipName_Click()
    OnApplySort Me.lblShipName
End Sub
Private Sub lblShippedDate_Click()
    OnApplySort Me.lblShippedDate
End Sub
```

When you click a label, the sort order of the report should be changed to match the field that was clicked and the sort should flip between ascending and descending.

Part III: Interacting with the Application

Filtering a Report Using Controls

Because reports in Report view are interactive, you can add controls to a report that fire events. These controls can be used to extend the functionality of a report beyond that of print preview. In this example, you'll add list boxes to a report that will allow the user to filter a report. You'll set up the report so that the controls appear only in Report view, and not when the report is printed.

Start by creating a new report in Design View in the Northwind 2007 template in Access 2007. Set the Record Source property of the report to the following SQL statement:

```
SELECT Customers.Company,
       [Employees].[First Name] & " " & [Employees].[Last Name] AS EmployeeName,
       [Order ID],
       [Order Date],
       [Shipped Date],
       [Ship Name],
       [Ship City]
  FROM Employees
 INNER JOIN (Customers RIGHT JOIN Orders ON Customers.ID = Orders.[Customer ID])
   ON Employees.ID = Orders.[Employee ID];
```

This is the same record source used for the sorting example earlier. Add the fields to the report in a similar manner.

Next, make the report header section visible and add three list box controls and two command buttons to the report header. Position the list boxes above the Company, EmployeeName, and Ship City fields. Use the table that follows to set properties for the controls.

Control	Property Name	Property Value
List box 1	Name	1stFilterCompany
	Display When	Screen Only
	Row Source	SELECT DISTINCT Company FROM Customers;
List box 2	Name	1stFilterEmployee
	Display When	Screen Only
	Row Source	SELECT DISTINCT [First Name] & " " & [Last Name] AS EmployeeName FROM Employees ORDER BY [First Name] & " " & [Last Name];

Chapter 9: Using Code to Add Advanced Functionality to Reports

Control	Property Name	Property Value
List box 3	Name	lstFilterShipCity
	Display When	Screen Only
	Row Source	SELECT DISTINCT [Ship City] FROM Orders;
Command Button 1	Name	cmdFilter
	Caption	Apply Filter
	Picture Caption Arrangement	General
	Picture	Use the Picture Builder to select the Apply Filter image
	Back Style	Transparent
	Display When	Screen Only
	Cursor on Hover	Hyperlink Hand
	Name	cmdClearCriteria
	Caption	Clear Criteria
Command Button 2	Picture Caption Arrangement	General
	Picture	Use the Picture Builder to select the Pencil (Edit) image
	Back Style	Transparent
	Display When	Screen Only
	Cursor on Hover	Hyperlink Hand

After setting the properties, you should have a report that looks something like the one shown in Figure 9-2.

Part III: Interacting with the Application

The screenshot shows a Windows application window. At the top left, there are three dropdown list boxes labeled 'Company', 'EmployeeName', and 'Ship City'. The 'Company' list contains 'Company A', 'Company AA', 'Company B', 'Company BB', 'Company C', and 'Company CC'. The 'EmployeeName' list contains 'Andrew Cencini', 'Anne Hellung-Larsen', 'Jan Kotas', 'Laura Giustani', 'Mariya Sergienko', and 'Michael Neipper'. The 'Ship City' list contains 'Boise', 'Chicago', 'Denver', 'Las Vegas', 'Los Angeles', and 'Memphis'. To the right of these lists is a vertical scroll bar. At the top right of the window are two buttons: 'Toggle Filter' with a checkmark icon and 'Clear Criteria' with a brush icon. Below the lists is a grid table with columns: Company, EmployeeName, Ship City, Order ID, Order Date, Shipped Date, and Ship Name. The data rows are: Company AA (Anne Hellung-Larsen, Las Vegas, 30, 1/15/2006, 1/22/2006, Karen Toh); Company D (Jan Kotas, New York, 31, 1/20/2006, 1/22/2006, Christina Lee); Company L (Mariya Sergienko, Las Vegas, 32, 1/22/2006, 1/22/2006, John Edwards); Company H (Michael Neipper, Portland, 33, 1/30/2006, 1/31/2006, Elizabeth Andersen); and Company D (Anne Hellung-Larsen, New York, 34, 2/6/2006, 2/7/2006, Christina Lee). The rows for Company D and Company L are highlighted in grey.

Company	EmployeeName	Ship City	Order ID	Order Date	Shipped Date	Ship Name
Company AA	Anne Hellung-Larsen	Las Vegas	30	1/15/2006	1/22/2006	Karen Toh
Company D	Jan Kotas	New York	31	1/20/2006	1/22/2006	Christina Lee
Company L	Mariya Sergienko	Las Vegas	32	1/22/2006	1/22/2006	John Edwards
Company H	Michael Neipper	Portland	33	1/30/2006	1/31/2006	Elizabeth Andersen
Company D	Anne Hellung-Larsen	New York	34	2/6/2006	2/7/2006	Christina Lee

Figure 9-2

Okay, you've got the report set up the way you want so it's time to add some code. You'll start with the cmdClearCriteria button. This button will be used to clear the selections from the list boxes. Add the following code to the Click event for the button:

```
Private Sub cmdClearCriteria_Click()
    ' clear the listbox selections
    Me.lstFilterCompany = Null
    Me.lstFilterEmployee = Null
    Me.lstFilterShipCity = Null
End Sub
```

Next, add the code to the Click event of the cmdFilter button. This is where most of the work happens. Start with the declarations:

```
Private Sub cmdFilter_Click()
    ' build the filter
    Dim stFilter As String
```

You'll begin creating the filter by determining whether each list box is Null. If it is not, and the user has made a selection, you'll append a filter string to the stFilter variable defined in the event handler:

```
If (Not IsNull(Me.lstFilterCompany)) Then
    stFilter = stFilter & "Company = '" & Me.lstFilterCompany & "' AND "
End If

If (Not IsNull(Me.lstFilterEmployee)) Then
    stFilter = stFilter & "EmployeeName = '" & Me.lstFilterEmployee & "' AND "
End If

If (Not IsNull(Me.lstFilterShipCity)) Then
    stFilter = stFilter & "[Ship City] = '" & Me.lstFilterShipCity & "'"
End If
```

Depending on the user's selections, there may be an extra AND keyword left at the end of the filter string. If so, remove it:

```
' remove the last AND
If (Right(stFilter, 5) = " AND ") Then
```

Chapter 9: Using Code to Add Advanced Functionality to Reports

```
    stFilter = Left$(stFilter, Len(stFilter) - 5)
End If
```

Next, check the `FilterOn` property of the report to set the `Filter` property and toggle the caption of the `cmdFilter` button to reflect the state of the filter:

```
If (Me.FilterOn) Then
    ' remove the filter
    Me.cmdFilter.Caption = "Apply Filter"
    Me.Filter = ""
    Me.FilterOn = False
Else
    ' apply the filter
    If (Len(stFilter) > 0) Then
        Me.cmdFilter.Caption = "Remove Filter"
        Me.Filter = stFilter
        Me.FilterOn = True
    Else
        MsgBox "Please make a filter selection", vbInformation
    End If
End If
End Sub
```

Run this report in Report view and make selections using the list boxes. Click the Apply Filter button to set the filter on the report. Click the button again to clear the filter.

If you open the report in print preview, you'll notice that the list boxes and command buttons don't appear, but there is a large amount of empty space at the top of the report where the report header section exists. To prevent this, you can use the `Format` event of the section to prevent it from printing. Add the following code to the report:

```
Private Sub ReportHeader_Format(Cancel As Integer, FormatCount As Integer)
    ' hide this section in print preview, but not on screen
    Cancel = True
End Sub
```

When you view the report in print preview, the report should appear as expected.

Drill-Down

Another interesting scenario enabled by Report view in Access 2007 is drill-down. Drill-down allows you to create a report that can be used to open another report or to run other code. Let's take a look at adding interactivity using the `Click` event of controls on a report. You'll also see some new features in Access 2007 such as rich text and display as hyperlink.

Using the Northwind 2007 template included with Access 2007, create a new report based on the `Customers` table. Save the report as `rptCustomers` and add the following fields: `ID`, `Company`, and `Job Title`. Add a text box to the report and name it `txtNameAddress`. Set the properties of the `txtNameAddress` text box shown in the table that follows.

Part III: Interacting with the Application

Control Name	Property Name	Property Value
Company	Display As Hyperlink	Screen Only
txtNameAddress	Display As Hyperlink	Screen Only
	Is Hyperlink	Yes
	Text Format	Rich Text
	Control Source	= [First Name] & " " & [Last Name] & " " & [Address] & " " & [City] & ", " & [State/Province] & " " & [Zip/Postal Code]
	Can Grow	Yes

Notice that we've used the new `Display As Hyperlink` property to format the control as a hyperlink when displayed on the screen. This adds hyperlink formatting and the hand cursor to the control. If you print this report, the controls are not displayed as hyperlinks.

We've also used the new `Text Format` property of a control to include rich text in the control source of the `txtNameAddress` text box. Rich text in Access 2007 is used to add HTML formatting to text. In our case, we've added the HTML line break, `
`, as part of the control source of the field to embed carriage returns in the control. You'll notice that we've used some other features of Access 2007 such as layouts in this report, but after you've created the report it should look something like Figure 9-3.

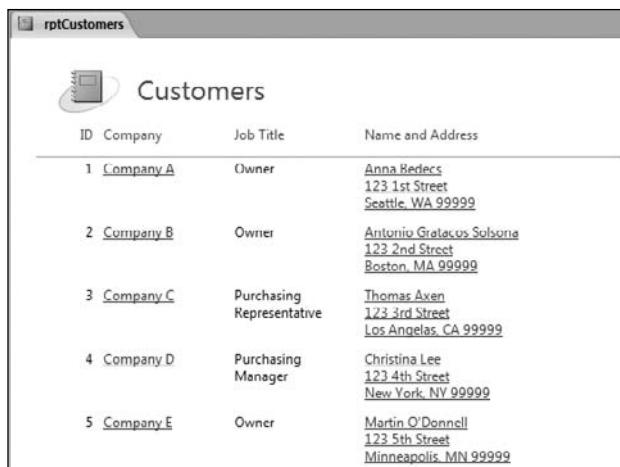


Figure 9-3

Chapter 9: Using Code to Add Advanced Functionality to Reports

Because this report is based on the `Customers` table, you probably want to drill down to order information for the selected customer. To do this, you'll first need a report based on the `Orders` table. Create a new report based on the `Orders` table in Northwind 2007. Save the report as `rptOrders`.

Now that both reports are created, you need to add some code. After all, this is a programming book! In Design View of the `rptCustomers` report, add code to the `Click` event of the `Company` text box. Add the following code to the event to open the `rptOrders` report in Report view. The report should open with orders for the currently selected customer:

```
Private Sub Company_Click()
    ' open the report with a filter
    DoCmd.OpenReport "rptOrders", acViewReport, , "[Customer ID] = " & Me.ID
End Sub
```

Navigating to a Map Dynamically

In the previous example, you actually formatted two controls as hyperlinks, where the second control was `txtNameAddress`. This text box displays the name and address of the customer in a stacked format such as:

```
Customer Name
Customer Address
Customer City, State, Postal Code
```

For this example, when the user clicks on the address, you want to open a Web page to view the address in a map. The Web site we are using is <http://maps.live.com>.

Remember that you generated the customer address using six fields: First Name, Last Name, Address, City, State/Province, and Zip/Postal Code. The address also included some HTML line-break characters for formatting. As a result, you need to parse the address out of this string. In the `rptCustomers` report created in the previous section, add the following helper function:

```
Private Function ParseAddress(stAddress As String) As String
    ' line break constant
    Const LINE_BREAK As String = "<br>

    stAddress = Mid(stAddress, InStr(stAddress, LINE_BREAK) + Len(LINE_BREAK) + 1)
    stAddress = Replace(stAddress, LINE_BREAK, " ")

    ' remove the zip code
    stAddress = Trim$(Replace(stAddress, Me.[ZIP/Postal Code], ""))
    
    ' return
    ParseAddress = stAddress
End Function
```

This code will remove the customer name and the postal code from the address block. Next, add code to the `Click` event of `txtNameAddress`, starting with the declarations:

```
Private Sub txtNameAddress_Click()
    ' locals
```

Part III: Interacting with the Application

```
Dim stAddress As String  
Dim stURL      As String
```

Now, you need to get the address using the `ParseAddress` function and then create the URL for `http://maps.live.com`. You'll then use the `FollowHyperlink` method in Access to navigate to the page:

```
' remove the name from the string  
stAddress = ParseAddress(Me.txtNameAddress)  
  
' navigate to the map of this location  
stURL = "http://maps.live.com/default.aspx?where1=" & stAddress  
FollowHyperlink stURL  
End Sub
```

When you click on an address in the `rptCustomers` report, your default Web browser should open `http://maps.live.com` with the address specified in the report.

When you created the `txtNameAddress` control you should have set the `Is Hyperlink` property to Yes. If not, you may see strange behavior where the Click event fires twice and you get two browser windows for the specified address.

Report Scenarios

The truth is that many reports in Access can be created without code and look great. However, adding code to your reports enables you to create reports that really pop and that are highly dynamic and flexible. In the following sections, you'll look at a few examples of doing just that.

Creating a Report with Data Bars

Excel 2007 includes some pretty cool conditional formatting features such as data bars and icon sets. An example of data bars in Excel 2007 is shown in Figure 9-4.

Before you do a double-take, this is still an Access book. But, we're going to create a report that simulates the data bars that appear in Excel 2007. Data bars display a gradient formatted bar in a given cell in Excel 2007. The width of the bar is a percentage of the maximum value for the range of cells. This is the logic that you'll duplicate for a report. You won't create gradients, but you will use a few tricks to pull this one off.

Again, you're using the Northwind 2007 template in Access 2007. This database includes a query called `Inventory Sold` that is the total number of products sold in the inventory system. Create a new report in Design View that contains the following SQL statement for the record source:

```
SELECT * FROM [Inventory Sold] ORDER BY [Quantity Sold] DESC;
```

Chapter 9: Using Code to Add Advanced Functionality to Reports

Product ID	Quantity Sold
Northwind Traders Beer	487
Northwind Traders Coffee	325
Northwind Traders Clam Chowder	290
Northwind Traders Chocolate	200
Northwind Traders Green Tea	200
Northwind Traders Crab Meat	120
Northwind Traders Boysenberry Spread	100
Northwind Traders Ravioli	100
Northwind Traders Mozzarella	90
Northwind Traders Chocolate Biscuits Mix	85
Northwind Traders Curry Sauce	65
Northwind Traders Dried Plums	55
Northwind Traders Syrup	50
Northwind Traders Cajun Seasoning	40
Northwind Traders Dried Pears	40
Northwind Traders Fruit Cocktail	40
Northwind Traders Marmalade	40
Northwind Traders Dried Apples	40
Northwind Traders Long Grain Rice	40
Northwind Traders Olive Oil	25
Northwind Traders Scones	20
Northwind Traders Almonds	20
Northwind Traders Chai	15

Figure 9-4

Format the report to suit your tastes, and add a label to the `Detail` section. Type the letter `g` in the caption for the label and set the properties for the label shown in the following table.

Property Name	Property Value
Name	lblDataBar
Font Name	Webdings
Font Size	10
Width	2.0417"

You'll notice that we've used the Webdings font for the label. This is because the letter `g` in the Webding font appears as a solid bar, as shown in Figure 9-5.



Product	Quantity Sold
Northwind Traders Beer	487 ■
Northwind Traders Coffee	325 ■
Northwind Traders Clam Chowder	290 ■
Northwind Traders Green Tea	200 ■
Northwind Traders Chocolate	200 ■
Northwind Traders Crab Meat	120 ■
Northwind Traders Ravioli	100 ■
Northwind Traders Boysenberry Spread	100 ■

Figure 9-5

Part III: Interacting with the Application

With all of the interactivity added by Report view in Access 2007, many new events have been added to reports to make them behave in much the same way as forms. An additional event that has been added to the Section object is the Paint event. This event fires every time Access needs to paint a section in Report view.

The Format and Print events of a section do not fire in Report view. Instead, use the Paint event. We've used Static variables called by the Paint event to reduce screen flickering.

Because you want your data bars to appear when the report is opened in both Report view and print preview, you'll need to handle both the Paint event and the Format event of the Detail section. In the report, add event handlers for both events as follows:

```
Private Sub Detail_Format(Cancel As Integer, FormatCount As Integer)
    DrawDataBar
End Sub
Private Sub Detail_Paint()
    DrawDataBar
End Sub
```

You'll notice that we're calling a routine called `DrawDataBar` that hasn't been written yet. Calling methods that you'll fill in later is called **programming by intention**. This technique is designed to help you determine which methods you'll actually need.

Now, define the `DrawDataBar` routine. Start with the declarations. Use Static variables again to reduce screen flickering that will be caused by the Paint event. This event is called frequently so Static variables can help you limit the number of times it is called.

```
Private Sub DrawDataBar()
    Static fPainted As Boolean
    Static sProduct As String
    Static lMax      As Long
    Static lMin      As Long
    Dim n           As Long
```

Next, define some constants related to the data bars. Given the width of the label, 15 characters will fit in the data bar. This is the number you'll use to determine how many characters to print.

```
Const NCHARS_IN_BAR As Long = 15      ' 15 Webdings characters fit in the bar
Const BAR_CHAR      As String = "g"    ' bar character in Webding font
```

Remember that in Excel, the width of the data bar appears as a percentage of the maximum value. The minimum values always have a width and are never empty so you need the minimum value as well.

```
' get the maximum and minimum values (once)
' for the minimum value, get the smallest non-zero value
If (lMax = 0) Then
    lMax = DMax("[Quantity Sold]", "[Inventory Sold]")
    lMin = DMin("[Quantity Sold]", "[Inventory Sold]", "[Quantity Sold] > 0")
End If
```

Chapter 9: Using Code to Add Advanced Functionality to Reports

For the minimum value, you are concerned only with non-zero values because zero values will always have the same width as the minimum value.

Reset the static flag variable when the product name changes:

```
' use static variables to reduce the amount of painting
If (sProduct <> Me.Product_ID) Then
    fPainted = False
End If
```

You need to determine how many characters to print. If the `Quantity Sold` (from the query) is the maximum value, then print the maximum number of characters (15). If the `Quantity Sold` is less than or equal to the minimum, then default to 1; otherwise, calculate the number of characters as a percentage of the maximum value.

```
If (Not fPainted) Then
    ' determine how many characters we need
    If (Me.[Quantity Sold] = lMax) Then
        n = NCHARS_IN_BAR
    ElseIf (Me.[Quantity Sold] <= lMin) Then
        n = 1 ' always show at least 1
    Else
        ' show as a percentage of the max
        n = (Me.[Quantity Sold] / lMax) * NCHARS_IN_BAR
    End If
```

Last, print the characters using the `String` function in VBA:

```
' draw the bar using the Webding font
Me.lblDataBar.Caption = String(n, BAR_CHAR)
End If
End Sub
```

When you view the report in Report view, it should look something like the report shown in Figure 9-6.

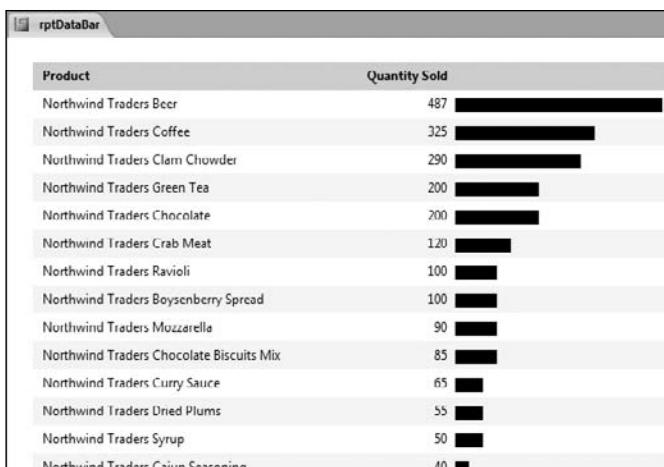


Figure 9-6

Issues to Keep in Mind When Using the Paint Event

The Paint event is very useful for some conditional formatting scenarios in Report view, but there are some things you should be aware of.

- Controls cannot be resized in the event.
- Controls cannot be hidden in the event.
- The event fires frequently, which can affect calculations. If you are filling a control with a calculated value in code, consider setting the Enabled property of the control to False.
- Certain operations can lead to flickering because the event fires frequently.

Creating a Calendar Report

Calendars are useful things. For an event tracking system, calendars provide a view of data that helps people stay organized. For an order tracking system such as Northwind 2007, a calendar can provide a snapshot of orders for a given period of time. Many, many years ago, we wrote an application for a carpet cleaning business to track work orders. To help with scheduling, we wrote a dynamic calendar report that could be printed on a monthly basis. We still use this report in newer applications that we've written. The report is completely unbound and filled with data from a recordset.

Creating the Report

One trick to creating a monthly calendar is to make sure you have enough controls. A month in the Gregorian calendar can span over a period of six different calendar weeks depending on the number of days in the month and when the first day of the month falls. Another trick is to sequentially name the controls to help with the date calculations.

So, we know that we need a report with at least 42 text boxes, sequentially named. We could start off by asking you to create a report with these controls and then format them and name them manually, but truthfully, that's somewhat boring and prone to error. We didn't do this when we created the report initially so we won't walk you through that process. Utility code to the rescue!

Start by creating a new module called `basCreateCalendarReport` and a new Sub routine called `CreateMonthlyCalendar`. Add the following declarations. Because this is utility code, we were somewhat liberal with naming conventions for this one:

```
' creates the monthly calendar report
Public Sub CreateMonthlyCalendar()
    Dim r As Report      ' report object
    Dim c As Control     ' control object
    Dim i As Integer     ' counter
    Dim s As String       ' report name
    Dim l As Double      ' control left
    Dim t As Double      ' control top
    Dim h As Double      ' control height
    Dim tb As TextBox     ' textbox object

    Const CTL_PADDING As Long = 50  ' control padding
```

Chapter 9: Using Code to Add Advanced Functionality to Reports

To create the report in code, use the `CreateReport` method in Access:

```
' create the report
Set r = CreateReport()
```

Calculate the height of the control in twips:

```
' calculate control height (0.95")
h = 0.95 * 1440
```

Next, add labels in the page header section to represent the days of the week. Controls on reports are created using the `CreateReportControl` method in Access. We're using the `WeekdayName` function in VBA to set the captions for the labels to get a localized string. The left edge of the control is calculated in the loop as being the current index of the control multiplied by the default width of a text box on a report. The width of the default control is determined by using the `DefaultControl` property of a `Report` object, which is only available in Design View.

```
' create the column headings in the page header
For i = 1 To 7
    ' calculate left
    l = (i - 1) * (r.DefaultControl(acTextBox).Width + CTL_PADDING)

    ' create the control and set the caption
    Set c = CreateReportControl(r.Name, acLabel, acPageHeader, , , l, , ,
        r.DefaultControl(acTextBox).Width, ,
        r.DefaultControl(acTextBox).Height)
    c.Caption = WeekdayName(i)

    ' formatting
    c.FontName = "Segoe UI"
    c.FontSize = 9
    c.FontWeight = 700
Next
```

We're going to re-calculate the left edge of controls so let's reset the variable that represents left (l):

```
' reset left
l = 0
```

Time to create the text boxes. Again, we're calculating the left edge of the control, which is reset every seven controls. We're also calculating the top of the control, which is the previous top of the control, plus the height (h), plus a padding of 50 twips. The code also sequentially names the controls using the prefix `txtDay` and applies some formatting, including rich text:

```
' create the text box controls
For i = 1 To 42
    ' calculate left and top
    If (i Mod 7 = 1 And i > 7) Then
        t = t + h + CTL_PADDING      ' move down
        l = 0                         ' reset left
    ElseIf i > 1 Then
        l = l + r.DefaultControl(acTextBox).Width + CTL_PADDING ' move left
    End If
```

Part III: Interacting with the Application

```
' create the control
Set c = CreateReportControl(r.Name, acTextBox, , , , -
    1, t, r.DefaultControl(acTextBox).Width, h)

' name the control
c.Name = "txtDay" & i

' formatting
Set tb = c
With tb
    .FontName = "Segoe UI"
    .FontSize = 9
    .BorderStyle = 1
    .BorderColor = vbBlack
    .TextFormat = acTextFormatHTMLRichText
End With
Next
```

Last, you want to close and save the report. You can't assign a name to the report using the `Close` method so we'll rename it:

```
' save and rename
s = r.Name
DoCmd.Close acReport, r.Name, acSaveYes
DoCmd.Rename "rptMonthlyCalendar", acReport, s
End Sub
```

When you run this code, you should have a report that looks something like the report shown in Figure 9-7.

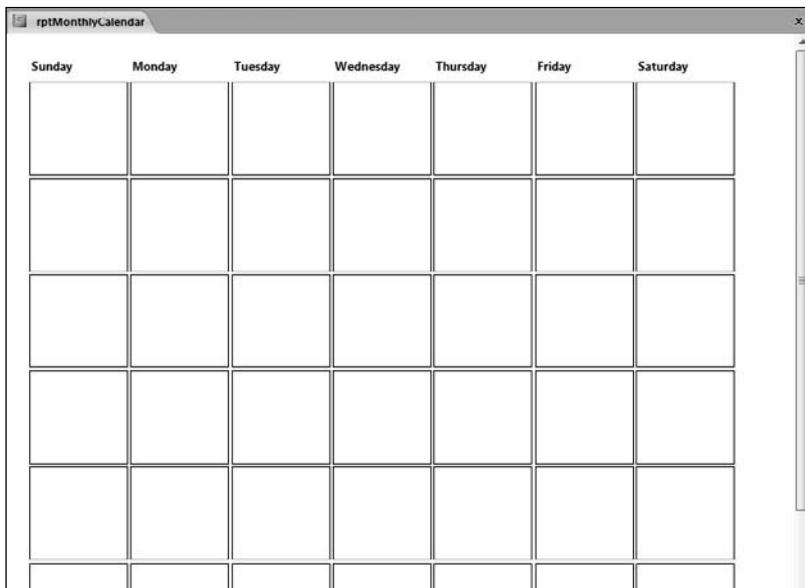


Figure 9-7

Chapter 9: Using Code to Add Advanced Functionality to Reports

This code will create a report with a fixed name: rptMonthlyCalendar. In the next section, you add code to this report to fill it with data. If you run the CreateMonthlyCalendar routine after creating the report, you may lose code. You might consider adding logic to save the report as a new name to avoid any potential of overwriting this report.

Filling the Report with Data

The rptMonthlyCalendar can be used to print empty calendars for writing information on paper. In our scenario, however, we want to fill it with data from a recordset based on the Orders table in Northwind 2007.

Start by opening the report in Design View and add the following code to the module. Again, we'll use programming by intention to define the routines that we'll actually need:

```
Dim vFirstDay As Variant      ' first of month
Private Sub Report_Open(Cancel As Integer)
    ' cancel?
    Cancel = Not ContinueFillCalendar()
End Sub
Private Sub Report_Load()
    ' made it this far - fill the calendar
    FillCalendar
End Sub
```

We've defined two routines here. The first is called ContinueFillCalendar and it returns a Boolean value that determines whether the report should be canceled. We'll explain how this works in a moment. Second, we've defined a routine called FillCalendar that actually fills the calendar. We're using the new Load event for reports to fill controls in both Report view and print preview.

Let's write these two routines, starting with ContinueFillCalendar. Add the declarations to the report module:

```
Private Function ContinueFillCalendar() As Boolean
    Const QUIT_KEYWD As String = "quit"
    Const MSG As String = "Enter the start date of the calendar" & vbCrLf & _
        "or type 'quit' to exit."
```

To provide some flexibility to the user, you're going to prompt for the beginning of the month for the report using an input box. You want to give the user a mechanism for canceling, however, so you're going to let users type the word quit in the input box to exit. Therefore, you need a loop:

```
' get the first day of the month from the user
' loop while the date is invalid or until the user enters 'quit'
Do
    vFirstDay = InputBox(MSG, "Monthly Calendar")
Loop While (Not IsDate(vFirstDay) And vFirstDay <> QUIT_KEYWD)
```

This loop will call the InputBox function until the user enters a valid date or the quit keyword. The input box appears in Figure 9-8.

Part III: Interacting with the Application

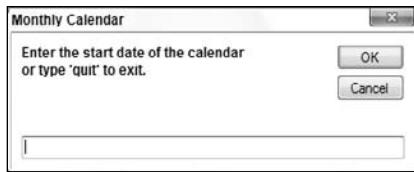


Figure 9-8

When you make it out of the loop, you need to return `True` or `False`, depending on whether you typed the `quit` keyword.

```
' determine whether we should continue
If (vFirstDay = QUIT_KEYWD) Then
    ContinueFillCalendar = False
Else
    ContinueFillCalendar = True
End If
End Function
```

The return value of this function is used as the `Cancel` argument in the `Open` event of the report as listed earlier.

Now let's move on to the `FillCalendar` routine. If the user enters a valid date and makes it past the `Open` event, the code in the `Load` event will fire, which will call this routine. So, start with the declarations:

```
Private Sub FillCalendar()
    Dim dLastDay As Date           ' last of month
    Dim dCounter As Date          ' counter
    Dim rs        As DAO.Recordset2 ' Orders recordset
    Dim rs2       As DAO.Recordset2
    Dim c         As Control
    Dim lLastCtl As Long           ' index of the last control
    Dim stSQL     As String
```

The routine will dynamically show and hide controls based on the month and year. This is to avoid empty space for days that don't exist in the calendar. To do this, you need a routine to show and hide controls depending on whether they have a value. Add the following code. We'll write this helper routine later.

```
' show all controls in case the month changes
ShowNullValueControls
```

Remember that you prompted the user for the first day of the month so you need the last day of the month. Calculate this using the `DateSerial` function in VBA:

```
' calculate the last day of the month
dLastDay = DateSerial(Year(vFirstDay), Month(vFirstDay) + 1, 0)
```

Set the caption of the report to the month and year:

```
' caption
Me.Caption = Format(vFirstDay, "mmmm yyyy")
```

Chapter 9: Using Code to Add Advanced Functionality to Reports

You've gotten some of the report set up so it's time to start filling in dates. Dates in VBA are stored as numbers of type `Double`, so you can loop through them. You initialize the index of the last control as the weekday of the first day of the month. Then, the `lLastCtl` variable is increased by one to remember the last control that was set. This value is then used with our sequential name prefix, `txtDay`, to set the value of the text box control:

```
' fill the days of the calendar
For dCounter = vFirstDay To dLastDay
    If (lLastCtl = 0) Then
        lLastCtl = Weekday(dCounter)
    Else
        lLastCtl = lLastCtl + 1
    End If

    Set c = Me.Controls("txtDay" & lLastCtl)
    c.Value = dCounter
Next
```

The next thing you want to do is to fill the calendar with orders from the `Orders` table. For this you need a recordset. The recordset will contain orders within the specified month. Start with the SQL statement and open the recordset:

```
' bind to recordset data
stSQL = "SELECT [Order ID], [Customer ID], [Order Date], Company "
stSQL = stSQL & "FROM Orders INNER JOIN Customers "
stSQL = stSQL & "ON Orders.[Customer ID] = Customers.ID "
stSQL = stSQL & "WHERE [Order Date] "
stSQL = stSQL & "BETWEEN #" & vFirstDay & "# AND #" & dLastDay & "# "
stSQL = stSQL & "ORDER BY [Order Date]"

Set rs = CurrentDb.OpenRecordset(stSQL)
```

Each control has already been filled with the date in the month so you can iterate through the controls and look for matching values in the recordset. You'll do this by filtering the recordset and enumerating through another recordset. When you find matching order dates, the order ID and company name are appended to the control's value:

```
' fill the control values with data from the recordset
For Each c In Me.Controls
    If (TypeOf c Is TextBox) Then
        If (Not IsNull(c.Value)) Then
            ' filter the recordset
            rs.Filter = "[Order Date] = #" & c.Value & "#"
            Set rs2 = rs.OpenRecordset()

            ' fill the controls with orders
            While (Not rs2.EOF)
                c.Value = c.Value & "<br>" &
                    rs2("Order ID") & "-" & rs2("Company")
                rs2.MoveNext
            Wend
            rs2.Close
```

Part III: Interacting with the Application

```
    End If  
End If  
Next
```

Now, close the recordset and do some cleanup:

```
rs.Close  
Set rs = Nothing  
Set rs2 = Nothing
```

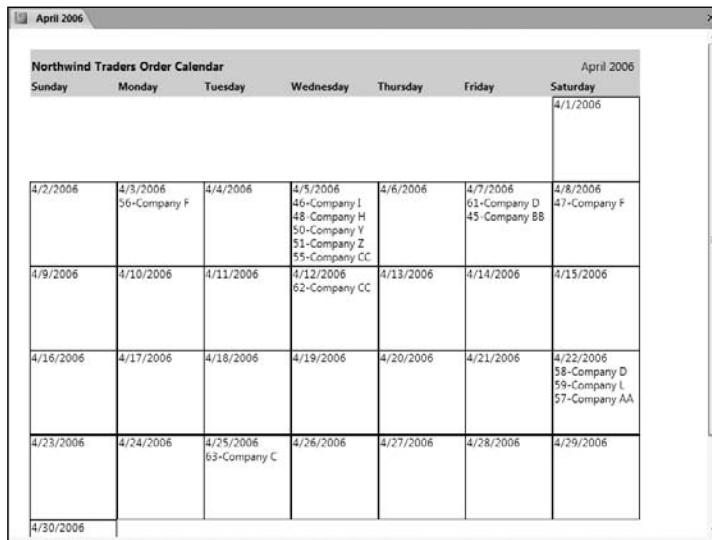
Last, you need to hide the controls that are not used:

```
' hide controls without values  
ShowNullValueControls  
End Sub
```

The `ShowNullValueControls` helper function is very straightforward. You need to iterate through the controls and hide the control if it is `Nothing`:

```
Private Sub ShowNullValueControls()  
    Dim c As Control  
  
    For Each c In Me.Controls  
        If (TypeOf c Is TextBox) Then  
            c.Visible = Not IsNull(c.Value)  
        End If  
    Next  
End Sub
```

When you open the report in the Northwind 2007 database, enter **4/1/2006** as the first day of the month. After you press OK in the input box, you should see a report similar to the one shown in Figure 9-9.



Northwind Traders Order Calendar						
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
						4/1/2006
4/2/2006	4/3/2006 56-Company F	4/4/2006	4/5/2006 46-Company I 48-Company H 50-Company Y 51-Company Z 55-Company Cc	4/6/2006	4/7/2006 61-Company D 45-Company BB	4/8/2006 47-Company F
4/9/2006	4/10/2006	4/11/2006	4/12/2006 62-Company CC	4/13/2006	4/14/2006	4/15/2006
4/16/2006	4/17/2006	4/18/2006	4/19/2006	4/20/2006	4/21/2006	4/22/2006 58-Company D 59-Company L 57-Company AA
4/23/2006	4/24/2006	4/25/2006 63-Company C	4/26/2006	4/27/2006	4/28/2006	4/29/2006
4/30/2006						

Figure 9-9

Displaying Images Using an Attachment Field

If you are using the new ACCDB file format in Access 2007, you've probably noticed a new data type called **Attachment**. This data type is used to store files in a database, but unlike the OLE Object data type, files may be compressed in the database. Finally, there is a way to store files and reduce the amount of bloat in a database!

Attachments make use of a new type of field supported by the Access database engine called a *multi-valued field*. A multi-valued field can store multiple values, or files in the case of an Attachment field. You're probably wondering whether this violates normalization rules, but not to worry — the multiple values are stored in a relational structure behind the scenes.

Because multiple attachments can be stored in a single Attachment field, you need a way to display all attachments in an Attachment field on a report. To do this, you can use a query and break down the Attachment field into multiple records. As a result, this example doesn't require any code other than the query.

Start by adding some images to the **Attachments** field in the **Products** table in Northwind 2007. Be sure to add multiple images to at least one of the records, as shown in the Attachments dialog box shown in Figure 9-10.



Figure 9-10

We've included some images along with the sample files available for download on the Web site for this book at www.wrox.com to help you get started. The images are in a subfolder named **ProductImages**.

Next, create a new report in Design View and set the Record Source property to the following query:

```
SELECT Category,
       [Product Name],
       [Product Code],
       [List Price],
       [Quantity Per Unit],
       Products.Attachments.FileName,
       Products.Attachments.FileData
  FROM Products
 WHERE Not Products.Attachments.FileName Is Null
```

Part III: Interacting with the Application

Multiple value fields store an embedded recordset so it makes sense that they contain other fields. For an attachment field, we'll retrieve the field called `FileName`, which stores the name of the file, and a field called `FileData`, which is the actual binary of the field. When you select fields in the embedded recordset, you end up with multiple records per parent record. Figure 9-11 shows the result of this query. Notice that there are two records for the product named Northwind Traders Tea.

Category	Product Name	Product Code	List Price	Quantity Per Unit	Products.Attachments.FileName
Oil	Northwind Traders Olive Oil	NWTO-5	\$21.35	36 boxes	Cooking oils with olives.png
Jams, Preserves	Northwind Traders Boysenberry Spread	NWTJP-6	\$25.00	12 - 8 oz jars	Food 559.bmp
Beverages	Northwind Traders Beer	NWTB-34	\$14.00	24 - 12 oz bottles	Half a pint of beer on bar table uid.png
Beverages	Northwind Traders Green Tea	NWTB-81	\$2.99	20 bags per box	cup of tea.png
Beverages	Northwind Traders Tea	NWTB-87	\$4.00	100 count per box	Coffee or Tea 67.bmp
Beverages	Northwind Traders Tea	NWTB-87	\$4.00	100 count per box	Food and Beverage uid 145.bmp

Figure 9-11

In the report, group on the Category and Product Name fields, as shown by the Group, Sort, and Total pane shown in Figure 9-12.



Figure 9-12

Add the Category field to the Category Header and the Product Name, Product Code, List Price, and Quantity per Unit fields to the Product Name Header. Drag the field called `Products.Attachments.FileData` to the Detail section to display the image. When you run the report, images for the product should appear below the product name on the report. For products where you added more than one image, multiple images should appear for the product, as shown by Figure 9-13.

Displaying Images Dynamically Using a Path

Storing images in an Attachment field is cool, but traditionally we've always stored the path to an image in the database rather than the image itself. This is the most efficient means of storage as far as the database is concerned.

This technique is much easier with Access 2007, and in fact, no longer requires code. The image control in Access 2007 now includes a Control Source property! This means that you can bind an image control to a field in a table or query that is the path to an image and it will render as expected.

Begin by adding a text field to the `Products` table in Northwind 2007 called `ProductImageName`. Create a subfolder in the directory where the database resides called `ProductImages` and copy some images into the new directory. In the `Products` table, add some values to the `ProductImageName` as shown:

```
\ProductImages\<ImageName.bmp>
```



Figure 9-13

Notice that we're storing a relative path to the images beginning with the name of the subfolder you created a moment ago. To retrieve the whole path, we'll use a query. Create a new report in Design View and set the Record Source property to the following query:

```
SELECT [Product Code],  
       [Product Name],  
       ProductImageName,  
       [CurrentProject].[Path] & [ProductImageName] AS ProductImagePath  
FROM Products  
WHERE Not Products.ProductImageName Is Null
```

The preceding code uses the `Path` property of the `CurrentProject` object to determine the path for the current database. We concatenate this to the name of the image to generate a new field called `ProductImagePath`. This field should now contain the entire path to an image.

In Design View of the report, add an image control from the Ribbon. Click the Cancel button when prompted for the image and set the `Control Source` property of the control to `ProductImagePath`. When you browse the report, you should have something that looks like the report shown in Figure 9-14.

Creating a Report Manager

Adding a report manager to your application provides a centralized location for users to work with reports. Depending on the number of reports in an application, the report manager might include features such as the following:

- Categorization of reports
- Usage tracking

Part III: Interacting with the Application

- Print reminders
- Print or preview of a single report
- Sending a report via e-mail
- Bulk print or e-mailing of reports
- Enabling users to add reports



Figure 9-14

Let's take a look at how you can implement some of these features in a report manager component for your applications.

Creating the Reports Table

The core of the report manager is a table to store information about reports that are tracked by the system. Many of us use naming conventions for objects, such as `rpt` for a report. One primary feature of the report manager therefore is to present the user with a friendly name for a report. This friendly name and a description of the report will be stored in the table. Create a new table in the Northwind 2007 database you've been using throughout this chapter with the fields shown in the following table.

Field Name	Data Type (Size)
ReportID	AutoNumber
ReportCategory	Text (50)
ReportName	Text (64)
ReportDisplayName	Text (100)
ReportDescription	Text (255)
LastOpened	Date/Time
TimesViewed	Number (Long Integer)
BulkPrint	Yes/No

Chapter 9: Using Code to Add Advanced Functionality to Reports

Save this table as `USysTblReports` so that Access will treat it like a system table and hide it in the navigation pane. Add some data to the table to track reports in Northwind 2007. In the sample available for download with the book, we've filled this table with information about the reports that were created in the chapter, as shown in Figure 9-15.

ReportID	ReportCategory	ReportName	ReportDisplayName	ReportDescription
1	Interactivity	rptCustomers	Customer Report	Demonstration of drill down using a report in Access 2007.
2	Formatting	rptDataBar	Data bars ala Excel 2007	Simulates the data bar conditional formatting found in Excel 2007.
3	Formatting	rptIconSet	Icon sets ala Excel 2007	Simulates the icon set conditional formatting found in Excel 2007.
4	Interactivity	rptInteractiveFilter	Interactive Filter	Shows how to filter a report using list boxes on the report.
5	Interactivity	rptInteractiveSort	Interactive Sort	Shows how to sort a report using labels on the report.
6	Calendars	rptMonthlyCalendar	Orders Monthly Calendar	An unbound, dynamic monthly calendar based on data from a recordset.
7	Interactivity	rptOrders	Orders Report	Demonstration of drill down using a report in Access 2007. This is the target report.
8	Catalogs	rptProductCatalog	Product Catalog 1 (Attachments)	This report displays images from an attachment field.
9	Catalogs	rptProductCatalog2	Product Catalog 2 (Paths)	This report uses the new Control Source property of an image control to display images.
*	(New)			

Figure 9-15

Implementing Report Manager Features

It's time to start adding features to the report manager. The report manager will be implemented using a form. Start by creating a new form called `USysFrmReportManager` and set the Record Source property of the form to `USysTblReports`. Add all fields from the table to the form. We're using the new Split Form view to provide navigation of the reports being tracked.

The end result of our form is shown in Figure 9-16.

The screenshot shows the `Report Manager` form in Microsoft Access. The form is split into two main sections: a navigation pane on the left and a details pane on the right.

Navigation Pane (Left):

- Header: `Report Manager`, `Filter report category` dropdown.
- Report Information section:
 - Display Name: `Customer Report`
 - Report Description: `Demonstration of drill-down using a report in Access 2007.`
- Report Name dropdown: `rptCustomers`.

Details Pane (Right):

- Usage section:
 - Last Opened
 - Times Viewed: 0
- Commands section:
 - Preview
 - Report View
 - Bulk Print
 - Email
- Bulk Print section:
 - Mark as Bulk Print
 - Bulk Print All
 - Clear Bulk Print
- Report list grid:

Mark as Bulk Print	Display Name	ReportCategory
<input type="checkbox"/>	Customer Report	Interactivity
<input type="checkbox"/>	Data bars ala Excel 2007	Formatting
<input type="checkbox"/>	Icon sets ala Excel 2007	Formatting
<input type="checkbox"/>	Interactive Filter	Interactivity
<input type="checkbox"/>	Interactive Sort	Interactivity
<input type="checkbox"/>	Orders Monthly Calendar	Calendars
<input type="checkbox"/>	Orders Report	Interactivity
<input type="checkbox"/>	Product Catalog 1 (Attachments)	Catalogs
<input type="checkbox"/>	Product Catalog 2 (Paths)	Catalogs
*		

Figure 9-16

Part III: Interacting with the Application

Categorization

We've added support for categorization of reports using the ReportCategory field. This field enables you to add structure to an application with many reports. Categorization can be displayed using a tree view control, multiple list boxes, or a combo box for filtering. In our report manager form, we'll add filtering by category.

Add a combo box to the Form Header section of the form. Name the combo box cboReportCategory and set the Row Source property to the following query:

```
SELECT DISTINCT "[ALL]" AS ReportCategory  
FROM MSysObjects  
UNION SELECT DISTINCT USysTblReports.ReportCategory  
FROM USysTblReports;
```

This query adds the string [All] to the beginning of the list of categories. You'll use this string to determine whether to clear the filter and show all reports. If a category is selected, you'll filter the report manager form by the selected category. Add the following code to the AfterUpdate event of the combo box:

```
Private Sub cboReportCategory_AfterUpdate()  
    ' filter the form by category  
    If (Me.cboReportCategory = "[All]") Then  
        Me.Filter = ""  
        Me.FilterOn = False  
    Else  
        Me.Filter = "ReportCategory = '" & Me.cboReportCategory & "'"  
        Me.FilterOn = True  
    End If  
End Sub
```

Print, Preview, and E-mail

The report manager provides a single location where users can interact with reports. Therefore, it makes sense to give them a way to print, preview, or e-mail the report. To do this, we've added three buttons to the form. Set the following properties of the buttons to achieve the look we created in Figure 9-16.

Control	Property Name	Property Value
Command Button	Name	cmdPreview
	Caption	Pre&view
	Picture Caption Arrangement	General
	Picture	Use the Picture Builder to select the Preview image
	Alignment	General
Command Button	Name	cmdReportView

Chapter 9: Using Code to Add Advanced Functionality to Reports

Control	Property Name	Property Value
	Caption	&Report View
	Picture Caption Arrangement	General
	Picture	Use the Picture Builder to select the MS Access Report image
	Alignment	General
Command Button	Name	cmdEmail
	Caption	&Email
	Picture Caption Arrangement	General
	Picture	Use the Picture Builder to select the Envelope (e-mail) image
	Alignment	General

To open the report in print preview, add the following code to the Click event of cmdPreview:

```
Private Sub cmdPreview_Click()
    If (Not IsNull(Me.ReportName)) Then
        DoCmd.OpenReport Me.ReportName, acViewPreview
    End If
End Sub
```

To open the report in Report view, add the following code to the Click event of cmdReportView. You might also add a button to print to the printer.

```
Private Sub cmdReportView_Click()
    If (Not IsNull(Me.ReportName)) Then
        DoCmd.OpenReport Me.ReportName, acViewReport
    End If
End Sub
```

To send the report in e-mail, add the following code to the Click event of cmdEmail. This sends the report as a PDF attachment and requires the Save as PDF download from the Microsoft Web site.

```
Private Sub cmdEmail_Click()
    Dim stPath As String

    If (Not IsNull(Me.ReportName)) Then
        stPath = CurrentProject.Path & "\" & Me.ReportName & ".pdf"
        DoCmd.SendObject acSendReport, Me.ReportName, acFormatPDF, , , , , True
    End If
End Sub
```

Part III: Interacting with the Application

Tracking Report Usage

The next feature you want to add is usage information. When a report is opened, you want to track the last time it was opened and the number of times it was opened. This information is useful for determining which reports are viewed most often and most recently. You might use this data to present the user with a button or menu item to open the report that was viewed last.

To track this information you need a function. Create a new module called `basReports` and add the following routine:

```
Public Function UpdateReportTable(oReport As Report) As Long
    Dim rsReports As DAO.Recordset
    Dim stSQL      As String

    ' build the SQL statement for the reports recordset
    stSQL = "SELECT * FROM USysTblReports "
    stSQL = stSQL & "WHERE ReportName = '" & oReport.Name & "'"
    Set rsReports = CurrentDb.OpenRecordset(stSQL)

    ' update TimesViewed and LastOpened
    rsReports.Edit
    rsReports("TimesViewed") = Nz(rsReports("TimesViewed"), 0) + 1
    rsReports("LastOpened") = Now()
    rsReports.Update

    ' cleanup
    rsReports.Close
    Set rsReports = Nothing
End Function
```

This routine accepts a `Report` object as a parameter. To track usage data for a report, call this function from the `Open` event of each report. Open the `rptCustomers` report you created earlier in Design View. Set the `OnOpen` property of the report to:

```
=UpdateReportTable([Report])
```

We've used an expression here to call the routine but you could have just as easily called it from other code in the event. Each time you open the `rptCustomers` report, the `LastOpened` and `TimesViewed` fields should be updated.

Bulk Print

The bulk print feature lets you print multiple reports at once. To mark a report for bulk print, set the `BulkPrint` field in the table to Yes. Add a command button to the form named `cmdBulkPrint` and add the following code for the `Click` event of the button:

```
Private Sub cmdBulkPrint_Click()
    Dim rsBulk As DAO.Recordset
    Dim stSQL  As String
```

Chapter 9: Using Code to Add Advanced Functionality to Reports

The first thing you want to do is commit any records on the form in the event there is a pending change. Add the following code:

```
' commit any changes so we have the latest data  
Me.Dirty = False
```

Next, get the reports that are marked for bulk print:

```
' get the reports marked for bulk print  
stSQL = "SELECT * FROM USysTblReports "  
stSQL = stSQL & "WHERE BulkPrint = True"  
Set rsBulk = CurrentDb.OpenRecordset(stSQL)
```

If no reports are marked, you want to display a message so the user is not left wondering if anything happened.

```
' display a message if there are no reports  
If (rsBulk.RecordCount = 0) Then  
    MsgBox "There are no reports marked as bulk print", vbExclamation  
End If
```

Now, open each report in the recordset. For demo purposes here, we're just opening the reports in Report view:

```
' open all reports (in Report view for the demo)  
While (Not rsBulk.EOF)  
    DoCmd.OpenReport rsBulk("ReportName"), acViewReport  
    rsBulk.MoveNext  
Wend
```

Finally, clean up the recordset object and end the routine:

```
' cleanup  
rsBulk.Close  
Set rsBulk = Nothing  
End Sub
```

To test this functionality, check the `BulkPrint` field for some of the reports and click the `cmdBulkPrint` button.

Allowing Users to Add Their Own Reports

If you're designing an off-the-shelf application, chances are you're going to fill it with as many reports as you feel meet users' needs. However, if your application includes a component to let users create custom queries, you may also want to allow them to create custom reports. You may even have a framework for building reports as a part of your application using report generation code as you saw earlier in this chapter. If your users can create their own reports, it's pretty simple to open up the report manager so they can find them later.

We'll use a combo box to give users a choice of the report to add to the report manager. Start by opening the report manager form in Design View. Right-click on the `ReportName` control and select `Change To` and then choose `Combo Box`. This will change the control from a text box to a combo box.

Part III: Interacting with the Application

To give the user a choice of reports in the database, set the `Row Source` property of the `ReportName` combo box to the following SQL statement:

```
SELECT DISTINCT [Name]
FROM MSysObjects
WHERE ((([Name]) Not Like "~*")
AND (([Type])=32764)
AND (([Name]) Not IN (SELECT ReportName FROM USysTblReports)));
```

This statement will retrieve all reports from the database that are not being tracked in `USysTblReports`. We're using the `MSysObjects` system table to retrieve the list of reports.

Last, we want to prevent users from updating existing reports so add the following code to the `Current` event of the form to enable or disable the control depending on whether the form is on a new record.

```
Private Sub Form_Current()
    Me.ReportName.Enabled = Me.NewRecord
End Sub
```

Summary

Users and managers alike love reports and they are a vital part of many applications. Even though you can create really cool reports without using code, we looked at some scenarios where adding even a little code can really make a difference to the feel of the application. More specifically we looked at:

- Creating a report manager component to help users manage reports and to provide information about reports
- Using the new Report view in Access 2007 to make reports come to life
- Dynamically generated reports such as a monthly calendar, and how you can use code to create the report to reduce chances for error
- Two new ways to display images on a report in Access 2007

We're halfway done with Part III at this point but still have a lot to cover. In the next chapter, you learn how to use automation to turn your applications into solutions that provide features for your users beyond what Access itself can provide.

10

Using Automation to Add Functionality

As much power as Access provides on its own, it doesn't exist in a vacuum. Applications may have requirements to send e-mail to recipients or print custom invoices using Word. When an application grows outside the core functionality of Access, it may be time to start looking at using other applications to help get the job done. By using tools for what they are designed to do and piecing them together, applications turn into solutions. In this chapter, you learn how to use automation to drive other applications to help get the job done. More specifically, you learn the following:

- ❑ The basics of automation, including the low-level difference between the `CreateObject` and `GetObject` functions
- ❑ How to automate objects in Windows such as the Windows Shell and the file system
- ❑ How to automate applications in Office to extend the functionality of your applications beyond Access
- ❑ How to automate Internet Explorer to integrate the Web into your applications

Automation Basics

The term *automation* refers to driving another process via code. There are several ways to do this in VBA, ranging from simply launching an application to actually manipulating an application using its object model. We look at both of these techniques in this chapter.

The basis for automation is the Component Object Model (COM), which states that communication with objects happens via their interfaces. Thus, under the hood, automation works by connecting to an interface of a particular object called the `IDispatch` interface. This interface provides the framework necessary for discovery and execution of methods in an object model. An in-depth discussion of the `IDispatch` interface is beyond the scope of this book, but there are many great resources on the subject.

Part III: Interacting with the Application

When to Automate

When you need a piece of functionality that Access doesn't provide, or when another application might be better suited, it is time to consider automation. Knowing the environment in which your application is running is important. If the application that you're trying to automate is not on the machine, you'll run into problems.

Shell Function

The `Shell` function in VBA allows you to run virtually any application from code. The application does not need to be an ActiveX application, and does not need to expose a type library. The `Shell` function accepts two arguments: a command line, which is required, and an optional `WindowStyle` argument.

The command line must include the name of an application to run, and may include additional command-line arguments that will be passed to the application. For example, you can pass the name of an application by itself:

```
Shell "notepad.exe"
```

or you can pass the application name and a set of arguments:

```
Shell "notepad.exe c:\a.txt"
```

In addition, you may specify a `vbAppWinStyle` enumerated constant to control how the resulting application window is displayed. The following table describes these constant values.

vbAppWinStyle constant	Value	Description
<code>vbHide</code>	0	Window is hidden and focus is passed to the hidden window.
<code>vbNormalFocus</code>	1	Window has focus and is restored to its original size and position.
<code>vbMinimizedFocus</code>	2	Window is displayed as an icon with focus.
<code>vbMaximizedFocus</code>	3	Window is maximized with focus.
<code>vbNormalNoFocus</code>	4	Window is restored to its most recent size and position. The currently active window remains active.
<code>vbMinimizedNoFocus</code>	6	Window is displayed as an icon. The currently active window remains active.

CreateObject vs. GetObject

Before we discuss the differences between `CreateObject` and `GetObject`, let's review the term "automation server." An automation server is an application that registers itself as being able to be controlled via automation. These servers can be referenced by either a programmatic ID, or `ProgID`, or the classification identifier, or `CLSID`. The `CLSID` is a globally unique identifier (GUID) that represents an application. A GUID is a 128-bit integer that can be hard to use, so in VBA we tend to use the `ProgID` to refer to applications.

Unless you use the `New` keyword to instantiate automation objects, you'll likely use either the `CreateObject` or `GetObject` functions to create instances of objects. The `CreateObject` function accepts two parameters: `Class` and `ServerName` where the `ServerName` argument is optional. This function is used to create a new instance of an automation server using its programmatic ID, or `ProgID` as the `Class` argument. The `CreateObject` function calls down to the `CoCreateInstance` API function.

The `GetObject` function also accepts two parameters: `PathName` and `Class`, both of which are marked as optional. Although both arguments are optional, one of them must be supplied. When you open a file using an application that is registered as an automation server such as Access, Excel, or Word, the application adds an entry for the file into the running object table, or ROT. The ROT is used by Windows to track which objects are currently running. When you specify the `PathName` argument, the `GetObject` function tries to find an entry in the ROT for the file. If it finds one, it will then try to obtain the `IDispatch` pointer to the automation server that is registered for the file extension specified by the `PathName` argument.

The `IDispatch` pointer is used to control an automation server such as the Office applications.

The `Class` argument of the `GetObject` function is also the `ProgID` of a given application. If you specify the `Class` but not the `PathName`, such as `Set obj = GetObject(, "Word.Application")`, the `GetObject` function will try to return a running instance of the application, Word in this case. If the application is not running, a new instance will be created as it is with `CreateObject`. If multiple instances are running, the `GetObject` function will return the instance that was launched most recently. This is because on each boot of the automation server, the ROT is updated to point to the document registered for the class defined by the application. The `GetObject` function ultimately uses the `GetActiveObject` API function to read from the running object table.

Because class modules created in VBA are not registered with the system, they cannot be instantiated using `CreateObject` or `GetObject`.

Early Binding vs. Late Binding

Early binding is used when you know the type of the object you want to automate at compile-time and the environment of the application being deployed is known ahead of time or not subject to change. Here are some of the advantages of early binding:

- ❑ Compile-time validation of property and method references
- ❑ Improved performance
- ❑ Fewer runtime errors

Part III: Interacting with the Application

The opposite of early binding is late binding. Early binding forces your hand with regard to the environment of the target computers because it uses a reference to an object model. As you know, a broken reference can cause problems in an application. Late binding is useful in the following situations:

- When multiple objects share an object model or members of an object model
- When you don't know at compile time which object will actually be used
- When you want version independence and portability
- When an object cannot be instantiated and you want trappable runtime errors

Discovering Object Models Using Built-in Tools

The Visual Basic environment includes an object browser, which allows you to explore each of the methods and properties exposed in an object model that is referenced in your application.

When you are in the Visual Basic Environment, choose Object Browser from the View menu, or press the F2 shortcut key to launch the browser. The object browser is shown in Figure 10-1.

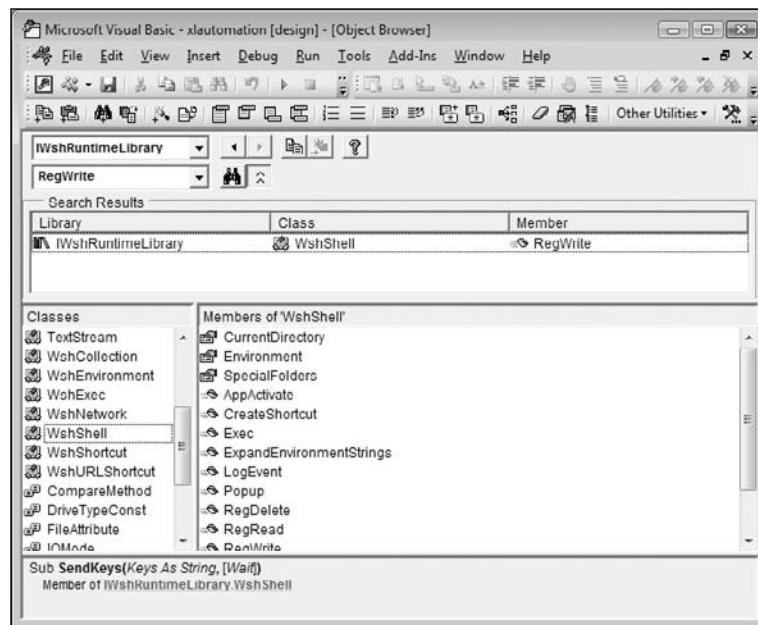


Figure 10-1

You can explore every object model that is referenced by your project. By default, all non-hidden members in every referenced project are displayed in the browser. You can display hidden members by choosing Show Hidden Members from the context menu. When you choose this option, hidden members are displayed in a grayed text. Hidden members in the object browser are shown in Figure 10-2.

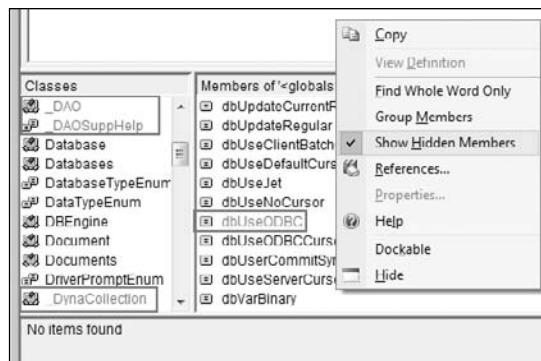


Figure 10-2

You can filter the objects displayed in the object browser by choosing a specific object model library from the Project/Library drop-down list. If you know the name of the object you want to view, enter the name or partial name into the search box. This will display all matching objects in the Search Results list. For example, if you want to know which enumeration the dbLong constant belongs to, you can enter dbLong in the search box and press Enter. This is also helpful when you can't remember the exact value you are looking for, but you remember something related. Figure 10-3 shows where to choose a reference from the Project/Library drop-down list.

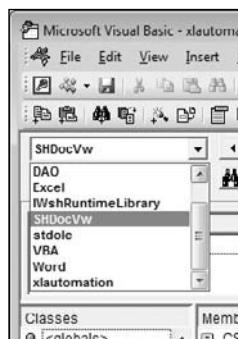


Figure 10-3

If you need to browse an object model that isn't required in your application, you can temporarily add a reference to an object model. When you are finished browsing, you should remove the reference to improve performance and reduce the chance of errors when you distribute your application.

Automating Windows

We can use VBA code to automate many aspects of the Windows environment. Many Windows system components act as Automation servers.

Part III: Interacting with the Application

In this section, we look at two of the most useful Windows automation servers (at least from the VBA automation perspective): the Microsoft Shell Controls and Automation library, and the Windows Scripting Host.

The **Shell** Object

To reference the Windows Shell objects, add a reference to %systemroot%\system32\shell32.dll, or browse the Add Reference dialog box for Microsoft Shell Controls and Automation. The objects, methods, and properties exposed in this library provide advanced customization and control of the Windows Shell (file system, Explorer windows, and controls).

Opening Folders

The `Shell` object exposes two methods that can be used to open Explorer windows on a folder. The `Open` method opens an explorer window in the default view as defined by your system settings. The `Explore` method opens an Explorer window, and displays a folder tree.

The following class demonstrates how easy it is to use the `Shell` object to open folders. Create a new standard module named `Shell32Functions`, which declares a module level object variable. Use the `New` operator to automatically instantiate the object whenever it is used.

```
Option Compare Database
Option Explicit
Dim m_sh As New Shell32.Shell
Sub OpenFolder(folderPath As String)
    m_sh.Open folderPath
End Sub
Sub ExploreFolder(folderPath As String)
    m_sh.Explore folderPath
End Sub
```

The `OpenFolder` method wraps the `Open` method, and the `ExploreFolder` method wraps the `Explore` method on the `Shell` object. Simply call the method passing a fully qualified folder path!

```
ExploreFolder "c:\windows\system32"
OpenFolder "c:\program files\microsoft office"
```

Running a Command

Instead of using the `Shell` VBA function, the `ShellExecute` method on the `Shell` object provides additional control and flexibility. `ShellExecute` allows you to choose which directory the process will use as the current directory.

`ShellExecute` accepts five arguments, as shown in the table that follows. The `file` argument is required, and specifies either the name of an application to run such as `notepad.exe` or the name of a document to execute such as `c:\a.txt`. The remaining arguments are all optional.

ShellExecute Argument	Description
File	Name of the application to run or document file to open. Application name may be fully qualified or relative to the current directory.

Chapter 10: Using Automation to Add Functionality

ShellExecute Argument	Description
vArguments	Command line to pass to the application. Optional.
vDirectory	Folder to use as the current directory. Optional.
vOperation	Operation to take on the file. The possible operations correspond to the <i>verbs</i> that can be used with a file such as <i>open</i> or <i>print</i> .
vShow	One of the vShowConstants shown in the table below.

Unlike the `Shell` VBA function, the command line is not specified as one string. Instead, you must identify the application file to name in one argument, and the argument list in a separate argument. The ability to specify a document name is unique to the `ShellExecute` method and a nice enhancement over the `Shell` VBA function. If you specify a document name, this method uses the system registry to find the default handler associated with the document extension. If a handler is registered for the document extension, the handler is launched and the document name passed to the application according to the settings in the registry.

If you specify a document name, you can also pass a supported verb in the `vOperation` argument. If a verb is present, when the `ShellExecute` method performs the registry lookup, it will check for the handler associated with the verb you pass. For example, if you pass a text file name in the file argument, and specify the verb `print`, as follows:

```
ShellExecute "c:\a.txt", , , "print"
```

the file will be passed to Notepad, printed to the default printer, and then Notepad will terminate.

If you specify a `vShow` argument, you can control how the resulting application window is displayed. When this argument is omitted, the default behavior is to open the window in normal state (similar to passing the constant value 1). The valid values for the `vShow` argument are shown in the table below.

vShow Value	Description
0	Opens application with a hidden window.
1	Opens the application with a normal window.
2	Opens the application with a minimized window.
3	Opens the application with a maximized window.
4	Opens the application with a normal window; active window remains unchanged.
5	Opens the application with its window at its current size and position.
7	Opens the application with a minimized window; active window unchanged.
10	Opens the application with its window in the default state.

Part III: Interacting with the Application

The following code demonstrates how to use the `ShellExecute` method to open an html document in Notepad, start iTunes, and print an image file.

```
Dim objShell as Object
Set objShell = CreateObject("Shell.Application")

objShell.ShellExecute "notepad.exe", "foo.html", _
"c:\users", , 2

objShell.ShellExecute "itunes.exe", vbNullString, _
"c:\users\michael\music"

objShell.ShellExecute "c:\Screenshot.jpg", , , "print", 1
```

Launching an Item in the Control Panel

The `ControlPanelItem` method on the `Shell` object opens a specified Control Panel item. The argument is the name of a valid Control Panel applet. If the name is not fully qualified, the path is searched using normal path rules. If the file specified cannot be located, no error is thrown and the method fails silently.

The following code demonstrates how to use the `ControlPanelItem` method to display the Display Settings and the Windows Firewall Control Panel applets.

```
Dim objShell As New Shell32.Shell
objShell.ControlPanelItem "desk.cpl"
objShell.ControlPanelItem +"c:\windows\system32\firewall.cpl"
```

Most Control Panel applets are installed in the `%systemroot%\system32` folder, and are identified by a `.cpl` file extension.

Sending Mail Using the Default Mail Program

Using the `ShellExecute` method, you can create a simple procedure to create an e-mail using the default e-mail program registered on your system. The following procedure prepares a fully formed `mailto:` protocol URL, and passes the URL to the `ShellExecute` function. When the URL is passed to the `ShellExecute` method, the default e-mail handler is called.

```
Sub SendEmail(recipients As String, subj As String, _
body As String)

Const fmt As String = "mailto:{0}?subject={1}&body={2}"

m_sh.ShellExecute _
    StringFormat(recipients, olto, subj, body)
End Sub
```

The `ShellExecute` method of the `Shell` object is a wrapper around the `ShellExecute` API function introduced in Chapter 2.

The `StringFormat` function was introduced in Chapter 5. As a reminder, this function is repeated here:

```
Public Function StringFormat(ByVal item As String, _
ParamArray args() As Variant) As String
```

```
Dim i As Integer
For i = LBound(args) To UBound(args)
    item = Replace(item, "{" & i & "}", args(i))
Next
StringFormat = item
End Function
```

Windows Scripting Host

In Chapter 5, we looked at how the Windows Scripting Host object model makes it easy to work with files in your file system. But this object model has many other uses. In fact, it is one of our favorite object models because of the wide variety of Windows functionality that it exposes and how easy it is to control.

In this section, we explore some of the other features of this very useful and flexible control.

Reading from and Writing to the Registry

Three methods are exposed by the `WshShell` object that provide access to the Windows Registry. Use the `RegRead` method to read a value from the registry. The `RegWrite` method writes a value to the registry and will modify an existing value or create a new value if it does not already exist. Finally, the `RegDelete` method will remove a value or node from the registry.

All three methods have a required argument that contains a fully qualified path to the registry key or value to be used by the method. If you are not familiar with the Windows Registry and how to form a path to a Windows Registry key or value, see the discussion in Chapter 12. The path name must begin with one of the five root key names listed in the table that follows.

Root Key Name	Abbreviation
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_CLASSES_ROOT	HKCR
HKEY_USERS	<i>no abbreviation</i>
HKEY_CURRENT_CONFIG	<i>no abbreviation</i>

You may use either the root key name or the associated abbreviation if one is available. To specify a registry key name instead of a value, add a terminating backslash. If there is no terminating backslash, the argument specifies a value.

RegWrite Method

The `RegWrite` method writes a value to a specified registry key or value:

```
RegWrite strRegistryPath, varnewValue
RegWrite strRegistryPath, varnewValue, strType
```

Part III: Interacting with the Application

If the registry value already exists, the value will be updated with the new value you pass. If the registry key or value does not exist, it will be created.

By default, new items are created as string values, but you can specify the type of value to create using the `strType` argument. The following table shows the names of types that you can use and a description of what you will get when using them. Pass these types as a `String` to the `strType` argument of the `RegWrite` method.

Type	Description
REG_SZ	A string value
REG_DWORD	A numeric value (maximum 32-bit integer value)
REG_BINARY	A binary value (expressed as an integer array)
REG_EXPAND_SZ	A string value with optional environment string tokens

You can change the type of an existing registry value by specifying a new type when you call `RegWrite`.

If you specify a registry key in the path argument, the value will be written to the default value for the key. If you specify a registry value in the path argument, the value you pass will be written to that registry value.

The following code will create or modify several registry values. This code requires a reference to the Windows Script Host Object Model to run.

```
Public Sub RegistryDemo
    Dim objShell As New IWshRuntimeLibrary.WshShell

    // uses full registry hive name
    objShell.RegWrite "HKEY_CURRENT_USER\Wrox174029>Title",_
        "Expert Access Programming"

    // uses abbreviated registry hive name
    objShell.RegWrite "HKCU\Wrox174029\Author1", _
        "Cooper, Rob"

    // explicitly declares value type REG_SZ
    objShell.RegWrite "HKCU\Wrox174029\Author2", _
        "Tucker, Michael", "REG_SZ"

    // type argument is required if value is not a string
    objShell.RegWrite "HKCU\Wrox174029\AuthorCount", 2, _
        "REG_DWORD"

    // value contains an embedded environment variable
    objShell.RegWrite "HKCU\Wrox174029\SamplesFolder", _
        "%USERPROFILE%\Wrox\174029", "REG_EXPAND_SZ"
```

RegRead Method

The `RegRead` method reads the value of a specified registry key or value:

```
RegRead strRegistryPath
```

If you specify a path to a registry key (by placing a backslash at the end of the path argument), `RegRead` will return the default value for the key. If no default value exists, `RegRead` returns an empty string instead of an error. If you are unexpectedly receiving empty strings, check to see if you have a backslash character at the end of your path argument.

If the path you specify does not exist, a slightly misleading runtime error is thrown: The system cannot find the file specified (number -2147024894). You will want to handle this error in your code.

The following code extends the `RegistryDemo` function to read the values that we wrote using the `RegWrite` method:

```
Dim i As Long
Dim strTitle As String
Dim cAuthors As Long
Dim rgAuthors() As String
Dim strFolder As String
Dim strMessage As String

'// read title from registry
strTitle = objShell.RegRead("HKCU\Wrox174029>Title")

'// read authors from registry into an array
cAuthors = _
    objShell.RegRead("HKCU\Wrox174029\AuthorCount")
If cAuthors > 0 Then
    ReDim rgAuthors(0 To (cAuthors - 1))
    For i = 0 To cAuthors - 1
        rgAuthors(i) = objShell.RegRead(_
            "HKCU\Wrox174029\Author" & (i + 1))
    Next i
End If

'// read the REG_EXPAND_SZ registry value
strFolder = _
    objShell.RegRead("HKCU\Wrox174029\SamplesFolder")
```

Now that we have read our values from the Registry, let's display them in a message box:

```
strMessage = strFolder

For i = 0 To cAuthors - 1
    strMessage = strMessage & vbCrLf & rgAuthors(i)
Next i

'// Display the message box
MsgBox strMessage, vbInformation, strTitle
```

Part III: Interacting with the Application

RegDelete Method

The `RegDelete` method deletes a registry key or value:

```
RegDelete strRegistryPath
```

If you specify a path to a registry value, that value will be deleted from the registry. If you specify a path to a registry key, that key and all values under that key are removed. However, `RegDelete` cannot delete a key if there are child keys under it. You must delete all nested keys before you delete an outer key.

The following code further extends the `RegistryDemo` function to remove the registry keys we created. Each `RegDelete` call removes one registry value.

```
'// Delete the registry strings  
shell.RegDelete "HKCU\Wrox174029\Title"  
shell.RegDelete "HKCU\Wrox174029\Author1"
```

Here, we delete all of the remaining values under the `Wrox174029` key as well as the key itself.

```
'// Delete everything in the node  
shell.RegDelete "HKCU\Wrox174029\"  
End Sub
```

Expanding REG_EXPAND_SZ Values

You may create a registry value with a `REG_EXPAND_SZ` type value. Many programs have the capability to expand environment variables embedded within this type of registry value. Even though `RegWrite` has the capability to create this type of registry value, `RegRead` does not automatically expand any embedded environment variables. Use the `ExpandEnvironmentStrings` method of the `WshShell` object to perform this expansion for you.

```
objShell.ExpandEnvironmentStrings( _  
    shell.RegRead("HKCU\Wrox174029\SamplesFolder"))
```

Working with REG_MULTI_SZ Values

You cannot create `REG_MULTI_SZ` values with the `RegWrite` method, but you can read them with the `RegRead` method. However, if you attempt to assign the result to a string variable, you will receive a `Type mismatch` error because `RegRead` returns `REG_MULTI_SZ` values as a variant array.

When you need to read `REG_MULTI_SZ` values, assign the result to a variant array.

For example, you can create a `REG_MULTI_SZ` registry key using the registry editor REGEDIT, as follows:

- ❑ Start REGEDIT.
- ❑ Select the `HKEY_CURRENT_USER` node.
- ❑ Click the `Edit` menu, and then choose `New ↴ Multi-String Value`.
- Figure 10-4 shows the creation of a new Multi-String Registry Key.
- ❑ Name the new key `Authors`.

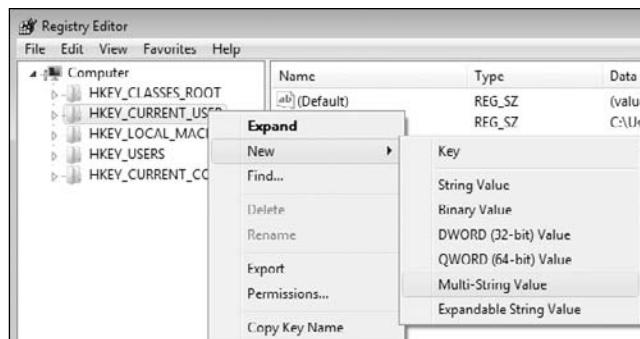


Figure 10-4

- ❑ Double-click on the new key value to launch the Edit Multi-String dialog box.
Figure 10-5 shows the Edit Multi-String dialog box.

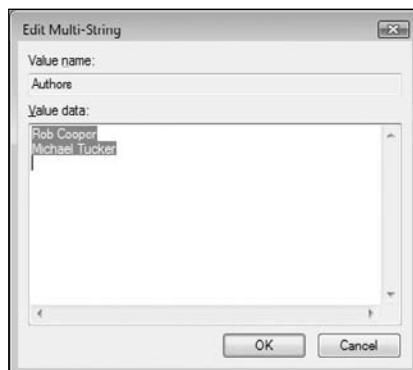


Figure 10-5

- ❑ Enter two names (Rob Cooper, Michael Tucker), and then click OK to save your changes.

Once you have created the multi-string registry key, the following code will read the value and display the names in a message box:

```
Sub ReadMultiStringKey()
    Dim objShell As New IWshRuntimeLibrary.WshShell

    Dim vrgAuthors() As Variant
    Dim vItem As Variant
    Dim strMessage As String

    'RegRead returns REG_MULTI_SZ values as a variant array
    vrgAuthors = objShell.RegRead("HKCU\Authors")

    For Each vItem In vrgAuthors
        strMessage = strMessage & vItem & vbCrLf
    End For
    MsgBox strMessage
End Sub
```

Part III: Interacting with the Application

```
    Next vItem  
  
    MsgBox strMessage  
End Sub
```

Determining Special Folders

The `WshShell` object exposes the `SpecialFolders` collection, which can be used to determine the path to the Windows special folders. Use one of the `SpecialFolder` string constants listed in the table that follows as the index into the collection.

WshShell.SpecialFolder String	Description
AllUsersDesktop	Shared desktop folder
AllUsersStartMenu	Shared start menu (shortcuts that appear in every user's Start menu)
AllUsersPrograms	Shared Programs folder
Desktop	User desktop folder
Favorites	User favorites folder
Fonts	System fonts folder
MyDocuments	User document folder
NetHood	Network Neighborhood folder
PrintHood	Printers folder
Programs	User programs folder
Recent	Folder with shortcuts to documents recently used by the user
SendTo	User Send To folder
StartMenu	Folder with shortcuts that appear on the StartMenu
Startup	Folder with shortcuts to applications to run when user logs into Windows
Templates	Default Template folder

The following code demonstrates how to use the `WshShell` object to determine the actual path to the current user's Desktop and My Documents folders. Notice that you must pass the constant value as a string.

```
Dim objShell As New IWshRuntimeLibrary.WshShell  
Debug.Print objShell.SpecialFolders("Desktop")  
Debug.Print objShell.SpecialFolders("MyDocuments")
```

Chapter 10: Using Automation to Add Functionality

There are three additional, very useful, special folders that are not exposed in this collection, but instead are exposed from the `GetSpecialFolder` method of the `FileSystemObject`. The `GetSpecialFolder` method returns a `Folder` object in a `Variant`, which means you can cache the object in a `Folder` variable and then use any of the methods on the `Folder` object such as `CreateTextFile`. Because it is a `Variant`, it also behaves intelligently if you reference the return value as a `String`: It displays the fully qualified path to the temporary folder.

The `GetSpecialFolder` method takes one argument, a `SpecialFolder` enumerated constant. The enumeration has three values, described in the following table.

FileSystemObject.SpecialFolder Enumeration Constants	Description
WindowsFolder	%windir%
SystemFolder	%windir%\System32
TemporaryFolder	User's temporary object folder

Unlike the `SpecialFolders` collection described earlier, which accepts a string constant argument, the argument here is an enumerated constant. This means you must pass the argument without quotation marks.

The following code demonstrates how to use the `GetSpecialFolder` method to identify these special folders:

```
Dim fso As New IWshRuntimeLibrary.FileSystemObject  
  
MsgBox fso.GetSpecialFolder(TemporaryFolder)  
MsgBox fso.GetSpecialFolder(SystemFolder)  
MsgBox fso.GetSpecialFolder(WindowsFolder)
```

Creating a Temporary File in the Temp Folder

We can use the `GetSpecialFolder` method together with the `GetTempName` method to generate a randomly generated file name rooted in the Temporary folder. This is a highly reliable (but not 100 percent guaranteed) way to generate a file name that can be created with minimal risk of collision.

The following function demonstrates the use of these two methods to create a temporary file and return a `TextStream` object that can be used elsewhere to read and write data to the file:

```
Function CreateTempFolder() As iWshRuntimeLibrary.TextStream  
  
    Dim fso As New IWshRuntimeLibrary.FileSystemObject  
    Dim fileName As String  
  
    On Error GoTo Err_Handler
```

In this sample, we instantiate a new `FileSystemObject` inside the function, but if you have a globally defined `FileSystemObject`, you can use that without instantiating a new one here.

Part III: Interacting with the Application

The file name is generated in one line of code. We use the `BuildPath` method to combine the folder name that is returned by the `GetSpecialFolder` method with the randomly generated file name returned by the `GetTempName` method.

```
Generate_Unique_Name:  
    fileName = fso.BuildPath( _  
        fso.GetSpecialFolder(TemporaryFolder), fso.GetTempName() )
```

That generated file name is passed to the `CreateTextFile` method. The `overwrite` argument of the `CreateTextFile` method is set to false, to ensure that if the file already exists, an error will be thrown. Because `GetTempName` generates random file names, it is unlikely but possible that a file name will be generated that already exists. If the `CreateTextFile` method cannot create the file, it throws a runtime error 58, which we trap in the error handler.

```
Set CreateTempFolder = fso.CreateTextFile(fileName, False, True)
```

Our cleanup block merely releases the `FileSystemObject` that we instantiated (this is not necessary if you use a globally declared `FileSystemObject`).

```
Cleanup:  
    Set fso = Nothing  
  
Terminate:  
    On Error GoTo 0  
    Exit Function
```

The error handler specifically handles the runtime error 58 by clearing the error, and then resuming execution at the line of code that generates a new file name. Any other errors result in a message box displaying the error, and then returning control to the calling procedure without opening a file.

```
Err_Handler:  
    Select Case Err.Number  
        Case 58      '// File already exists  
            Err.Clear  
            Resume Generate_Unique_Name:  
        Case Else  
            MsgBox Err.Description, vbCritical, Err.Number  
    End Select  
    On Error Resume Next  
    Set CreateTempFolder = Nothing  
    Resume Cleanup:  
End Function
```

Creating Desktop Shortcuts

The `WshShell` object exposes the `CreateShortcut` method to create a new file system shortcut. The method accepts one argument, which is a fully qualified file name that will hold the shortcut data, and returns a `WshShortcut` object.

```
Dim objShell As New IWshRuntimeLibrary.WshShell  
Dim myShortcut As IWshRuntimeLibrary.WshShortcut  
  
myShortcut = objShell.CreateShortcut(FileName)
```

Chapter 10: Using Automation to Add Functionality

To create a shortcut, you must perform three actions:

- ❑ Call `CreateShortcut` to obtain a valid `WshShortcut` object.
- ❑ Set properties on the `WshShortcut` object.
- ❑ Call the `Save` method on the `WshShortcut` object.

At a minimum, you must set the `TargetPath` property of the `WshShortcut`. This specifies the file that will be opened when a user clicks on the shortcut. There are many other properties that provide additional functionality. The code that follows sets several additional properties.

For a complete list of the properties, search the Windows SDK Documentation for `WshShortcut` Object.

The following code demonstrates how to create a desktop shortcut that, when clicked, will open the database:

```
Function CreateShortcut()
    Dim objShell As New IWshRuntimeLibrary.WshShell
    Dim scItem As IWshRuntimeLibrary.WshShortcut
    Dim scFileName As String
```

The `CreateShortcut` method and `WshShortcut` object are exposed through the `WshShell` object. This code instantiates a new `WshShell` object, but if you have one globally available, you can use that object instead.

We build the file name for the shortcut by getting the fully qualified path to the desktop, and then concatenating the file name we want. The shortcut will use the file name as the display text (minus the extension). We then pass the constructed file name to the `CreateShortcut` method. The method instantiates a new `WshShortcut` object and returns it; we cache that object in the `scItem` variable.

```
scFileName = objShell.SpecialFolders("Desktop") & "\Our Access Shortcut.lnk"
Set scItem = objShell.CreateShortcut(scFileName)
```

Once we have the valid `WshShortcut` object, we need to set properties on the object. At a minimum, we must set the `TargetPath` property. We set this to the value returned by the `CurrentProject.FullName` property.

```
With scItem
    .TargetPath = CurrentProject.FullName
    .Description =
        "A desktop shortcut to your Access application"
    .WindowStyle = 3      '// Maximize the window
    .WorkingDirectory =
        shell.SpecialFolders("MyDocuments")
    .IconLocation =
        "%systemroot%\system32\shell32.dll, 5"
    .Save
End With
```

Part III: Interacting with the Application

In addition to the required `TargetPath` property, we set several other properties. The `Description` property contains text that is displayed in a tooltip when you hover over the shortcut. The `WindowStyle` property controls how the application window is created; here, we have set the `WindowStyle` to 3, which will maximize the window. (1 will have it open in the default style, and 7 opens the window minimized in the system tray.)

The `WorkingDirectory` property allows you to override the default Windows behavior and specify what directory will be the “current” directory when the application is opened. In this case, we set the working directory to the user’s Documents directory.

Finally, we set the `IconLocation` property. This property accepts the name of an icon file, or an executable or dynamic-link library file that contains embedded icon resources. The file name must be either fully qualified or must contain environment variables that, when expanding, result in a fully qualified file name. Here, we point to the shell32.dll system library, which contains a large number of icons designed specifically for shortcut and application icons. By default, the first icon in the file will be used; however, you can choose a specific icon by adding the ordinal index of the icon in the file. Here, I have requested the fifth icon in the file be used for the shortcut.

Once all the properties have been set, you must call the `Save` method on the `WshShortcut` item to cause it to be saved to disk. If you set the properties but fail to call the `Save` method, no error occurs, but no shortcut appears either!

Finally, we include a simple cleanup block to release each of the items we instantiated.

```
Cleanup:  
    Set scitem = Nothing  
    Set shell = Nothing
```

```
Terminate:  
    Exit Function  
  
End Function
```

Using the `WshShell` Object to Wait for a Process to Complete

We can use the `WshShell` object to implement a simpler version of the `RunApplication` procedure that we implemented earlier in Chapter 2. The `WshShell` object implements an `Exec` method that accepts a command line to execute. The method returns a `WshExec` object, which contains information about the process that was started by the `Exec` method. Unlike the `Shell` VBA function, the `Exec` method does not provide any way to control the window style of the resulting process. If you do not need this functionality, however, this `WaitShell` implementation simplifies your code slightly, at the expense of requiring a properly registered Windows Scripting Host library.

```
Sub WaitShell(cmdline As String)  
    On Error GoTo Err_Handler  
  
    Dim objShell As New IWshRuntimeLibrary.WshShell  
    Dim processInfo As IWshRuntimeLibrary.WshExec  
  
    Set processInfo = objShell.Exec(cmdline)
```

Chapter 10: Using Automation to Add Functionality

As with all helper functions of this sort, we enable error handling to ensure that our code is well behaved, especially when running under the Access runtime environment. We then create a new instance of the `WshShell` object, and create a variable to hold the `WshExec` object that will be returned by the `Exec` method. Then, we call the `Exec` method, forwarding the command-line argument. Assuming the command line is well formed, the `Exec` method will run the command line and return a `WshExec` object that contains, among other things, the process ID of the new process and a status indicator for the process.

You do not need to do anything with the `WshExec` object that is returned. If you don't cache the object, you can finish your procedure and terminate. The process that was created by the `Exec` method will continue to run independent of your application. Because we want to wait until the process has terminated, we need to cache the object, so that we can check the status indicator to know when the process has finished.

We enter a loop that simply calls the `DoEvents` method as long as the `Status` property on the `WshExec` object returns 0, which means the process is still running.

```
Do While 0 = processInfo.Status
    DoEvents
Loop
```

The `Cleanup` block checks to see if the process is still running (i.e., if the `Status` property is 0); if it is running, it calls the `Terminate` method on the `WshExec` object to tell Windows to shut down the process. It then releases the object variables. Our `Terminate` block simply disables error handling and returns control to the calling procedure.

```
Cleanup:
If Not processInfo Is Nothing Then
    If processInfo.Status = 0 Then
        Debug.Print "Process still running: ending"
        processInfo.Terminate
    End If
    Set processInfo = Nothing
End If
Set objShell = Nothing

Terminate:
On Error GoTo 0
Exit Sub
```

The error handler merely reports the description and error number from the VBA `Error` object. If there is a valid `WshExec` object, we also report the process ID and status code of the process. If the command line is invalid, the `Exec` method will not be able to create a process, and so no `WshExec` object will be stored in the `processInfo` variable; thus, the need for the `If` block. (Alternatively, we could skip reporting the process ID and status code.)

```
Err_Handler:
If processInfo Is Nothing Then
    MsgBox Err.Description, vbCritical, Err.Number
Else
    MsgBox Err.Description & vbCrLf & "Status:" & _
        Nz(processInfo.Status, "Null"), vbCritical, _
        "PID: " & Nz(processInfo.ProcessID, "Null") & _
        " Error: " & Err.Number
```

Part III: Interacting with the Application

```
End If  
Resume Cleanup  
  
End Sub
```

Working with Printers

The `WshNetwork` object exposes several methods that can be used to add, remove, and enumerate printer connections on a computer. The following are the methods that work with printers:

- `AddWindowsPrinterConnection`
- `AddPrinterConnection`
- `EnumPrinterConnections`
- `RemovePrinterConnection`
- `SetDefaultPrinter`

Adding Printers

The `AddWindowsPrinterConnection` method creates a new network printer definition on the computer. If the printer connection already exists on the computer, it is refreshed. The method requires one string argument, which specifies the UNC address of the printer connection. Two optional arguments are present for compatibility with Windows 95/98/ME; these arguments are ignored on Windows 2000 and newer operating systems.

The following code will add a new printer connection for the printer located at network address `\PrinterPool1\ColorPrinter`.

```
Dim net As New IWshRuntimeLibrary.WshNetwork  
  
net.AddWindowsPrinterConnection _  
    "\PrinterPool1\ColorPrinter"
```

The following procedure demonstrates how to create a generic `AddPrinter` procedure using the `AddWindowsPrinterConnection` and `SetDefaultPrinter` methods. The procedure accepts two arguments: a required string that contains a UNC path to the printer we want to connect, and an optional Boolean argument that indicates whether we want this new printer to be the default printer on this system (by default, this argument is set to False).

```
Sub AddPrinter(uncPrinter As String, _  
    Optional bDefaultPrinter As Boolean = False)  
    Const ERR_PRINTERNAMEINVALID As Long = -2147023095  
    Const ERR_AUTOMATIONEXCEPTION As Long = -2147352567  
    Const ERR_REMOTE SERVERUNAVAIL As Long = 462  
    On Error GoTo Err_Handler
```

These constants are likely error codes that are trapped in the error handler.

Chapter 10: Using Automation to Add Functionality

A new `WshNetwork` object is instantiated, and then the `AddWindowsPrinterConnection` is called, passing in the printer UNC path argument.

```
Dim net As New IWshRuntimeLibrary.WshNetwork  
  
net.AddWindowsPrinterConnection uncPrinter
```

We test the optional argument; if it is true, we call the `SetDefaultPrinter` method, passing the same UNC path that we used to create the printer connection.

```
If bDefaultPrinter Then  
    net.SetDefaultPrinter uncPrinter  
End If
```

The `Cleanup` block releases the `WshNetwork` object we instantiated. The `Terminate` block is our standard block that merely resets error handling to the default behavior.

```
Cleanup:  
    Set net = Nothing  
Terminate:  
    On Error GoTo 0  
    Exit Sub
```

The error handler traps three common errors. If the network is unavailable, we will receive an `ERR_REMOTE SERVER UNAVAIL` error. If the `uncPath` argument points to a non-existent location or is otherwise malformed, we will receive an `ERR_PRINTERNAMEINVALID` error. If we call the `SetDefaultPrinter` method passing a name that does not exist in the `Printers` collection on this machine, a generic `ERR_AUTOMATIONEXCEPTION` error is thrown.

```
Err_Handler:  
    Select Case Err.Number  
        Case ERR_REMOTE SERVER UNAVAIL  
            MsgBox "Remote server unavailable.", vbCritical, Err.Number  
        Case ERR_PRINTERNAMEINVALID  
            MsgBox "Unable to create printer connection." _  
                & "The printer name is invalid.", _  
                vbCritical, Err.Number  
        Case ERR_AUTOMATIONEXCEPTION  
            // probably the UNC isn't connected  
            MsgBox "Unable to set default printer", _  
                vbCritical, Err.Number  
        Case Else  
            MsgBox Err.Description, vbCritical, Err.Number  
    End Select  
  
    Err.Clear  
    Resume Cleanup  
  
End Sub
```

Part III: Interacting with the Application

Removing Printers

The `RemovePrinterConnection` method removes an existing printer definition on the computer.

The following code uses the `RemovePrinterConnection` method to remove a printer connection from the current machine's printer collection. The procedure requires a single string argument that specifies the name of the printer connection we want to remove:

```
Sub RemovePrinter(uncPrinter As String)
    Const ERR_NETWORKCONNDOESNOTEXIST As Long = -2147022646
    Dim net As New IWshRuntimeLibrary.WshNetwork
    On Error GoTo Err_Handler
    net.RemovePrinterConnection uncPrinter
```

We define a constant for a likely error condition, and then instantiate a new `WshNetwork` object. We then call the `RemovePrinterConnection` method, passing the printer name argument from the procedure.

The error handler traps for one likely error condition: If we pass a printer name that does not exist in the Printer Connection collection on our machine, when we call `RemovePrinterConnection`, we will receive a runtime error. In this case, the error handler swallows the error and proceeds with cleanup. Any other errors are reported and then we proceed to the Cleanup block.

```
Cleanup:
    Set net = Nothing

Terminate:
    On Error Goto 0
    Exit Sub
Err_Handler:
    Select Case Err.Number
        Case ERR_NETWORKCONNDOESNOTEXIST
            '// NO-OP
        Case Else
            MsgBox Err.Description, vbCritical, Err.Number
    End Select

    Err.Clear
    Resume Cleanup

End Sub
```

Enumerating Printers

The `EnumPrinterConnections` method can be used to enumerate each of the printers currently connected on this system. This method is called with no arguments, and it returns a `WshCollection` object.

The content of the `WshCollection` is a bit odd. There are two items in the collection for each defined printer: the first item is the Printer Port identifier, and the second item is the name of the printer connection. (This will be the UNC path if you created the printer connection using the `AddWindowsPrinterConnection` method.)

The following code shows how to enumerate the collection using a `For...Next` loop with a step value of 2 to read the printer pairs:

```
Sub EnumPrinters()
    Dim net As New IWshRuntimeLibrary.WshNetwork
    Dim ptrs As IWshRuntimeLibrary.WshCollection
    Dim i As Long

    On Error GoTo Err_Handler

    Set ptrs = net.EnumPrinterConnections
    For i = 0 To ptrs.Count - 1 Step 2
        Debug.Print "Port: " & ptrs(i), _
                    " URL: " & ptrs(i + 1)
    Next i

    Cleanup:
    Set ptrs = Nothing
    Set net = Nothing

    Terminate:
    Exit Sub

    Err_Handler:
    MsgBox Err.Description, vbCritical, Err.Number
    Resume Cleanup

End Sub
```

Automating Office Applications

Given that Access is included with the Office suite of products, it makes sense that you would want to automate other applications in the suite. For example, you might choose to automate Word to generate an invoice or letter, Excel to create a chart, or Outlook to create an e-mail. Let's look at some ways that you can automate some of the other applications in Office to add this functionality to your applications.

Determining If an Office Application Is Installed

Before you run code that automates another Office application, you should verify that the Office application is installed on the computer.

You can determine this in several ways. One of the simplest techniques is to check the `Program Files\Microsoft Office` folder tree for the main executable file for the application you want to automate. For example, if you want to verify whether Microsoft Word is installed, you can check for `c:\program files\microsoft office\office 12\winword.exe`. This is reliable in most cases, but if the program is installed into a non-default location, this technique might incorrectly report that an application is not installed.

Part III: Interacting with the Application

We will use a technique that interrogates the Registry for the file location of the automation handler for the application we are interested in, and then verifies that the file actually exists on disk. This technique will work regardless of the install location specified when the application was installed, and will correctly detect the case where a user has incorrectly removed an application (for example, by deleting the folder instead of running an uninstall routine).

We will create two functions to support this scenario, as shown in the table that follows.

Function	Description
PathToApplication(progId [,version])	Returns a fully qualified path to the automation handler for the specified ProgID (and optional version)
ApplicationInstalled(progId)	Returns true if the specified ProgID has an automation handler installed

The `PathToApplication` function accepts two arguments, the `ProgID` of the application you are verifying, and optionally, a version number. If you omit the version argument, the default handler (usually, the most current version of the application) is returned. Programs that support Automation register themselves when they are installed by creating a key under the `HKLM\Software\Classes` node. The key name is the `ProgID` of the application. For example, Microsoft Access (whose `ProgID` is `Access.Application`) creates a key named `HKLM\Software\Classes\Access.Application`. Under this node, it creates a string key named `CLSID`, which contains a GUID.

This GUID is the name of another Registry key, located at `HKLM\Software\Classes\CLSID`, which contains implementation details for the automation controller, including, among other items, the path to the automation handler application. Continuing with the Access example, the `HKLM\Software\Classes\Access.Application\CLSID` key contains the default value `{73A4C9C1-D68D-11D0-98BF-00A0C90DC8D9}`. We use the GUID to query `HKLM\Software\Classes\CLSID\{73A4C9C1-D68D-11D0-98BF-00A0C90DC8D9}\LocalServer`. The default value of that key reports the install location for the automation handler. On our machine, this value appears as `C:\PROGRA~1\MI1933~1\Office12\MSACCESS.EXE`, which is the short file name representation of the `C:\Program Files\Microsoft Office` folder on our machine.

Let's look at the actual implementation of this function. First, we declare two string constants with the template of the registry keys we are going to be searching for. Tokens for the `ProgID`, Version number, and `CLSID` GUID have been added that will be replaced by the `StringFormat` function (introduced in Chapter 5).

```
Option Compare Database
Option Explicit
Public Const ERR_REG_READ As Long = -2147024894
Function PathToApplication(progid As String, _
    Optional version As Long = -1) As String
    Const rkClSID As String = _
        "HKLM\Software\Classes\{0}{1}\CLSID\" &
    Const rkServerPath As String = _
        "HKLM\Software\Classes\CLSID\{0}\LocalServer32\"
```

Chapter 10: Using Automation to Add Functionality

If we pass a version value when we call the function, we create a string variable, `strVersion`, that contains the formatted version. For example, if we are looking for version 12, we need to append the string `.12` to the `ProgID`, so we prepend a dot to the version number. When we call `StringFormat`, the `strVersion` variable is passed for the second token. If no version value is passed, `strVersion` will remain empty, and `StringFormat` will simply remove the second token.

```
Dim objShell As New IWshRuntimeLibrary.WshShell

Dim strVersion As String
Dim strArg As String
Dim strResponse As String

On Error GoTo Err_Handler:

'// If we passed a version value, create a
'// version string we can pass to StringFormat
'// (if no version value is passed, strVersion
'// will remain empty and StringFormat will
'// omit the version information
If version <> -1 Then strVersion = "." & CStr(version)

'// expand the tokens and search for the registry key
strArg = StringFormat(rkClsid, progid, _
    Nz(strVersion, vbNullString))
strResponse = objShell.RegRead(strarg)
```

We are using the `RegRead` method on the `WshShell` object because it is so convenient. You could also use the API functions introduced in Chapter 2 if you have already declared those functions for use elsewhere in your application.

If the registry key exists, we will have the `CLSID` GUID in the `strResponse` variable. Next, we want to query the second registry key, so we expand the second registry key with `StringFormat`, passing the `strResponse` variable. If this second registry key exists, `strResponse` will now have the path to the automation handler application.

```
If strResponse <> vbNullString Then
    strArg = StringFormat(rkServerPath, strResponse)
    strResponse = objShell.RegRead(strArg)
End If
```

Some automation handlers register themselves with command-line arguments. For example, Microsoft Word returns the following `LocalServer32` value on our system:

```
C:\PROGRA~1\MI1933~1\Office12\WINWORD.EXE /Automation
```

Because this function is supposed to return the path to an application, we need to remove the command-line arguments. We use the `Trim`, `Mid`, and `InStr` functions to remove any text following the first slash character.

Other applications may use a different standard for specifying command-line arguments. You may need to augment this command-line cleanup code to handle other standards depending upon how the application you are automating has registered itself.

Part III: Interacting with the Application

The cleanup block releases the `WshShell` object we instantiated, and then the cleaned path string is returned.

```
'// some servers are registered with arguments
'// (/automation, etc); remove the arguments
If InStr(strResponse, "/") > 0 Then
    strResponse = Trim$(Mid$(strResponse, 1, InStr(strResponse, "/") - 1))
End If
Cleanup:
Set shell = Nothing
Terminate:
PathToApplication = strResponse
Exit Function
```

Finally, the error handler catches any errors. We specifically trap the error returned by `RegRead` if a registry key does not exist. In this case, we fail silently, and return `vbNullString` to indicate that no automation handler is registered.

```
Err_Handler:
Select Case Err.Number
Case ERR_REG_READ:
    PathToApplication = vbNullString
    strResponse = vbNullString
    Resume Cleanup
Case Else
    MsgBox Err.Description, vbCritical, Err.Number
End Select

strResponse = vbNullString

End Function
```

This function accurately reports the registered path for the Automation handler application, but it doesn't verify that the file specified actually exists. The `ApplicationInstalled` function tests the path returned by `PathToApplication` and returns `True` if the application file exists in the specified location. It uses the `FileExists` method on the `FileSystemObject` to verify the existence of the file.

```
Public Function ApplicationInstalled(progid As String, _
Optional version As Long = 0) As Boolean
Dim fso As New IWshRuntimeLibrary.FileSystemObject

ApplicationInstalled = _
fso.FileExists(PathToApplication(progid, version))

Cleanup:
Set fso = Nothing

End Function
```

Word: Creating a Formatted Letter with Data

The Microsoft Word Object Library is used to automate the creation and editing of documents. The object model (OM) can be used to create documents and templates, add new content, and edit existing text, as well as automate all aspects of content formatting.

Chapter 10: Using Automation to Add Functionality

In this section, we look at how you can create a report document in Word from data in an Access database. We will create a new document that contains an order summary for each customer in a query. We will list the order data, invoice number, and invoice total for each order in a query. The query consolidates all of the information we will need to create the order summaries, and orders the data so that our code can walk the resulting recordset and generate the word document in the proper order. The resulting word document will have one or more pages for each company in our query.

To automate Word with early binding, you need to add a reference to the Microsoft Word Object Library. The library is registered by default when Word is installed, and appears in the reference list as Microsoft Word 12.0 Object Library — the version number may be different depending upon which version(s) of Office you have installed. You may also browse to the library file, which is located at %programfiles-dir%\Microsoft Office\Office 12\msword.olb.

Complete documentation for the Microsoft Word object model is available online from the MSDN website at <http://msdn.microsoft.com>. Search MSDN for “Microsoft Word object model reference.”

This scenario uses data from the Northwind.mdb database file that was available in previous versions of Access. If you do not have this file, it can be downloaded from <http://download.microsoft.com>. Create a new query in the database, named WordOrderSummaryData, using the following SQL:

```
SELECT Customers.CustomerID,
       CompanyName,
       [Address] & Chr$(13) & Chr$(10) &
       [city] & " " & [Region] & " " &
       [postalcode] & Chr$(13) & Chr$(10) &
       [country] AS FullAddress,
       Orders.OrderDate,
       DateAdd("d",[OrderDate],14) AS DueDate,
       Orders.OrderID,
       Sum([unitprice]*[quantity]*(1-[discount]))+[Freight] AS InvoiceTotal
  FROM [Order Details] INNER JOIN
       (Orders INNER JOIN Customers ON
          Orders.CustomerID = Customers.CustomerID) ON
          [Order Details].OrderID = Orders.OrderID
 WHERE (((Customers.Region)="ca" Or (Customers.Region)="mt" Or (Customers.Region)="or"))
 GROUP BY Customers.CustomerID,
          Customers.CompanyName,
          [Address] & Chr$(13) & Chr$(10) &
          [city] & " " & [Region] & " " &
          [postalcode] & Chr$(13) & Chr$(10) & [country],
          Orders.OrderDate,
          DateAdd("d",[OrderDate],14),
          Orders.OrderID,
          Orders.Freight,
          Customers.Country,
          Customers.PostalCode
 ORDER BY Customers.Country DESC,
          Customers.PostalCode DESC;
```

This query combines customer data from the Customer table, with order data from the Orders table, and line item detail information from the Order Details table. It calculates the total dollar amount for each order, and builds a single, formatted FullAddress field that we will use to print a mailing address on each report.

Part III: Interacting with the Application

To make this demonstration more manageable, we've included a WHERE clause to restrict the number of rows returned.

We will create seven functions to support this scenario, as shown in the table that follows. Create a new standard module in the northwind.mdb database to add these functions.

Function	Description
CreateOrderSummaryDocument	Creates a Word document. Add a single page for each customer in a query listing a summary of all orders by the customer.
AddDocumentHeaderPage	Adds a header page to a selection.
ForceNewPage	Adds a page break to a selection.
AddLetterHead	Adds the Letterhead content to a selection.
AddAddress	Adds the customer address information to a selection.
AddDetailHeader	Add a header object for the detail block to a selection.
AddDetailItem	Add an invoice line item to the detail block to a selection.

The `CreateOrderSummaryDocument` procedure drives the whole process. The procedure will create a recordset based on the `WordInvoiceData` query we created earlier. It then instantiates Microsoft Word, creates a new document, and creates a header page.

Then, it enumerates each row in the recordset, starting a new page with a letterhead and customer address data every time a new customer is reached. Each invoice for the customer is added to the address block. Let's look at the implementation of this procedure in detail.

The procedure creates a recordset against the query we created earlier in this section. If the recordset is empty, there is nothing to create so we terminate with a message.

```
Public Sub CreateOrderSummaryDocument()
    Dim rs As DAO.Recordset
    Dim objWord As Word.Application
    Dim objDoc As Word.Document

    Dim idCustomer As String

    On Error GoTo Err_Handler

    ' // Ensure there is data to process
    Set rs = CurrentDb.OpenRecordset( _
        "WordOrderSummaryData", dbOpenDynaset)
    If rs.EOF Then
        MsgBox "The recordset is empty. " & _
            "No document will be created."
        Resume cleanup
    End If
```

Chapter 10: Using Automation to Add Functionality

The `idCustomer` variable that tracks the current customer is initialized as an empty string. This ensures that the header data for the first customer is created correctly (we will see why shortly). An instance of Word is created and made visible so that we can watch the document created.

Then, a new document is created by calling the `Add` method on the `Documents` collection. The return value of the `Add` method is cached in the `objDoc` object variable. The document is selected by calling the `Select` method on the `Document` object.

```
'// initialize the first customer id
idCustomer = vbNullString

'// Instantiate Microsoft Word
Set objWord = New Word.Application
objWord.Visible = True

'// Create a new document
Set objDoc = objWord.Documents.Add
objDoc.Select
```

Once the document has been created and selected, we call the `AddDocumentHeaderPage` method to add a header page with the name of the report. We pass a reference to the `Word.Application.Selection` object, which we use to access the document surface. Each one of our helper functions accepts a `Selection` object argument to tell the function where to add its content.

```
'// Add document header page
AddDocumentHeaderPage objWord.Selection
```

After we create the header page, we enter a loop that enumerates through each row in the recordset. Inside the loop, we look at the `CustomerID` of the current row, and compare it to the value of the `CustomerID` of the previous row, which is stored in the `idCustomer` string variable. The first time through the loop, the current `CustomerID` value will not match `idCustomer` because we initialized `idCustomer` to `vbNullString`.

If the current `CustomerID` does not match the value stored in `idCustomer`, we must be on the first row for a new customer (this assumes that the recordset is sorted correctly and that all orders for a customer appear together in the recordset). In this case, we call `ForceNewPage` to start a new page, and then store the new `CustomerID` into the `idCustomer` variable.

```
Do While Not rs.EOF
    With objWord
        If idCustomer <> rs("CustomerID") Then
            '// The current customerID doesn't match
            '// the previous one, force a new page and
            '// cache the new current customerID
            ForceNewPage .Selection
            idCustomer = rs("CustomerID")
```

Next, we add the letterhead content to the new page by calling the `AddLetterHead` function. Then we pass a reference to our recordset to the `AddAddress` function. This function reads the address data for the current row in the recordset, and adds that information to the page.

Part III: Interacting with the Application

At this point, we are ready to write the detail line item data to the page.

```
    AddLetterHead .Selection  
    AddAddress .Selection, rs  
    AddDetailHeader .Selection  
End If
```

The detailed line item data is added to the page by calling the `AddDetailItem` function, passing the `Selection` object and a reference to our recordset. `AddDetailItem` will read the line item detail information for the current row and add it to the document.

Finally, we call the `MoveNext` method on the recordset to process the next row.

```
'// Write the detail  
AddDetailItem .Selection, rs  
  
End With  
  
'// Process the next letter  
rs.MoveNext  
  
Loop
```

Our cleanup method closes the recordset and then releases the object variables we instantiated. The error handler is a generic error handler that reports every error and does not trap any specific errors.

```
cleanup:  
On Error Resume Next  
rs.Close  
Set rs = Nothing  
Set objDoc = Nothing  
Set objWord = Nothing  
  
Terminate:  
On Error Goto 0  
Exit Sub  
  
Err_Handler:  
Select Case Err.Number  
Case Else  
    MsgBox Err.Description, vbCritical, Err.Number  
End Select  
Err.Clear  
Resume cleanup  
  
End Sub
```

Now that we have the framework prepared, let's create the functions that actually write content into the document. We'll work from the top down, beginning with the `AddDocumentHeader` procedure, which adds the following text to the beginning of our document:

```
Northwind Traders  
Order Summary Report
```

Chapter 10: Using Automation to Add Functionality

The procedure takes a `Word.Selection` object argument. We will follow a typical pattern in all of the procedures that write content to the document. We create a string that is the content we want to write to the document. Then we set font properties (font face, size, color, bold, and underline state) on the `Selection` object's `Font` object and paragraph properties (alignment, and paragraph padding) on the `ParagraphFormat` object. Finally, we'll call the `Selection` object's `TypeText` method to insert the formatted text into the document at the current location. Let's look at the implementation in detail.

We set a `String` variable with the actual content we want to write to the document. We separate the two lines by a `vbVerticalTab` character (`Chr(11)`). Word uses a `vbVerticalTab` to indicate a manual line break, which is a line break that does not separate paragraphs.

```
Private Sub AddDocumentHeaderPage(target As Word.Selection)

    Dim content As String

    content = "Northwind Traders" & vbVerticalTab & _
              "Order Summary Report"
```

Next, we set the font properties. In the case of the `AddDocumentHeader` function, we want 24pt Arial, Bold, with no underlining and the default color.

```
With target
    With .Font
        .Name = "Arial"
        .Size = 24
        .Bold = True
        .Color = wdColorAutomatic
        .Underline = wdUnderlineNone
    End With
```

Next, we set the paragraph format properties. In this case, we want the paragraph to be centered, with 72 points of space above and below the text.

```
With .ParagraphFormat
    .Alignment = wdAlignParagraphCenter
    .SpaceBefore = 72
    .SpaceAfter = 72
End With
```

Finally, we call the `TypeText` method on the `Selection` object to insert the formatted text into the document.

```
.TypeText content
End With

End Sub
```

All of the remaining `Add*` functions follow the same pattern.

Part III: Interacting with the Application

The next procedure we encounter in our framework function is the ForceNewPage function. Word uses a vbFormFeed character to signal a page break. We simply send a vbFormFeed to the document using the TypeText method.

```
Private Sub ForceNewPage(target As Word.Selection)
    target.TypeText vbFormFeed
End Sub
```

The AddLetterhead procedure adds the return address information to the beginning of each customer page, as shown in the following text:

```
Northwind Traders
1 Oak Street
Seattle WA 98015
+1 (206) 555-5555
O R D E R     S U M M A R Y
```

Just as with the AddDocumentHeader procedure, we'll follow the standard pattern: Create a String variable with the content, set formatting properties, and then write the formatted content to the document using the TypeText method.

Here, we set the content string to Northwind Traders, set the font properties (14pt Arial Bold, no underline, standard color), and paragraph properties (centered, no extra space before and after the paragraph). Then we add the content to the document.

```
Private Sub AddLetterHead(target As Word.Selection)

    Dim content As String
    content = "Northwind Traders" & vbCrLf

    With target
        With .Font
            .Name = "Arial"
            .Size = 14
            .Bold = True
            .Color = wdColorAutomatic
            .Underline = wdUnderlineNone
        End With

        With .ParagraphFormat
            .Alignment = wdAlignParagraphCenter
            .SpaceBefore = 0
            .SpaceAfter = 0
        End With
        .TypeText content
    End With

```

Because we want the return address to be formatted differently from the company name, we reset the content string variable and set the new font properties for the address block. We need to set only the properties that have changed. In this case, the address will be formatted in 10pt Arial, non-bolded, still with no underline and using the default color. Because we are not setting paragraph properties,

Chapter 10: Using Automation to Add Functionality

we will retain the previous paragraph formatting (centered, no extra padding before and after the paragraph). Again, we pass the `content` variable to the `TypeText` method to add the formatted content to the document.

```
content = "1 Oak Street" & vbVerticalTab & _
    "Seattle WA 98015" & vbVerticalTab & _
    vbVerticalTab & "+1 (206) 555-5555" & vbCrLf
With .Font
    .Size = 10
    .Bold = False
End With

.TypeText content
```

The `TypeText` method accepts any string content. It is not required to pass a variable; you can pass string literals as well. In the preceding code, we change the font size to 18pt, and set the paragraph spacing to 36pt before and after, and then pass a string literal to the `TypeText` method. The resulting text will be formatted as 18pt Arial, no bold, no underline, the default color, centered, with 36 points of extra space before and after the paragraph.

```
.Font.Size = 18
.ParagraphFormat.SpaceBefore = 36
.ParagraphFormat.SpaceAfter = 36
.TypeText "O R D E R     S U M M A R Y" & vbCrLf
End With
End Sub
```

Our next helper procedure, `AddAddress`, takes a `Recordset` argument as well as the `Selection object` argument. *This is the first helper procedure that is data-bound — that is, the content generated by this routine will change depending upon the data contained in the current row of the recordset.*

Following the pattern, we set the `content` variable with the text we want to appear in the document. Unlike previous cases, however, instead of setting a static text string, we are reading content from the recordset. The value of the `CompanyName` field in the recordset is concatenated to the `FullAddress` field from the recordset, separated by a `vbCrLf` character. We then use the `Replace` VBA function to transform the `vbCrLf` characters to `vbVerticalTab` characters. We do this because we want the address data to appear as a single paragraph in Word, separated by manual line breaks. As mentioned earlier, Word uses the `vbVerticalTab` character to represent manual line breaks. Finally we add a `vbCrLf` character at the end to terminate the paragraph.

```
Private Sub AddAddress(target As Word.Selection, rs As DAO.Recordset)
    Dim content As String

    content = rs("CompanyName") & vbCrLf & _
        rs("FullAddress")
    content = Replace(content, vbCrLf, vbVerticalTab) & _
        vbCrLf
```

Part III: Interacting with the Application

Next, we set the font and paragraph properties and write the content to the document by calling the `TypeText` method.

```
With target
    With .Font
        .Name = "Courier New"
        .Size = 12
        .Bold = False
        .Color = wdColorAutomatic
        .Underline = wdUnderlineNone
    End With

    With .ParagraphFormat
        .SpaceBefore = 48
        .SpaceAfter = 48
        .Alignment = wdAlignParagraphLeft
    End With
    .TypeText content
End With
End Sub
```

The `AddDetailHeader` procedure simply writes three static text labels separated by `vbTab` characters. This function could be enhanced to create a three-column table, setting the labels into cells in the first row of the table.

```
Private Sub AddDetailHeader(target As Word.Selection)
    Dim content As String
    content = "Order Number" & vbTab & "Order Date" & _
              vbTab & "Invoice Total" & vbCrLf

    With target
        With .Font
            .Name = "Times New Roman"
            .Size = 12
            .Bold = True
            .Color = wdColorAutomatic
            .Underline = wdUnderlineNone
        End With

        With .ParagraphFormat
            .Alignment = wdAlignParagraphCenter
            .SpaceBefore = 0
            .SpaceAfter = 0
        End With

        .TypeText content
    End With
End Sub
```

The `AddDetailItem` procedure, like the `AddAddress` procedure, accepts a `DAO.Recordset` object so that we can access the line item detail information for the current row in the recordset.

Chapter 10: Using Automation to Add Functionality

Here, we use a nifty capability of the `Mid` function to replace an arbitrary substring of text. We define a fixed-length string, 40 characters in length. Then, we write the formatted values of the `OrderID`, `OrderDate`, and `InvoiceTotal` fields from the recordset into the string at specific locations.

```
Private Sub AddDetailItem(target As Word.Selection, rs As DAO.Recordset)
    Dim content As String * 40
    content = String(40, " ")
    
    Mid$(content, 1, 7) = rs("OrderID")
    Mid$(content, 10, 11) = _
        Format(rs("OrderDate"), "dd MMM yyyy")
    Mid$(content, 27, 10) = _
        Format(rs("InvoiceTotal"), "$#,##0.00")
```

Following our standard pattern, we set the font and paragraph properties, and then write the content to the document using the `TypeText` method.

```
With target
    With .Font
        .Name = "Courier New"
        .Size = 10
        .Bold = False
        .Color = wdColorAutomatic
        .Underline = wdUnderlineNone
    End With

    With .ParagraphFormat
        .Alignment = wdAlignParagraphCenter
        .SpaceBefore = 0
        .SpaceAfter = 0
    End With
    .TypeText content & vbCrLf
End With
End Sub
```

And that completes our inspection of the code for the Word Automation scenario. Time to give it a try! Run the framework function from the Immediate window. A Word document created from Access Data is shown in Figure 10-6.

Excel: Creating an Excel Chart with Data

The Microsoft Excel Object Library provides complete automation support for this popular spreadsheet application. The object model can be used to create workbooks and worksheets; add, manipulate, and remove data; apply formatting; create pivot tables and charts; and much, much more.

In this section, we will look at how to create a chart in Excel from data in an Access database. We will create a new workbook, copy data from an Access query into the new workbook, and then create a chart in the workbook.

Part III: Interacting with the Application



Figure 10-6

To automate Excel with early binding, you need to add a reference to the Microsoft Excel Object Library. The library is registered by default when Excel is installed, and appears in the reference list as Microsoft Excel 12.0 Object Library (the version number may be different depending upon which version of Office you have installed). You may also browse to the library file, which is located at %programfilesdir%\Microsoft Office\Office 12\excel.exe by default. Unlike most of the other object model libraries we've looked at, the object model type library for Excel is compiled right into the Excel executable file. There is no standalone .tlb or .olb file for the Excel object model.

Complete documentation for the Microsoft Excel Object Model is available online from the MSDN website at <http://msdn.microsoft.com>. Search MSDN for "Microsoft Excel object model reference."

This scenario uses data from the Northwind.mdb database file, which was available with previous versions of Access. Create a new standard module in the northwind.mdb database. We will add two functions to support this scenario, as shown in the table that follows.

Function	Description
<pre>TransferRecordsetToWorksheet() Excel.Worksheet,</pre> <pre>DAO.Recordset</pre> <pre>[, startRow]</pre> <pre>[, startCol]</pre>	

Chapter 10: Using Automation to Add Functionality

Function	Description
[, rangeName]	
[, rangeTitle])	Transfers data from a DAO recordset to an Excel worksheet object. Optional arguments control where the data appears in the sheet, and how the data is named and titled.
CreateChartObject(
Excel.Worksheet,	
rangeName)	Creates a chart based on data in an Excel worksheet.

Let's look at the `TransferRecordsetToWorksheet` function first. This function accepts an `Excel.Worksheet` object and a `DAO.Recordset` object as required arguments. It will copy all of the data from the recordset to the worksheet. It creates a header row with the names of each of the fields. By passing optional arguments, the placement of the data within the sheet can be controlled, and a named range can be applied to the data (this capability will be very useful when we create the chart).

We create a format string that will be used to create an R1C1 range reference. When we create a named range, we need to specify which range of cells is included. This range can be specified in one of several ways; because our code will have individual variables to track a row index and a column index, the R1C1 notation will be the easiest to implement. For example, if we have four rows of data of three columns each, we could specify an R1C1 reference as follows:

```
=R1C1:R4C3
```

The string constant will be passed to our `StringFormat` function, along with our row and column index variables to generate the correct R1C1 range reference for our data.

```
Sub TransferRecordsetToWorksheet( _
    sheet As Excel.Worksheet, _
    rs As DAO.Recordset, _
    Optional startRow As Long = 1, _
    Optional startCol As Long = 1, _
    Optional rangeName As String = vbNullString, _
    Optional rangeTitle As String = vbNullString)

    Const R1C1_RANGE_FORMAT As String = _
        "=R{0}C{1}:R{2}C{3}"
```

We declare some variables, enable error checking, and then cache the column count of the recordset. We cache this value because it will be used several times throughout the code, and it is faster and more efficient to read the property value once and save the value into a local variable.

```
Dim idxField As Long
Dim idxRow As Long
Dim cCols As Long
```

Part III: Interacting with the Application

```
Dim strRangeName As String  
  
On Error GoTo Err_Handler  
  
'// Cache the column count  
cCols = rs.Fields.Count
```

We use the `idxRow` variable to track where we will write data to in the worksheet. Think of it as a cursor pointing to the current row. As we move through the worksheet, `idxRow` will point to the row we are working with. This variable is initialized to the value of the `startRow` argument passed to the function.

```
'// write the rangeTitle  
idxRow = startRow  
sheet.Cells(idxRow, startCol).Value = _  
    Nz(rangeTitle, vbNullString)  
idxRow = idxRow + 1
```

The `Cells` collection of the `Excel.Worksheet` object provides access to each cell in the sheet. We use a row and column index argument to the collection to request a specific cell. For example,

```
sheet.Cells(5,14)
```

returns a reference to the cell at row 5, column 14, also known as cell N1. The `Value` property returns the content of the cell, and is a read/write property. Here, we write the value of the `rangeTitle` argument (or an empty string if no value is passed) to the cell on the first row (because `idxRow` here is equal to `startRow`) and the first column (because the `startCol` argument defines the first column for our output).

Next, we increment `idxRow` so that the next operation will occur on the next row.

To create the header row, we loop through each field in the recordset's `Fields` collection, copying the name into the appropriate column. Each time through this loop, the `idxRow` value remains constant — 2 — while the column argument uses the loop counter (`idxField`) to increment by one. Once all the field names have been written to the header row, we increment `idxRow` by one to move the row pointer to the next row.

```
'// Create the header row  
For idxField = 0 To cCols - 1  
    sheet.Cells(idxRow, startCol + idxField).Value = _  
        rs.Fields(idxField).Name  
Next idxField  
idxRow = idxRow + 1
```

This loop does the bulk of the work. We enumerate each row of the recordset, and within each row, we enumerate each column of the recordset. The `idxField` variable is used in the inner loop to step through each column of the recordset. Just as when we built the header row, here we use the `idxField` variable to increment the column index into the `Cells` collection. The only difference is that instead of writing the `Name` property of the field item, we write the value of the field of the current row of the recordset.

```
'// enumerate the recordset transferring each cell to the worksheet  
Do While Not rs.EOF  
    For idxField = 0 To cCols - 1  
        sheet.Cells(idxRow, startCol + idxField).Value_  
            = rs(idxField)  
    Next idxField
```

Chapter 10: Using Automation to Add Functionality

Each time we move to the next row of the recordset, we also increment `idxRow`. We also call `DoEvents` to ensure that our system remains responsive in case we have a large recordset with many cells to copy. Also, by calling `DoEvents`, we allow Excel to update the display so we can watch the data being copied to the new sheet.

```
    rs.MoveNext: idxRow = idxRow + 1
    DoEvents
Loop
```

After all the data is copied to the worksheet, we check to see if a range name was passed to the function. If it was, we calculate the R1C1 range reference for our data, and use the `Add` method of the worksheet's `Names` collection to add a new named range.

```
'// if a range name is specified, build an R1C1 range reference
'// and add the named range
If vbNullString <> rangeName Then
    strRangeName = StringFormat(R1C1_RANGE_FORMAT, _
        startRow + 1, _
        startCol, _
        idxRow - 1, _
        cCols + startCol - 1)
    sheet.Names.Add rangeName, RefersToR1C1:=strRangeName, Visible:=True
End If
```

To calculate the R1C1 range reference, we need the address of the cell at the upper-left of the range, and the address for the lower-right cell. The upper-left cell appears on the second row of the range (the first row contains the range title; we do want to include the header row inside our range) in the first column. Therefore, our upper-left cell is at `startRow+1` (the second row) and `startCol` (the first column).

The lower-right cell is on the last row in the last column. Because we incremented `idxRow` by one right before we left the recordset loop, our last row is really `idxRow-1`. The last column is the number of columns (`cCols`) + the value of `startCol` decremented by 1 (`cCols + startCol - 1`).

Once we've calculated the four data points for the R1C1 reference, we pass them to the `StringFormat` function and replace the tokens in the `R1C1_RANGE_FORMAT` string constant we declared earlier.

Because we didn't actually instantiate any objects inside this function (the `Worksheet` and `Recordset` objects were both passed in as arguments), we don't need a cleanup block. We have the standard `Terminate` block, and our error handler is a generic handler added for robustness to catch all errors.

```
Terminate:
On Error Goto 0
Exit Sub

Err_Handler:
Select Case Err.Number
Case Else
    MsgBox Err.Description, vbCritical, Err.Number
End Select

Err.Clear
Resume Terminate

End Sub
```

Part III: Interacting with the Application

The `CreateChartObject` function accepts a `Worksheet` object that contains the data we want to chart, and the name of the range that contains the data. The function creates a new `Excel.Chart` object by calling the `Add` method on the `Charts` collection. It then sets the `SetSourceData` property on the `Chart` object, passing the range name, to identify the data for the chart. It sets additional properties on the `Chart` object to control formatting and other attributes of the chart.

```
Private Sub CreateChartObject(dataSheet As Excel.Worksheet, rangeName As String)
    Dim objChart As Excel.Chart

    On Error Resume Next
    Set objChart = dataSheet.Application.Charts.Add
    With objChart
        .Name = "Chart created by Microsoft Access"
        .ChartType = XlChartType.xl3DColumn
        .SetSourceData _
            dataSheet.Names(rangeName).RefersToRange
        .HasTitle = True
    End With

    Cleanup:
    Set objChart = Nothing

    Terminate:
    On Error GoTo 0
    Exit Sub

    Err_Handler:
    Select Case Err.Number
        Case Else
            MsgBox Err.Description, vbCritical, Err.Number
    End Select

    Err.Clear
    Resume Cleanup
End Sub
```

Now that we have a way to get data out of Access and into an Excel worksheet, and a way to create a chart from data in a worksheet, let's create a procedure to test the complete scenario. This procedure will create a new Excel workbook, use the `TransferRecordsetToWorksheet` function to copy a data from Access to the workbook, and then call `CreateChartObject` to create a new chart based on the data we copied.

We start by instantiating a new instance of the Excel application and set the `Visible` property according to the `fVisible` argument passed to this procedure. If `fVisible` is true, we can watch Excel create the sheet, add the data, and create the chart. If `fVisible` is false, the sheet will be created in the background with no visible activity.

```
Sub CreateWorksheetWithChart(fileName As String, _
    rs As Recordset, Optional fVisible As Boolean = False)
    Dim objExcel As Excel.Application
    Dim objxls As Excel.Workbook
    Dim objDataSheet As Excel.Worksheet
    Dim objChart As Excel.Chart
```

Chapter 10: Using Automation to Add Functionality

```
Dim col As Long
Dim row As Long

Const RANGE_NAME As String = "DataFromMicrosoftAccess"

On Error GoTo Err_Handler
'// Instantiate Excel
Set objExcel = New Excel.Application
objExcel.Visible = fVisible
```

The SheetsInNewWorkbook property is set to 1 so that when we create a new Workbook, it will contain only one sheet. Then, the Add method of the Workbooks collection is called to create a new Workbook object. The Add method returns a reference to the new workbook, which we cache in a variable. We will use this variable to get a reference to the contained worksheet, and when we save the workbook after we add the data and create the chart.

```
'// use the add method to create a new empty workbook
'// and a new chart object
objExcel.SheetsInNewWorkbook = 1
Set objxls = objExcel.Workbooks.Add
```

We call the TransferRecordsetToWorksheet function, passing a reference to the first (and so far, only) worksheet in our workbook, the recordset, and the range name and title. We leave the startRow and startCol arguments blank, which means we will start creating our data in cell A1 (row 1, column 1).

```
'// Copy sample data to a sheet in the workbook
TransferRecordsetToWorksheet objxls.Worksheets(1), _
    rs, , , RANGE_NAME, "Chart Title"
```

Once the data has been transferred to the worksheet, we call the CreateChartObject function, again passing a reference to the worksheet, and the name of the range we created.

```
'// Create the chart object
CreateChartObject objxls.Worksheets(1), RANGE_NAME
```

We do a lot of work in our Cleanup block. First, we save our workbook, using the file name we passed as an argument. We indicate that we want to use the default file format.

```
Cleanup:
On Error Resume Next

'// Save our worksheet
objxls.SaveAs fileName, XlFileFormat.xlWorkbookDefault
```

If fVisible is False, everything happens in the background. In this case, we want to close the workbook, and then close Excel. So we close the workbook we created. Then we check for any other workbooks that may be open and close each of them, indicating that we do want to save any changes that may be unsaved. Finally, we call the Quit method on the Excel Application object to shut down the Excel application.

```
If Not fVisible Then
    objxls.Close
```

Part III: Interacting with the Application

```
'// Check for any additional open workbooks
If objExcel.Workbooks.Count > 0 Then
    MsgBox "wow! extra books open!"
    For Each objxls In objExcel.Workbooks
        objxls.Close True
    Next objxls
End If

'// Shutdown the Excel application
objExcel.Quit
End If
```

In every case, we release the object variables that tracked the workbook we created and the Excel application we instantiated.

```
Set objxls = Nothing
Set objExcel = Nothing
```

We have a standard Terminate block and a generic error handler that reports any error without trapping any specific error codes.

```
Terminate:
On Error GoTo 0
Exit Sub

Err_Handler:
Select Case Err.Number
Case Else
    MsgBox Err.Description, vbCritical, Err.Number
End Select

Err.Clear
Resume Cleanup

End Sub
```

We now have all the pieces in place to test our scenario. Let's create a test harness procedure that creates a recordset that we can pass to the CreateWorksheetWithChart function.

This procedure merely creates a recordset against data in the Northwind database, and then calls our CreateWorksheetWithChart function, passing a file name, the recordset, and the visibility value. You can change the SQL to any valid SQL query, and see the data and chart change accordingly.

```
Sub Harness(Optional fVisible As Boolean = True)

Dim strSQL As String
strSQL = "SELECT " & _
    "Choose(ShipVia, ""UPS"", ""FedEx"""", ""DHL""")" &
    "_As [Shipper], Count(OrderID) as [Order Count]" &_
    "FROM Orders GROUP BY ShipVia"

Dim rs As DAO.Recordset
```

Chapter 10: Using Automation to Add Functionality

```
Set rs = CurrentDb.OpenRecordset(strSQL)

CreateWorksheetWithChart "c:\test.xlsx", rs, fVisible
If Not rs Is Nothing Then
    rs.Close
    Set rs = Nothing
End If
End Sub
```

An Excel chart from Access Data is shown in Figure 10-7.

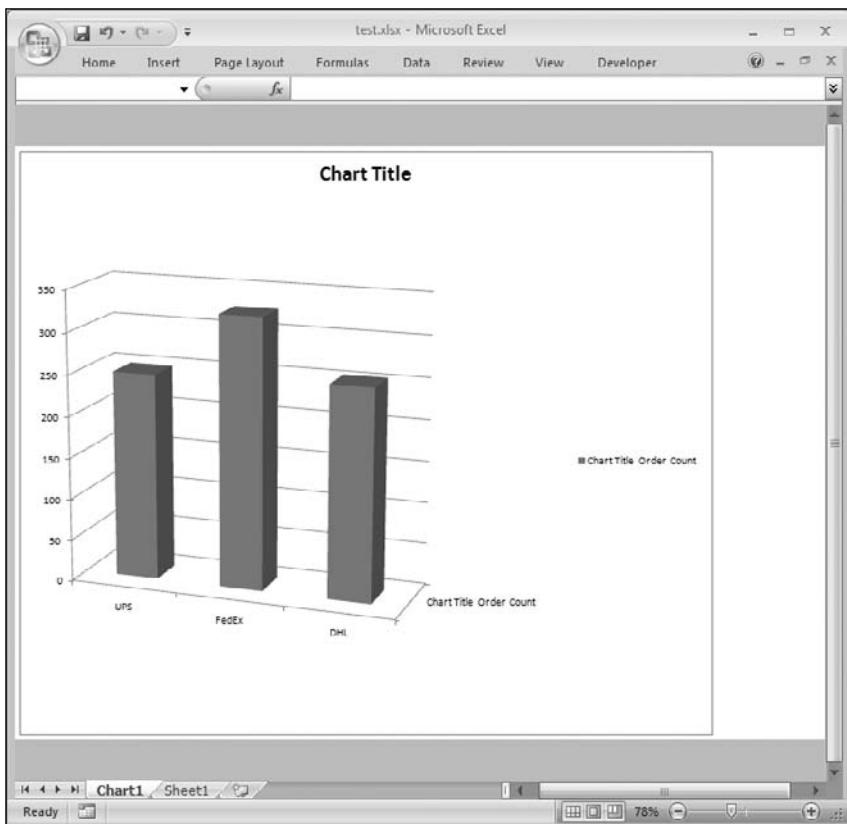


Figure 10-7

Outlook: Create Appointments from an Events Database

The Microsoft Outlook Object Library exposes a very rich object model for manipulating the wide variety of e-mail and calendar objects available in Microsoft Outlook. We will look at two scenarios: how to create appointments in Outlook, and how to bring mail items into an Access database using the Outlook object model.

Part III: Interacting with the Application

To automate Outlook with early binding, you need to add a reference to the Microsoft Outlook Object Library. The library is registered by default when Outlook is installed, and appears in the reference list as "Microsoft Outlook 12.0 Object Library." You may also browse to the library file, which is located at %programfilesdir%\Microsoft\Office\Office 12\msoutl.olb.

Complete documentation for the Microsoft Outlook Object Model is available online from the MSDN Web site at <http://msdn.microsoft.com>. Search MSDN for "Microsoft Outlook object model reference."

In our first Outlook scenario, we have an Access database that tracks a variety of events. The database was created from the Events template in Access 2007, and contains basic event data, and a simple UI to enter and edit event records. Now we would like to extend the application to create appointments in an Outlook calendar for each of the events in our database. We will do this by adding a button to the Event entry form that, when pressed, will create an appointment using values from the record displayed on the form.

To prepare for this scenario, create a new database based on the Events template.

You may also download the OutlookEvents database from this books' Web site at www.wrox.com.

The resulting database will have a form named "Event Details." Open this form in design view, and add a button named cmdCreateAppointment. We will use this button to create the appointment in our Outlook calendar.

Add the following code to the button's event handler.

```
Private Sub cmdCreateAppointment_Click()
    Dim idAppt As String
    idAppt = CreateOutlookAppointment( Me.Title, _
        Me.Location, Me.Start_Time, Me.End_Time, _
        "Appointment created by the EventDetails form", _
        "Automation;WroxBook;Outlook")
End Sub
```

This code calls a function named CreateOutlookAppointment, passing in the Title, Location, Start_Time, and End_Time values from the form fields. We'll create this routine in a moment. It also passes some text that will be used for the body of the appointment, and three categories that will be assigned to the appointments.

Next, add a standard module to your application and add the following code:

```
Option Compare Database
Option Explicit
Public Const DEFAULT_BODY As String = _
    "Appointment created by Access database '{0}'"
Public Const DEFAULT_CATEGORY_LIST As String = "OutlookAutomationDemo"
```

These constants are used to provide default values for the appointment body and category list if values are not passed to the function. If your application does not require these items, you may omit the constants and modify the code to remove any reference to them.

Now, we'll look at the CreateOutlookAppointment function. This function creates a new appointment in the calendars folder of the default profile for the user running this code. The function accepts a subject

Chapter 10: Using Automation to Add Functionality

(which appears as the title of the appointment), a location, and the start and end time. It also accepts optional arguments for the body text and category list. If either of these optional arguments is present, it overrides the default values defined in the constant strings we just declared.

Objects are declared to hold references to the Outlook application and the appointment that we will be creating. We test each of the optional arguments, and if they do not contain any values, the default values are copied into the argument variables.

```
Public Function CreateOutlookAppointment( _
    subj As String, Location As String, _
    starttime As Date, endtime As Date, _
    Optional body As String, Optional category As String) _ As String

    Dim objOutlook As Outlook.Application
    Dim objOlAppt As Outlook.AppointmentItem
    Dim idAppointment As String

    On Error GoTo Err_Handler

    ' // process optional arguments
    If vbNullString = Nz(body) Then body = _
        StringFormat(DEFAULT_BODY, CurrentDb.Name)
    If vbNullString = Nz(category) Then
        category = DEFAULT_CATEGORY_LIST
    End If
```

We instantiate Outlook using the `New` command, and then create a new appointment using the `CreateItem` method on the `Outlook.Application` object.

```
' // Instantiate Outlook application and
' // appointment object
Set objOutlook = New Outlook.Application
Set objOlAppt = _
    objOutlook.CreateItem(olAppointmentItem)
```

With the appointment item that we created, we set several properties. There are many additional properties available to you; consult the online document or the object browser for a complete list.

```
' // Set properties on the appointment
With objOlAppt
    .Subject = subj
    .start = starttime
    .End = endtime
    .body = body
    .BusyStatus = olBusy
    .Categories = category
    .Location = Location
    .ReminderMinutesBeforeStart = 1440
    .Sensitivity = olNormal
```

Part III: Interacting with the Application

Once all the properties have been set, we call the `Save` method on the appointment item. This causes the appointment to be created in the default calendar folder. Until you call the `Save` method, nothing will appear in the Outlook client.

```
.Save  
  
CreateOutlookAppointment = .GlobalAppointmentID  
End With
```

Finally, we add our standard `Cleanup` and `Terminate` blocks. The object variables are released, and the error handler provides a generic message box for all errors.`Cleanup`:

```
'// Free instantiated objects  
Set objOlAppt = Nothing  
Set objOutlook = Nothing  
  
Terminate:  
On Error GoTo 0  
Exit Function  
  
Err_Handler:  
Select Case Err.Number  
Case Else  
    MsgBox Err.Description, vbCritical, Err.Number  
End Select  
Err.Clear  
Resume cleanup  
End Function
```

To exercise this code, open the form in browse mode. Create a new event. Add data to each of the fields and save the record. Then, click the button to create the appointment in Outlook, and open the Outlook calendar to see your new item.

There is an issue with an Outlook 2007 mailbox that is joined to a domain to an Exchange e-mail account. If the Outlook client is not running in the background, creation of the appointment fails silently. To work around this issue, open Outlook before you create appointments. This problem does not reproduce if Outlook is configured to work with another e-mail account, such as a Hotmail or POP e-mail account.

Automating Internet Explorer

To automate Internet Explorer using early binding, you must add a reference to the Microsoft Internet Controls library. If this reference does not appear in the References dialog box, you may browse to `shdocvw.dll` in the `system32` folder of your Windows directory.

Opening a URL in a New Tab in IE7

Internet Explorer version 7 adds the ability to view Web pages in multiple tabs in the application. This is a very convenient feature that allows you to reduce the number of windows that are taken up by the browser. If you are automating Internet Explorer, you might want to take advantage of this new feature.

Chapter 10: Using Automation to Add Functionality

To do so, you'll need to pass an optional parameter value to the `Navigate2` method of the Internet Explorer Application object, as shown in the following code:

```
Public Enum BrowserNavConstants
    navOpenInNewWindow = &H1
    navNoHistory = &H2
    navNoReadFromCache = &H4
    navNoWriteToCache = &H8
    navAllowAutosearch = &H10
    navBrowserBar = &H20
    navHyperlink = &H40
    navEnforceRestricted = &H80
    navNewWindowsManaged = &H100
    navUntrustedForDownload = &H200
    navTrustedForActiveX = &H400
    navOpenInNewTab = &H800
    navOpenInBackgroundTab = &H1000
    navKeepWordWheelText = &H2000
End Enum

Sub LaunchInNewTab()
    Const SZ_URL1 As String = "http://www.utteraccess.com"
    Const SZ_URL2 As String = "http://www.live.com"

    ' launch IE
    Dim obj As SHDocVw.InternetExplorer
    Set obj = New SHDocVw.InternetExplorer

    obj.Navigate2 SZ_URL1, BrowserNavConstants.navOpenInBackgroundTab
    obj.Navigate2 SZ_URL2, BrowserNavConstants.navOpenInNewTab
End Sub
```

Common Web Queries

Many Web sites include parameters in their addresses that can be used to pass information to the Web site. For example, if you were using `www.live.com` for a Web search, you could pass the text of your search to the Web site as follows:

```
http://search.live.com/results.aspx?q=%22Expert+Access+2007+Programming%22
```

To launch Internet Explorer with this query, you simply need to concatenate the search string to the Web address. Using this technique, users can be given a choice of search providers that are stored in a table. Create a new table in design view and add the fields listed in the table that follows.

Field Name	Data Type (Size)
ID	AutoNumber
ProviderName	Text (50)
ProviderURL	Memo

Part III: Interacting with the Application

Save the table with the name `USysSearchProviders`. Add the data shown in the following table.

ID	ProviderName	ProviderURL
1	live.com	<code>http://search.live.com/results.aspx?q={0}</code>
2	Google	<code>http://www.google.com/search?q={0}</code>
3	Microsoft Knowledge Base	<code>http://support.microsoft.com/search/default.aspx?query={0}</code>

Next, create a form and add a text box and a combo box. Change the name of the text box to `txtSearch`. Change the name of the combo box to `cboProvider`. Set the following properties of the combo box.

Property Name	Property Value
Column Count	3
Column Widths	0";1";0"
Row Source	<code>tblSearchProviders</code>

Add the following code to the `AfterUpdate` event of the text box. This code requires a reference to the Microsoft Internet Controls mentioned earlier.

```
Private Sub txtSearch_AfterUpdate()
    ' do a search
    Dim objIE As SHDocVw.InternetExplorer
    Dim stUrl As String

    If (Not (IsNull(Me.txtSearch))) Then
        Set objIE = New SHDocVw.InternetExplorer

        If (Not (IsNull(Me.cboProvider))) Then
            stUrl = StringFormat( _
                Me.cboProvider.Column(2), Me.txtSearch)
        Else
            stUrl = StringFormat(DLookup( _
                "ProviderUrl", "tblSearchProviders", _
                "ID=1"), Me.txtSearch)
        End If

        ' navigate
        objIE.Navigate2 stUrl, _
            BrowserNavConstants.navOpenInNewTab
    End If
End Sub
```

To test the form, select a search provider in the combo box. Then, enter text in the text box and press Enter. The Web site for the selected provider should launch with your search.

Parsing HTML from Internet Explorer

A likely candidate of data on a page that you might be interested in is information stored in an HTML table on the page. Tables on the page are defined using the TABLE tag in HTML. Knowing this, we can use the Document Object Model (DOM) for the page to retrieve the data in a TABLE tag. For the purposes of this example, start by creating the following HTML using Notepad. Save the file on the local computer with a .htm file extension. This page defines multiple HTML tables.

```
<html>
<body>
    <!-- table1 -->
    <table id="Table1">
        <!-- heading -->
        <tr><th>Field1</th><th>Field2</th></tr>
        <!-- data -->
        <tr><td>Data1</td><td>Data2</td></tr>
        <tr><td>Data3</td><td>Data4</td></tr>
        <tr><td>Data5</td><td>Data6</td></tr>
    </table>
    <!-- table1 -->
    <table id="Table2">
        <!-- heading -->
        <tr><th>Field1</th><th>Field2</th><th>Field3</th></tr>
        <!-- data -->
        <tr><td>Data1</td><td>Data2</td><td>Data3</td></tr>
        <tr><td>Data4</td><td>Data5</td><td>Data6</td></tr>
        <tr><td>Data7</td><td>Data8</td><td>Data9</td></tr>
    </table>
</body>
</html>
```

The function we'll write will include the following functionality:

- Create the Access table if it does not exist.
- Import multiple tables as specified.
- Delete existing tables.
- Import structure only or include data.

Let's get started! Begin by creating a new standard module. Add the following code:

```
Sub ImportHTMLTables(stFileName As String, _
    fCreateTables As Boolean, _
    fStructureOnly As Boolean, _
    fDropExisting As Boolean, _
    ParamArray stTableIds() As Variant)
    On Error GoTo ImportHTMLTablesErrors

    ' other locals
    Dim varTableIds As Variant
    varTableIds = stTableIds

    ' HTML objects
```

Part III: Interacting with the Application

```
Dim objDOM As Object  
Dim objTable As Object  
Dim objIE As InternetExplorer
```

Launch an instance of Internet Explorer and navigate to the page as follows:

```
' launch IE  
Set objIE = New InternetExplorer  
  
' open the page  
objIE.Navigate2 stFileName  
Do  
    DoEvents  
Loop While objIE.ReadyState <> READYSTATE_COMPLETE
```

Add the following code to get the DOM for the page. This will provide an object model for traversing elements in the page.

```
' get the DOM for the page  
Set objDOM = objIE.Document
```

The DOM provides a collection of tags that we'll use to retrieve all the HTML tables in the page.

```
' iterate through all the tables on the page  
For Each objTable In objDOM.all.tags("TABLE")
```

Next, we are only looking for tables that have been passed to our function as a part of the ParamArray. Add the following code to look inside the array for a matching table. We'll define the ArrayContains table function shortly.

```
' check the ID of the table to see if it is  
' in the ParamArray  
If (ArrayContains(objTable.id, varTableIds)) Then
```

One of the arguments to the function is to drop the existing tables. Add the following code to do this:

```
' drop existing tables  
If (fDropExisting) Then  
    If (ObjectExists(objTable.id, acTable)) Then  
        CurrentDb.TableDefs.Delete objTable.id  
    End If  
End If
```

Next, we need to create the table if it does not exist:

```
' create table if it does not exist  
If (fCreateTables And Not _  
    ObjectExists(objTable.id, acTable)) Then  
    CreateTableFromHTMLTable objTable  
End If
```

Chapter 10: Using Automation to Add Functionality

Before adding the cleanup code, we need to import the actual data. Add the following code to import the data:

```
' if not structure only then import data
If (Not fStructureOnly) Then
    ImportHTMLTable objTable
End If
End If
Next
```

Last, add the cleanup and error handling code to exit this routine.

```
Cleanup:
If (IsArray(varTableIds)) Then
    Erase varTableIds
End If

' close IE
objIE.Quit

Set objDOM = Nothing
Set objIE = Nothing
Set objTable = Nothing
Exit Sub

ImportHTMLTablesErrors:
' page not found
If (Err = &H800704C7) Then
    MsgBox "Cannot navigate to: " & stFileName, _
        vbExclamation
    Resume Cleanup
ElseIf (Err <> 0) Then
    Debug.Assert False
    Resume
End If
End Sub
```

We added several helper functions in this routine so let's take a closer look at those, beginning with the `ArrayContains` function. This function walks the specified array to determine if an element is contained in it.

```
Private Function ArrayContains(vValue As Variant, ByRef vArray As Variant) As Boolean
Dim i As Integer

For i = LBound(vArray) To UBound(vArray)
    If (vArray(i) = vValue) Then
        ArrayContains = True
        Exit For
    End If
Next
End Function
```

Part III: Interacting with the Application

We also added an `ObjectExists` function to determine whether a particular object exists in the database. We're using an argument of type `AcObjectType`, which allows you to search for forms or reports. By using an `AccessObject` object and an `AllObjects` collection, this function will also work for ADP files in Access 2007.

```
Public Function ObjectExists(stObjName As String, _ lngObjType As AcObjectType) As Boolean

    Dim col As AllObjects
    Dim obj As AccessObject

    ' get the appropriate collection for the object type
    Select Case lngObjType
        Case acTable: Set col = CurrentData.AllTables
        Case acQuery: Set col = CurrentData.AllQueries
        Case acForm: Set col = CurrentProject.AllForms
        Case acReport: Set col = CurrentProject.AllReports
        Case acMacro: Set col = CurrentProject.AllMacros
        Case acModule: Set col = CurrentProject.AllModules
    End Select

    ' look for the matching object
    On Error Resume Next
    Set obj = col(stObjName)
    ObjectExists = (Err = 0)
    On Error GoTo 0
End Function
```

The routine that creates the table is defined as follows. This routine accepts an `HTMLTable` object as an argument. We're using late-binding to this object.

```
Private Sub CreateTableFromHTMLTable(objTable As Object)
    ' DAO objects
    Dim db As DAO.Database
    Dim td As DAO.TableDef
    Dim fld As DAO.Field2

    ' HTML objects
    Dim header As Object      ' HTMLTableRow
    Dim cell As Object         ' HTMLTableCell
    ' other locals
    Dim i As Integer
```

Begin by creating the table with the same name as the `id` of the `HTMLTable` object:

```
' create the table using DAO
Set db = CurrentDb()
Set td = db.CreateTableDef(objTable.id)
```

Get the first row of the table to define the fields in the table:

```
' get the first row
Set header = objTable.rows(0)
```

Chapter 10: Using Automation to Add Functionality

To create the fields in the table, walk the cells collection of the row. We'll use the `innerText` property of the cell to determine the field name. The cell in the row could be either a TH or TD tag in HTML.

```
' create fields
For Each cell In header.cells
    Set fld = td.CreateField(cell.innerText, dbText)
    td.Fields.Append fld
Next

' append the table
db.TableDefs.Append td

' cleanup
db.Close
Set fld = Nothing
Set td = Nothing
Set db = Nothing
End Sub
```

The routine that imports the data is defined as follows. Again, we're passing the routine an `HTMLTable` object. This routine is very similar to the previous routine with the exception that we're skipping the first row of the table. This is because we used it to define the fields in the previous routine.

```
Private Sub ImportHTMLTable(objTable As Object)
    ' parse the HTMLTable object into a table
    ' DAO objects
    Dim db As DAO.Database
    Dim rs As DAO.Recordset2

    ' HTML objects
    Dim row As Object ' HTMLTableRow
    Dim cell As Object ' HTMLTableCell

    ' other data
    Dim lngCounter As Long
    Dim lngCellCounter As Long

    ' if the table does not exist, create it
    If (Not ObjectExists(objTable.id, acTable)) Then
        CreateTableFromHTMLTable objTable
    End If

    ' get a recordset - assumes the id of the HTML table
    ' is the name of the Access table
    Set db = CurrentDb()
    Set rs = db.OpenRecordset(objTable.id)

    ' walk the HTML table
    For lngCounter = 1 To objTable.rows.length - 1
        Set row = objTable.rows(lngCounter)

        ' add a row
        rs.AddNew
```

Part III: Interacting with the Application

```
' map fields to tables using a counter
For Each cell In row.cells
    rs(lngCellCounter) = cell.innerText
    lngCellCounter = lngCellCounter + 1
Next

' commit
rs.Update

' reset the cell counter
lngCellCounter = 0
Next

' cleanup
rs.Close
db.Close

Set rs = Nothing
Set db = Nothing
End Sub
```

To test this code, call the `ImportHTMLTables` routine in the Immediate window using the HTML file created earlier.

Summary

Automation can be your gateway to exciting and highly functional applications that tie together experiences for the user.

In this chapter, you saw:

- ❑ How automation works in terms of the `CreateObject` and `GetObject` functions
- ❑ That Access applications are not necessarily limited to the objects that Access provides
- ❑ How to automate Windows and Internet Explorer to provide rich integration with the operating system and browser
- ❑ How to automate Office applications such as Word, Excel, and Outlook to extend your applications and turn them into solutions

The next chapter will show you how to create customizations for the new Ribbon user interface in Access 2007 that are exciting and intuitive for your users.

11

Creating Dynamic Ribbon Customizations

The Ribbon, first introduced in Office 2007, provides many new and interesting opportunities for user-interface development in Access applications. Unlike menus, the Ribbon gives you a chance to expose functionality in an application that might otherwise be overlooked. For new users, the Ribbon is designed to reduce the barrier to entry, making it easier to find the item the user is looking for.

In this chapter, you learn how to:

- Create ribbon customizations for use in Access applications
- Program the ribbon to provide dynamic user experiences
- Use images effectively in ribbon customizations to provide experiences that are fun and easy-to-use
- Disable or repurpose built-in controls to provide your own functionality

Overview of Ribbon Customizations in Access

Ribbon customizations that you write for Access applications are generally designed to replace the menu bars and toolbars of applications created in previous versions. Although this may be how you typically create ribbon customizations, it is not the only way that you can use them.

You can create two types of customizations for the Ribbon. You can use the first type of customization with a particular database, which appears when a given database is open in Access. The second type of ribbon customization you can create is in a COM add-in. Because COM add-ins are available for any database open in Access, it stands to reason that ribbon customizations created as part of a

Part III: Interacting with the Application

COM add-in are also available for any database. In this chapter, we focus on the first type of customization. Refer to the MSDN Web site for more information about creating ribbon customizations as part of COM add-ins.

Ribbon customizations are written in XML that conforms to a defined XML schema that is available for download. The XML schema for developing ribbon customizations includes controls that are defined as elements in XML and attributes of those controls that define their behaviors and appearance. We take a look at some of the control elements and common attributes used on all types of controls later in this chapter.

Development Tips

Before we get started with developing ribbon customizations, let's go through some tips that will help you during development.

Discovering Errors

By default, you won't see any errors if there are problems with the XML that you've defined for a ribbon customization. To display errors during development, be sure to set the Show add-in user interface errors option in the Advanced page of the Access Options dialog box. Without this option set, ribbon customizations may not load and it may not necessarily be clear why.

When this error is set, Access displays any errors caused by a ribbon customization, as shown in Figure 11-1.

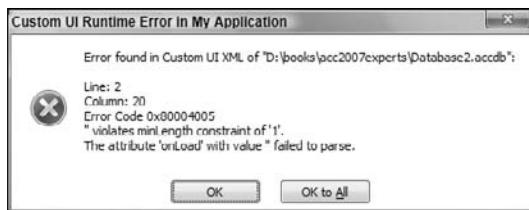


Figure 11-1

Using IntelliSense

As VBA developers, we are very glad to have IntelliSense. We see it as a great time saver during development because we can use it to help complete text while writing code. IntelliSense is also available when developing the XML for a ribbon customization using Visual Studio 2005.

In order to use IntelliSense in Visual Studio 2005, you need to download the XML schema for ribbon development. The schema is included as a part of the 2007 Office System XML Schema reference, which is available for download from the Microsoft Web site.

Once you have the schema, select the `customUI.xsd` schema for a given XML document in Visual Studio 2005 as follows:

1. Launch Visual Studio 2005. You can use any version of Visual Studio, including the Express Editions, which are freely available for download from the Microsoft Web site.

2. In Visual Studio 2005, click File, select New, and then click File.
3. Select XML File and click Open.
4. Click the builder button in the Schemas property for the file. This property is available in the Properties window for the file.
5. Click the Add button and browse to the customUI.xsd file that was installed as a part of the XML Schema reference.
6. Make sure the customUI.xsd schema is checked and click OK in the XSD Schemas dialog box in Visual Studio.

When the schema has been added to the document, you should receive IntelliSense for the `customUI` node, as shown in Figure 11-2.

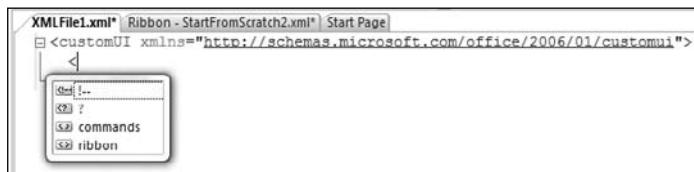


Figure 11-2

To make this schema easily available in Visual Studio, copy the customUI.xsd file to: C:\Program Files\Microsoft Visual Studio 8\xml\Schemas. The customUI.xsd file will appear in the list of schemas for XML documents in Visual Studio after copying to this location. Replace the path to Visual Studio 2005 as needed.

Prevent Loading at Startup

If you are developing a ribbon customization to replace the built-in Access Ribbon, you may want to prevent your customization from loading in order to use the development tools and ribbons included with Access. To prevent your ribbon customization from loading during development, hold down the shift key.

Finding Controls and Images

Office 2007 includes a great number of built-in controls and images that you can use in your applications. As you see later in this chapter, you are not limited to using images from Access. You can use images from other Office applications such as Word, Excel, and Outlook as well! With all of these options available, finding controls and images can be a bit daunting. Luckily, Microsoft has provided some resources to help you find controls and images for use in your applications.

Each application that supports the Ribbon in Office 2007 includes a built-in mechanism for finding controls in the Customize page of the Options dialog box for the application. For example, let's say that you wanted to find the built-in control ID for the Access Close Database button in the Office menu. Here's an easy way to find the control:

1. Open the Access Options dialog box and select the Customize page. The Access Options dialog box is available under the Office menu in Office 2007.

Part III: Interacting with the Application

2. Select Office Menu from the Choose commands from drop-down.
3. Hover the mouse over Close Database. You should see a tooltip that provides the name of the button, FileCloseDatabase, as shown in Figure 11-3.

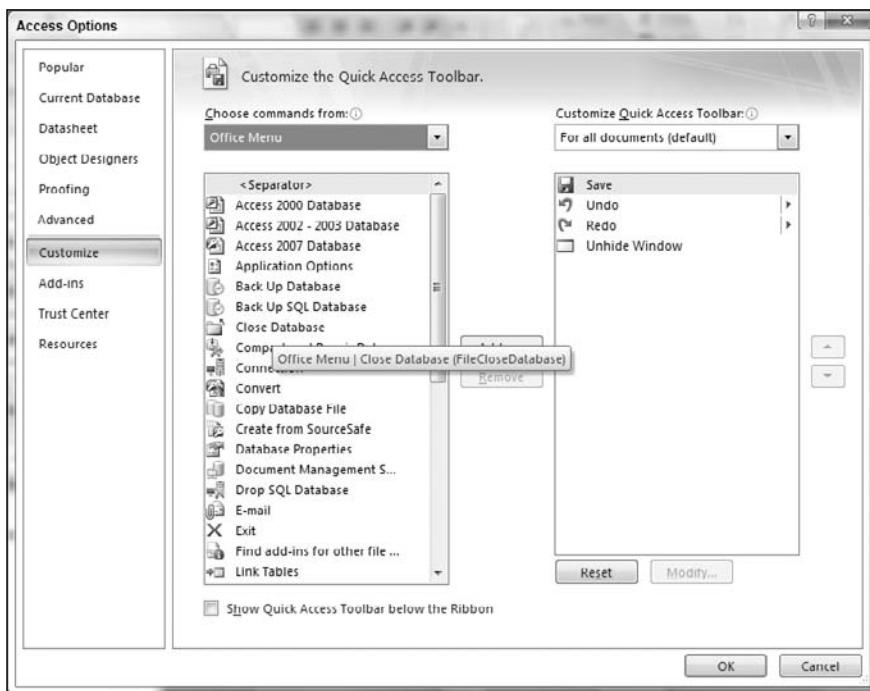


Figure 11-3

This is most helpful when you know exactly what you are looking for. To prevent wading through dialog boxes looking for a given control, Microsoft provides a download called “List of Control IDs” that contains lists of all control ID values in Office 2007. This is also available for download on the Microsoft Web site.

More Resources

Microsoft began to release documentation about the Ribbon for developers prior to the release of Office 2007. With the vast amount of changes made to the user interface, this was intended to get the word out early so that developers were prepared. One resource that has been very useful for us is the Office Ribbon Developer Center on MSDN.

How to Write Ribbon Customizations

As mentioned earlier, using IntelliSense to write the XML for a ribbon customization makes authoring the ribbon much easier. This gives you more time to focus on how you’d like the ribbon to look and function. With this in mind, we’ll use Visual Studio to write ribbon customizations. Later in this chapter, we show you where to save the customization for use in your application.

Getting Started

The root node of a ribbon customization is the `customUI` node, which defines the XML namespace that is tied to the schema for ribbon customizations in Office 2007. Let's use this node to start writing our first ribbon customization. This first customization is pretty straightforward, but it will introduce you to the process.

Start by creating a new XML file in Visual Studio and add the `customUI.xsd` schema to the document, as described in the section, "Using IntelliSense."

Next, add the XML for the `customUI` node, as shown in the code that follows. IntelliSense makes adding this node easier, complete with the namespace.

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">  
    </customUI>
```

The first thing you do is tell the Ribbon that we are adding to the built-in Access Ribbon. To do this, use the `ribbon` node as shown in the following XML:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">  
    <ribbon>  
        </ribbon>  
    </customUI>
```

Next, add a tab with one group to the built-in Access ribbon. To do this, add the XML as shown in the following. We discuss tabs and groups in more detail in the section "Organizing Ribbon Items."

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">  
    <ribbon>  
        <tabs>  
            <tab id="tab1" label="My Tab">  
                <group id="grp1" label="My Group">  
                    </group>  
                </tab>  
            </tabs>  
        </ribbon>  
    </customUI>
```

Now that you have a group, you can start to add controls. Many people start programming in Access and VBA with command buttons, so you'll do the same here. Add a button to the group using the `button` node as shown in the following:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">  
    <ribbon>  
        <tabs>  
            <tab id="tab1" label="My Tab">  
                <group id="grp1" label="My Group">  
                    <button id="btn1" label="Hello World" onAction="=MsgBox('Hello World')"/>  
                </group>  
            </tab>  
        </tabs>  
    </ribbon>  
</customUI>
```

Part III: Interacting with the Application

Note the use of the `onAction` attribute. This attribute is defined for several nodes in the Ribbon schema and is called when the user invokes some functionality on a control. Attributes on controls that call into custom code are called *callbacks*. In this case, we're using the `MsgBox` function in an expression to display the obligatory Hello World message. Using expressions for ribbon customizations in Access is convenient, but this is a programming book. We're here to write code. The remaining callbacks in this chapter use VBA in a database to provide more functionality.

Congratulations! You've written a ribbon customization! You'll see how to add this to a database in a few moments.

Common Attributes

The `onAction` attribute is one you are likely to see frequently. Other attributes that you'll commonly use are listed in the following table.

Attribute	Applies To	Description
<code>enabled</code>	Most controls	Specifies whether a control is enabled.
<code>id</code>	All controls	Defines a control ID. Must be unique in the ribbon customization.
<code>image</code>	Most controls	Specifies the custom image for a control. Typically used with the <code>getImage</code> or <code>loadImage</code> callbacks.
<code>imageMso</code>	Most controls	Specifies an image for a control that is defined by Office 2007.
<code>insertAfterMso</code>	All controls	Specifies the name of a built-in control that you want your control to appear after.
<code>insertBeforeMso</code>	All controls	Specifies the name of a built-in control that you want your control to appear before.
<code>label</code>	Most controls	Defines the text displayed for a control. This is similar to the <code>Caption</code> property for a label in Access.
<code>size</code>	button, gallery, menu, splitButton, toggleButton	Sets the size of a control. Possible values are <code>normal</code> or <code>large</code> .
<code>visible</code>	Most controls	Specifies whether a control is visible.

Loading Ribbons

Once you've written a ribbon customization, you need a way to load it into Access. There are two ways to do this. The easiest way is to use a user-defined system table called `USysRibbons`.

Using the USysRibbons Table

The easiest way to load a custom ribbon is to use a special table called `USysRibbons`. This table has two fields, as shown in the table that follows.

Field Name	Data Type (Size)
ID	AutoNumber
RibbonName	Text (255)
RibbonXml	Memo

Create this table now to add the ribbon customization that you created earlier. Add a record to the table and set the `RibbonName` field to `HelloWorld`, and copy and paste the XML created earlier in the `RibbonXml` field.

When Access opens a database, it looks for a `USysRibbons` table and loads any ribbon customizations that are defined by the records in the table. Ribbon customizations must be named uniquely throughout the application so you cannot duplicate a `RibbonName`. Make this field a primary key if you define a lot of ribbons in an application to prevent yourself from creating duplicate names.

Once you've added the `HelloWorld` ribbon in the table, let's tell Access to load this ribbon when the database is opened. To do this, Access includes a new property in the Current Database page of the Access Options dialog box called Ribbon Name. Set this property to the name of a ribbon customization to load, in our case, `HelloWorld`.

The drop-down list for this control is populated after closing and re-opening a database.

After setting the property, Access tells you that you need to close and re-open the database for the property to take effect. When you re-open the database, you should have a new tab on the Access Ribbon called My Tab, as shown in Figure 11-4.



Figure 11-4

The Ribbon Name property is stored in DAO as a database property on the `DAO.Database` object called `CustomRibbonId`. Use the following code to retrieve the property:

```
MsgBox CurrentDb().Properties("CustomRibbonId")
```

Part III: Interacting with the Application

Using the LoadCustomUI Method

Obviously any data you store in the database adds to its size. If you are using several ribbons in an application, you might want to store ribbons external to your database. Access 2007 includes a new method on the Application object called LoadCustomUI that allows you to do this.

This method only loads ribbon customizations into Access. It does not update the current ribbon. As a result, you need to tell Access to display a ribbon either by:

- ❑ Setting the Ribbon Name property for the database, or
- ❑ Setting the Ribbon Name property for a form or report

We also tend to call this method in a conditional statement as shown in the following example:

```
Function LoadRibbonFromCommandLine() As Long
    Dim stXmlPath As String
    Dim stXmlData As String

    ' get the path to the ribbon and make sure it exists
    stXmlPath = CurrentProject.Path & "\LoadCustomUITest.xml"
    Debug.Assert (Len(Dir(stXmlPath)) > 0)

    ' load the ribbon from disk
    Open stXmlPath For Input Access Read As #1
    stXmlData = Input(LOF(1), 1)

    ' remove the byte order mark
    stXmlData = Mid(stXmlData, InStr(stXmlData, "<customUI"))
    Close #1

    ' Check the command line. If you pass "DEBUG" to the /cmd switch,
    ' load the ribbon with startFromScratch="false"
    If (Command$() = "DEBUG") Then
        stXmlData = Replace(stXmlData, _
                            "startFromScratch=""true""", _
                            "startFromScratch=""false""")
    End If

    ' load the ribbon
    LoadCustomUI "rnbMain", stXmlData
End Function
```

The ribbon customization being loaded from disk defines the startFromScratch attribute as true. In this function, we are checking the command line using the VBA Command function. If we pass in the string DEBUG, then we change the XML after it has been loaded to set it to false. This is a simple indicator that we are working in a debug version of the database and we want to see the Access Ribbon to do development work. You could use a similar technique to load a different ribbon customization altogether either for yourself or for users in a particular security group.

To get the ball rolling for the application in this example, we have preset the `RibbonName` property to `rbnMain` and call the `LoadRibbonFromCommandLine` function from an autoexec macro using the `RunCode` macro action.

Programming the Ribbon

To write ribbon customizations that are flexible and dynamic, you need to write some code. If you were familiar with programming `CommandBar` objects in previous Office versions, you'll find that things have changed quite a bit.

The biggest difference is that there isn't a direct mechanism for setting properties on the Ribbon. In other words, with a `CommandBarControl` object, you could set its caption by using something like the following:

```
Dim objControl As Office.CommandBarControl  
Set objControl = CommandBars("Menu Bar").Controls(0)  
objControl.Caption = "Test Caption"
```

You cannot change properties of controls in the Ribbon in this manner. Instead, you must use *callback* routines that are named in attributes in the ribbon customization XML. A callback is a procedure that the Ribbon calls when it needs to check the state of an object, such as whether a control is enabled or visible; or when it needs data, such as in a combo box control; or when the user has taken some action, such as clicking a button.

Without an object model that includes properties, such as the `CommandBarControl` object, most values for controls in a customization are set using the arguments that are defined in the callback code. We take a closer look at this model throughout this chapter.

Ribbon Objects

The callback routines defined by the Ribbon typically include an instance of an `IRibbonControl` object that defines the control whose callback is being fired. To use this object, set a reference to the Office 12.0 Object Library. Once you've set this reference, you can write callback routines. For example, the signature for the `onAction` callback for a button is defined as follows:

```
Public Sub OnAction(ctl As IRibbonControl)  
End Sub
```

The routine can be named any valid procedure name in VBA. In the XML that defines the callback, use the name of the routine as shown:

```
<button id="btn1" label="button" onAction="OnAction"/>
```

Using Callback Routines

The Ribbon defines several callback attributes that are used for a number of scenarios as mentioned earlier. Callback routines are similar to events. When the Ribbon determines that it needs some information

Part III: Interacting with the Application

or when something occurs, it notifies you using callback routines. The good news is that you are given the control for each callback, so you can reuse the code for a callback routine for multiple controls.

For example, if you use buttons in your customizations, you're likely to use the `onAction` attribute. Let's say that you use several buttons to open forms in your application. Rather than writing a callback for each form, you might store the name of the form in the `tag` attribute of a control. As with the `Tag` property of Access controls, the `tag` attribute of Ribbon controls lets you store extra data with a control. This means you can write one callback routine to handle opening forms such as the following:

```
<button id="btnMoreOptions" onAction="OnOpenForm" tag="frmOptions"
       label="More Options"/>
<button id="btnHelp"          onAction="OnOpenForm" tag="frmHelp"
       label="Help"/>
<button id="btnHome"         onAction="OnOpenForm" tag="frmHome"
       label="Home"/>
```

Then the code for the callback is straightforward, as follows:

```
Public Sub OnOpenForm(Control as IRibbonControl)
    DoCmd.OpenForm Control.Tag
End Sub
```

Common Callback Routines

Several controls define callbacks that are common to many controls. The following table lists callbacks that are common across controls that you may use frequently. We use some of these callbacks later in this chapter in specific scenarios.

Callback	Applies To	Description
getEnabled	Most controls	Callback attribute that is called when the Ribbon is about to determine whether to enable a control. Enables you to dynamically set the <code>enabled</code> attribute for a control.
getLabel	Most controls	Callback attribute that is called when the Ribbon is about to display the text for a control. Enables you to dynamically set the <code>label</code> attribute for a control.
getVisible	Most controls	Callback attribute that is called when the Ribbon is about to determine whether to display a control. Enables you to dynamically set the <code>visible</code> attribute for a control.
onAction	button, checkBox, dropDown, gallery, toggleButton	Callback attribute that is called when the user takes action on a control.

Similar Properties, Methods, or Events in Access

The following table lists several of the callbacks for specific controls along with the equivalent event for the corresponding control in Access.

Control	Callback	Description	Similar Access Event, Method, or Property
button	onAction	Fires when a button is clicked	Click event
checkBox	getPressed	Sets the pressed state for a check box	Value property
	onAction	Fires when the check box is clicked	Click event or
AfterUpdate event			
comboBox	onChange	Fires when an item is selected in a combo box	AfterUpdate event
	getItemCount	Determines how many items will appear in the combo box	AddItem method
	getItemLabel	Sets the label for items in a combo box	AddItem method
	getText	Sets the initial text in a combo box	Value property
dropDown	onAction	Fires when an item is selected in a drop-down	AfterUpdate
	getItemCount	Determines how many items will appear in a drop-down	AddItem method
	getItemLabel	Sets the label for items in a drop-down	AddItem method
	getSelectedIndex	Returns the index of the selected item	ListIndex property
editBox	getText	Sets the initial text of an edit box	Value property
	onChange	Fires when the text has changed in an edit box	AfterUpdate event

Continued on next page

Part III: Interacting with the Application

Control	Callback	Description	Similar Access Event, Method, or Property
toggleButton	getPressed	Sets the pressed state of a toggle button	Value property
	onAction	Fires when a toggle button is clicked	Click event or

AfterUpdate event

For a complete list of callbacks with the expected signatures for the routines refer to MSDN.

Refreshing Controls

You may want to refresh controls in your customizations from time to time. For instance, let's say that you were writing an application for a doctor's office and wanted to use a drop-down control to allow users to set their status to either available or away. In the away state, you might want to lock controls in the Ribbon to prevent users from accessing items while the person was away. In order to do this, you need to handle the `onLoad` callback in the Ribbon.

The `onLoad` callback is defined on the `customUI` node as follows:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
onLoad="OnRibbonLoad">
```

The signature for this callback in code receives an `IRibbonUI` object, which defines two methods: `Invalidate` and `InvalidateControl`. Cache a copy of this object in a global variable to call these methods. The callback is called when the customization is loaded and only then. You cannot call it again. Because globals are reset when an unhandled error occurs, test the object for `Nothing` before calling the method. The following code shows you how to cache the object:

```
Public gobjRibbon As IRibbonUI
Public Sub OnRibbonLoad(objRibbon As IRibbonUI)
    Set gobjRibbon = objRibbon
End Sub
```

Then, to refresh, or invalidate all controls in the Ribbon, call:

```
If (Not gobjRibbon Is Nothing) Then
    gobjRibbon.Invalidate
End If
```

To refresh an individual control, call:

```
If (Not gobjRibbon Is Nothing) Then
    gobjRibbon.InvalidateControl "ControlName"
End If
```

Organizing Ribbon Items

The top level of the command bar structure in previous versions of Office and Access was the menu. Menus contained controls or other menus that sometimes resulted in deeply nested hierarchies that made items difficult to find. One goal of the Ribbon is to expose commonly used commands so that they are visible to users. Let's look at how you can organize controls in a ribbon customization.

Tabs

The top-level means of organization for items in a ribbon customization is a tab. Tabs can contain group nodes. To define a custom tab, use the tab node in the customization, such as the following:

```
<tabs>
  <tab id="tabMyTab" label="My Tab">
    </tab>
</tabs>
```

The tab node is contained within the tabs node unless you are using contextual tabs as described in the next section.

Contextual Tabs

Tabs that appear for a particular view of an object are called *contextual tabs*. Access provides several built-in contextual tabs. For example, there are contextual tabs in design view of Forms, Reports, Macros, and Queries, and when a table is open in datasheet view. You'll also find contextual tabs in other applications in Office. For example, Word provides contextual tabs when designing tables.

To create your own contextual tabs in a customization, use the contextualTabs element in XML. This node must contain another element named tabSet. In the tabSet element, specify an idMso attribute of TabSetFormReportExtensibility. This tells Access to provide contextual tabs for a form or report. Follow these steps to create contextual tabs:

1. Create a new XML file and add the root node and the ribbon node, as follows:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon>
```

2. Add the contextualTabs and tabSet nodes, as follows:

```
  <contextualTabs>
    <tabSet idMso="TabSetFormReportExtensibility">
```

3. Add the ribbon customization as normal. In this example, we create contextual tabs that contain two tabs and two groups:

```
    <tab id="tab1" label="Tab 1">
      <group id="t1grp1" label="Tab 1 - Group 1">
        </group>
      <group id="t1grp2" label="Tab 1 - Group 2">
        </group>
    </tab>
```

Part III: Interacting with the Application

```
<tab id="tab2" label="Tab 2">
    <group id="t2grp1" label="Tab 2 - Group 1">
    </group>
    <group id="t2grp2" label="Tab 2 - Group 2">
    </group>
</tab>
```

4. Close the nodes that were created above, as follows:

```
</tabSet>
</contextualTabs>
</ribbon>
</customUI>
```

5. Add this customization to a USysRibbons table and set the RibbonName field to ctabReport1.
6. Create a new report named Report1.
7. Open the report in design view and set the Ribbon Name property of the report to ctabReport1. This list is refreshed when you close and re-open the database. Set the Caption property of the report to My Report: Caption.
8. Close and re-open the database and open Report1 in Report View. You should see the two tabs that you defined in the customization, as shown in Figure 11-5.

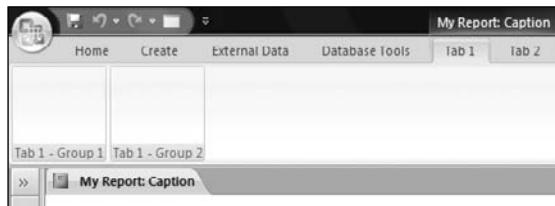


Figure 11-5

Notice that the caption of the report was used in the title bar over the two tabs that were defined. If the Caption property is empty, the name of the object is used in this title.

Contextual tabs do not appear in design view or layout view of an object.

Groups

We have already worked with groups in our customizations. The group node can contain other control types.

Ribbon Controls

The Ribbon provides many different types of controls that you can add to a customization. We've already looked at some of the different types of controls, but let's take a closer look.

The code for the samples that follow is contained in the sample file TestControls.accdb, which is available for download on the Web site for this book.

Buttons

Buttons are created using the `button` node and are likely to be the control you use the most when designing a ribbon customization. Buttons can be created in two different sizes — large or normal — as indicated by the `size` attribute, as shown in the following:

```
<button id="btnMyButton" label="My Button" size="large"/>
```

Figure 11-6 shows five buttons, two of which are large and three that are normal size.

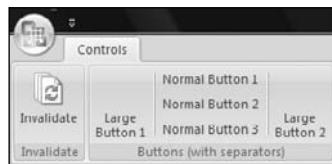


Figure 11-6

Toggle Buttons

As with toggle button controls in Access forms, toggle buttons in a ribbon customization enable you to reflect a true/false state. For example, imagine that you are creating a context-sensitive help system in your application that displays help information in a form. Using a toggle button, you can show or hide the help form.

Toggle buttons are created using the `toggleButton` node, as shown in the following, and can also appear in normal or large size:

```
<toggleButton id="tglHelpForm" size="large" imageMso="Help"  
label="Help Form" onAction="OnPressedAction"/>
```

The `onAction` callback for the toggle button is used to hide or show a form called `frmHelp`, as follows:

```
Public Sub OnPressedAction(ctl As IRibbonControl, Pressed)  
    If (ctl.ID = "tglHelpForm") Then  
        If (CurrentProject.AllForms("frmHelp").IsLoaded) Then  
            ' close the form  
            DoCmd.Close acForm, "frmHelp"  
        Else  
            ' show the form  
            DoCmd.OpenForm "frmHelp"  
        End If
    End If
```

Part III: Interacting with the Application

```
' refresh the control
If (Not gobjRibbon Is Nothing) Then
    gobjRibbon.InvalidateControl "tglHelpForm"
End If
End If
End Sub
```

This customization creates a toggle button, as shown in Figure 11-7.

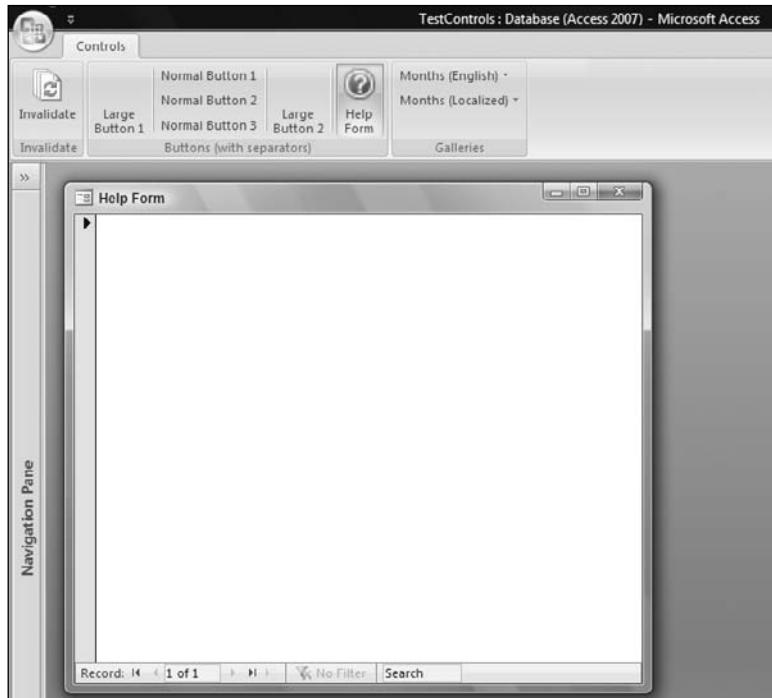


Figure 11-7

When you click the toggle button, the form is shown as depicted in Figure 11-7. When you click the toggle button again, the form should be closed.

Check Boxes

Check boxes have similar behavior to toggle buttons in that they can show a true/false state. They are created using the `checkbox` node, as follows:

```
<checkbox id="chk2" label="Checkbox 2" />
```

Figure 11-8 shows three check boxes, one of which is disabled using the `enabled` attribute with a value of `false`.

Chapter 11: Creating Dynamic Ribbon Customizations

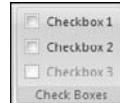


Figure 11-8

Check boxes can also appear in menus but they appear differently as shown in the previous example. The following XML creates two menus that contain check boxes:

```
<menu id="mnuCheck1" label="Normal checkboxes">
    <checkBox id="mnuchk1" label="Menu checkbox 1"/>
    <checkBox id="mnuchk2" label="Menu checkbox 2"/>
</menu>
<menu id="mnuCheck2" label="Large checkboxes" itemSize="large">
    <checkBox id="mnuchk3" label="Menu checkbox 1"/>
    <checkBox id="mnuchk4" label="Menu checkbox 2"/>
</menu>
```

When you check a normal check box in a menu, you should see something that resembles the check box in Figure 11-9.

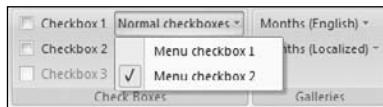


Figure 11-9

We'll talk about menus a little later, but you'll also notice that the menu node contains an attribute called `itemSize`. When you set this attribute to large, the items inside of a menu appear larger. When you select a large check box in a menu you should see something that resembles Figure 11-10.

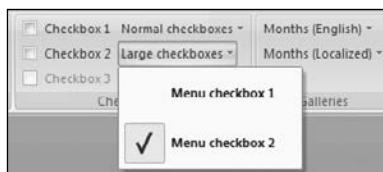


Figure 11-10

Combo Boxes and Drop-Downs

As with combo boxes in Access, combo boxes can be used to provide the user with a list of options to choose from. Combo boxes are created using the `comboBox` node and contain `item` nodes. Using `item` nodes in a combo box creates a static list of items as shown in the following XML and in Figure 11-11:

```
<comboBox id="cboStatic" label="Static combo box">
    <item id="cboItem1" label="Item 1"/>
    <item id="cboItem2" label="Item 2"/>
```

Part III: Interacting with the Application

```
<item id="cboItem3" label="Item 3"/>  
</comboBox>
```

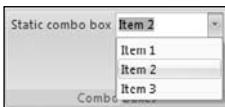


Figure 11-11

To create a combo box with a dynamic list of items, you need to use callbacks. Creating a dynamic combo box is described in the section “The NotInList Event — Ribbon Style.”

A drop-down control is very similar to a combo box, but the user cannot type text in it. Drop-down controls are created using the dropDown node. These controls are very useful for giving the user a list of items that cannot change, such as a status. The following XML defines a drop-down control with two items. (When you click the drop-down, you should see the items listed, as in Figure 11-12.)

```
<dropDown id="ddStatic" label="Static dropdown">  
    <item id="ddItem1" label="Item 1" imageMso="HappyFace" />  
    <item id="ddItem2" label="Item 2" imageMso="Info" />  
</dropDown>
```



Figure 11-12

You'll notice that these controls can also contain images. We'll go into more detail about images later in this chapter.

Labels and Edit Boxes

The Ribbon also defines labels and edit boxes that are similar to labels and text boxes in Access. Label controls can also be disabled, but we tend to leave them enabled to show information. Label controls are useful for displaying status information about data in the application. You might also use them to show the current date and time.

Edit boxes are useful for letting the user enter any information. Later on we'll take a look at using a label control and edit box for form navigation.

Menus

Menus are still available in the Ribbon. Their use, however, is limited to cases when there are multiple command choices that are grouped together, crowding the Ribbon. Menus take on the new appearance of the Ribbon, but still let you construct hierarchies as with previous versions of Office. Let's take a look at the different features of menus. In the following sections, we look at two buttons as they appear in menus that have been decorated with the specified features.

itemSize Attribute

As we mentioned earlier, menus contain an attribute called `itemSize` that lets you control the size of items under the menu. This attribute can be set to either `normal` or `large`. Figure 11-13 shows the two buttons when the `itemSize` is set to `normal`.



Figure 11-13

When the `itemSize` attribute is set to `large`, you should see something similar to Figure 11-14. We've also set the `size` attribute of the menu to `large` so that it fills all three rows of the Ribbon.



Figure 11-14

description Attribute

The `description` attribute is available for controls when you set the `itemSize` attribute of the menu to `large`. This attribute is used to provide more text inside the menu for a given control. The XML that defines this attribute looks like this:

```
<menu id="mnuDescriptions" label="Menu: Descriptions"
      itemSize="large" size="large">
    <button id="mnuBtn1D" label="Button1"
           description="Click here to run something cool"/>
    <button id="mnuBtn2D" label="Button2"
           description="Click here to run something even cooler"/>
</menu>
```

The two buttons that show descriptions are shown in Figure 11-15.



Figure 11-15

Part III: Interacting with the Application

Menu Separators

Menu separators provide a nice separation of controls in a menu and are created using the `menuSeparator` node. This control defines an attribute called `title` that is used to include text in the separator, as shown in the following XML:

```
<menuSeparator id="msCheckboxes" title="Check Boxes" />
```

This creates a menu separator that contains text, as shown in Figure 11-16.

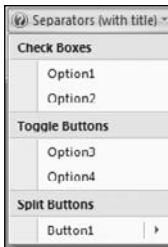


Figure 11-16

The `title` attribute is optional, however, so if you don't define it you get menu separators that look like the flat menu separators from previous versions of Office. These separators appear in Figure 11-17.

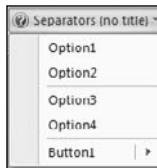


Figure 11-17

New Types of Controls

In addition to standard types of controls, such as buttons, edit boxes, and drop-downs, the Ribbon provides several new types of controls that enable you to create rich experiences for your users.

Dialog Box Launcher

You may have noticed at the bottom of certain groups, there is a very small button in the corner. This button is contained in a control called a dialog box launcher. This control is typically used to launch more options for the group than can be displayed in the Ribbon. For example, let's say that your application provides options to its users. If you have many options, it might not be feasible to put all of them into the Ribbon for fear of cluttering it. Instead, you might only include the most common options in the customization and expose the rest on an Access form. The dialog box launcher lets you create this button

Chapter 11: Creating Dynamic Ribbon Customizations

at the bottom of a group, as shown in the following XML. (This example shows the entire group as an example. Notice that this customization requires the OnOpenForm callback previously defined.)

```
<group id="grpOptions" label="Options">
    <labelControl id="lblOptions" label="Options go here..."/>
    <dialogBoxLauncher>
        <button id="btnMoreOptions" onAction="OnOpenForm" tag="frmOptions"/>
    </dialogBoxLauncher>
</group>
```

Gallery

Use the `gallery` node in the XML for the ribbon customization to create a gallery. Galleries are used to display items that can be arranged in a grid-type layout. Items in the gallery can display images, text, or both images and text. Galleries can also include one or more buttons that appear at the bottom of the gallery.

Let's say that you were building a custom filter for a report and would like to include the months of the year to filter by date. To prevent users from scrolling through a list of months or from selecting from a long list, you may split up the months by quarters, as shown in Figure 11-18.



Figure 11-18

The XML for the gallery, as shown in the following, includes 12 item nodes and a button node that the user can use to change their regional settings. The following layout for the gallery is three columns by four rows:

```
<gallery id="galMonthsEng" label="Months (English)"
    columns="3" rows="4">
    <item id="galMonth1" label="January"/>
    <item id="galMonth2" label="February"/>
    <item id="galMonth3" label="March"/>
    <item id="galMonth4" label="April"/>
    <item id="galMonth5" label="May"/>
    <item id="galMonth6" label="June"/>
    <item id="galMonth7" label="July"/>
    <item id="galMonth8" label="August"/>
    <item id="galMonth9" label="September"/>
    <item id="galMonth10" label="October"/>
    <item id="galMonth11" label="November"/>
    <item id="galMonth12" label="December"/>
    <button id="btnGal1" label="Regional Settings"
        imageMso="ShowTimeZones"
        onAction="ShowRegionalSettings"/>
</gallery>
```

Part III: Interacting with the Application

Split Button

Split buttons are controls that contain a button and a menu and are created using the `splitButton` node. The button inside the split button is displayed in the Ribbon, and as such, is used to set the `label` attribute for a split button. The `splitButton` node itself does *not* define the `label` attribute. As with other buttons, the `button` node inside a split button has an `onAction` attribute that you can handle to receive an event from the Ribbon. The menu items appear as additional choices beneath the button. We take a closer look at split buttons in the section “Creating a Split Button That Sticks.”

Dynamic Menu

As the name suggests, a dynamic menu is a menu that is filled at runtime. Use the `dynamicMenu` node to create a dynamic menu. To fill the content, you must provide a `getContent` callback for the dynamic menu. This callback is required. Dynamic menus are useful for scenarios where users can contribute to the content of the application. We look at an example for using the `dynamicMenu` control in a few moments.

Images

One cool thing about the Ribbon is that it is highly graphical in nature. We think this is cool not just because of the nice graphics, but because the graphics provide some really great opportunities for you and your users. Applications are easier to use because users have graphics as a guide. Naturally, you don’t have to use graphics in your applications, and there are likely to be many applications where they are not appropriate. However, the addition of rich graphics enables new scenarios that may not have been possible in the past.

For the remainder of this chapter, we look at specific scenarios for using ribbon customizations in your applications. The examples for the scenarios are available for download and we’ve included the name of the sample database at the top of the section.

The code for the samples that follow is contained in the sample file `RibbonImages.accdb`, which is available for download on the Web site for this book.

Images Included with Office

As cool as we think graphics are, we have to admit — we’re graphically challenged. We’re far more comfortable writing code than drawing bitmaps or icons. Luckily for us (and for you), there are many images included with Office that can be used in Access applications. Attributes in customizations that end with `Mso` are items that are included with Office. Using the `imageMso` attribute, you can specify the name of an image included with Office. As mentioned earlier in the section “Development Tips,” the easiest ways to find control or image names is to use the Options dialog box for a given application.

You are not limited to using images from Access. In fact, most of our applications that contain ribbon customizations tend to use images from Word or Outlook! The following XML is an example of using images built into Office. (Figure 11-19 shows the result of this customization.)

```
<group id="grpImageMso" label="Examples: imageMso attribute">
  <button id="btnWeather" imageMso="PictureBrightnessGallery"
```

```
label="Weather" size="large"/>/>
<button id="btnNotes" imageMso="ExchangeFolder"
        label="Notes" size="large"/>
<button id="btnSearch" imageMso="ZoomPrintPreviewExcel"
        label="Search" size="large"/>
<button id="btnHelp" imageMso="TentativeAcceptInvitation"
        label="Help" size="large"/>
<separator id="s1"/>
<button id="btnUsers" imageMso="DistributionListSelectMembers"
        label="Manage Users..." size="normal"/>
<button id="btnSysHealth" imageMso="OfficeDiagnostics"
        label="System Health" size="normal"/>
<button id="btnTechSupport" imageMso="TechnicalSupport"
        label="Technical Support" size="normal"/>
</group>
```

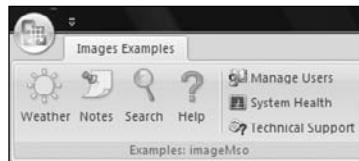


Figure 11-19

Loading Images from External Files

The `imageMso` attribute makes it really easy to get started with images in your applications. But, what if you want to use images that you have on the hard drive or want to display images in the Ribbon as a feature of your application? For example, if you have a product catalog that includes images of the product, wouldn't it be cool to put that into the Ribbon for your users as a selection item? Users would immediately be able to make selections based on a visual. This can go a long way toward making applications easy-to-use.

There are basically two ways to load images dynamically. The easiest way is to use the `loadImage` callback, which is defined in the `customUI` node of a customization. Alternatively, you can handle the `getImage` callback for a given control.

Let's take a look at these two methods.

Creating a Global Image Handler

In the root node of a customization, `customUI`, there is a `callback` attribute defined called `loadImage`. This callback is used in conjunction with the `image` attribute on controls, and is called when the Ribbon asks for an image. Using the `loadImage` callbacks enables you to define one image handler for the application and specify the name of the image in the `image` attribute. Let's take a closer look to see how this works.

Part III: Interacting with the Application

The gallery control mentioned earlier can also be used to display images. This is very useful for creating features, such as:

- Product catalog
- Membership photo gallery
- Options for a screen layout

Galleries can easily be created using the `loadImage` callback. Start with a new customization, as follows:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
    loadImage="OnLoadImage">
    <ribbon startFromScratch="true">
        <tabs>
            <tab id="tab1" label="Images Examples">
                <group id="grpLoadImage" label="Examples: loadImage">
```

Next, add the `gallery` control with images using the `image` attribute. Notice that we've filled the gallery with sample images from Windows Vista. We've also set the height and width of the items in the gallery using the `itemHeight` and `itemWidth` attributes respectively, as follows:

```
<gallery id="galVista" label="Vista Sample Images"
    itemHeight="100" itemWidth="100"
    size="large" imageMso="PictureEffectsShadowGallery">
    <item id="galImg1" image="Autumn Leaves.jpg"/>
    <item id="galImg2" image="Creek.jpg"/>
    <item id="galImg3" image="Desert Landscape.jpg"/>
    <item id="galImg4" image="Dock.jpg"/>
    <item id="galImg5" image="Forest Flowers.jpg"/>
    <item id="galImg6" image="Forest.jpg"/>
    <item id="galImg7" image="Frangipani Flowers.jpg"/>
    <item id="galImg8" image="Garden.jpg"/>
    <item id="galImg9" image="Green Sea Turtle.jpg"/>
    <item id="galImg10" image="Humpback Whale.jpg"/>
    <item id="galImg11" image="Oryx Antelope.jpg"/>
    <item id="galImg12" image="Toco Toucan.jpg"/>
    <item id="galImg13" image="Tree.jpg"/>
    <item id="galImg14" image="Waterfall.jpg"/>
    <item id="galImg15" image="Winter Leaves.jpg"/>
</gallery>
</group>
</tab>
</tabs>
</ribbon>
</customUI>
```

Write the `OnLoadImage` callback as follows. This routine uses the `Environ` function in VBA to help retrieve the path of the Sample Pictures folder on Vista. You may need to change this path, as follows, if you are not using Vista:

```
Sub OnLoadImage(ImageView As String, ByRef Image)
    ' get the image path
```

Chapter 11: Creating Dynamic Ribbon Customizations

```
Dim stPath As String  
  
' this path works on Vista only  
stPath = Environ("PUBLIC") & "\Pictures\Sample Pictures\" & ImageName  
Set Image = LoadPicture(stPath)  
End Sub
```

When you select the gallery, it should be filled with images, as shown in Figure 11-20.



Figure 11-20

getImage and getItemImage Callbacks

Instead of writing an image handler in the `loadImage` callback, many controls also provide a `getImage` callback that can be used to set the image for the control. This callback is useful when information about the image is stored in a table and you want to include more information about the image besides the file name that is stored in the `image` attribute.

Because we are working with a `gallery` control, let's actually set the image for items in the gallery, not the gallery control itself. To do this, we use the `getItemImage` callback. This works very similarly to the `getImage` callback. To use the `getItemImage` callback, we create a table to store information about images. This table maps the ID of the control in the ribbon customization to the path of the image, along with some additional information.

Create a new table with the following fields. This table stores information about pictures, such as the file name, a friendly name of the image, and camera information that we displays in a supertip. Save the table as `tblImages`.

Remember that if you've opened a database with a customization that includes the `startFromScratch` attribute set, you need to hold down the Shift key to get back to the design tools.

Part III: Interacting with the Application

Field Name	Data Type	Properties
ControlID	Text (255)	Primary Key
ImageFileName	Text (255)	
CameraMake	Text (255)	
CameraModel	Text (255)	
EXIFVersion	Text (255)	

This table with some sample data is available for download on the Web site for this book. The images used with the sample are also included.

We use the sample pictures included with Windows Vista for this example and include some arbitrary information for the other fields for testing. Because the data comes from the table, the following customization is pretty easy:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon startFromScratch="true">
    <tabs>
      <tab id="tab1" label="Images Examples">
        <group id="grpGetImage" label="Example: getImage callback">
          <gallery id="galGetImage" label="Vista Images (detailed)">
            size="large" imageMso="Camera"
            itemHeight="100" itemWidth="100"
            getItemCount="OnGetItemCount"
            getItemID="OnGetItemID"
            getItemImage="OnGetItemImage"
            getItemScreentip="OnGetItemScreentip"
            getItemSupertip="OnGetItemSupertip"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

You'll notice that there are a couple of additional callbacks we haven't discussed yet. So, let's discuss those for a moment. Screen tips are large tooltips that can be displayed for controls in a customization. The screentip portion appears in bold and is used to provide context to the tip. The supertip portion of a screentip is not in bold and can display extra information. As you've probably guessed, we're going to use the screentip to display the name of the image and the supertip to display the detailed information about the image. By using these callbacks, we can create data-driven tooltips for our applications.

To fill a gallery dynamically such as this, we first need to tell the Ribbon how many items are in the gallery. For this, we implement the `OnGetItemCount` callback. But, before we can do this, we need a Recordset object because we want to read data from the `tblImages` table. So, define the following variable at the top of a new module:

```
Dim m_rs As DAO.Recordset2
```

Chapter 11: Creating Dynamic Ribbon Customizations

Next, add the `OnGetItemCount` callback. The Ribbon calls this one first, so we open the recordset and return the number of records as the count of items, as follows:

```
Sub OnGetItemCount(ctl As IRibbonControl, ByRef Count)
    ' open the recordset
    Set m_rs = CurrentDb().OpenRecordset("tblImages")

    ' return the count
    Count = m_rs.RecordCount
End Sub
```

Next, add the `OnGetScreentip` callback to return the file name for the image in bold, as follows:

```
Sub OnGetItemScreentip(ctl As IRibbonControl, Index As Integer, ByRef Screentip)
    Screentip = m_rs("ImageFileName")
End Sub
```

Now, add the `OnGetSupertip` callback. This callback provides the detailed information to the tooltip.

```
Sub OnGetItemSupertip(ctl As IRibbonControl, Index As Integer, ByRef Supertip)
    Supertip = "Dimensions: " & m_rs("Dimensions") & vbCrLf
    Supertip = Supertip & "Camera maker: " & m_rs("CameraMaker") & vbCrLf
    Supertip = Supertip & "Camera model: " & m_rs("CameraModel") & vbCrLf
    Supertip = Supertip & "EXIF version: " & m_rs("EXIFVersion") & vbCrLf
End Sub
```

We need to load the actual image. For this, we'll implement the `getitemImage` callback. In this code, we'll create the full path to the image using the file name in the table and a subdirectory of the folder where the database resides called images. Use the `LoadPicture` function in VBA to load the image, as follows:

```
Sub OnGetItemImage(ctl As IRibbonControl, Index As Integer, ByRef Image)
    Dim stPath As String
    stPath = CurrentProject.Path & "\images\" & m_rs("ImageFileName")

    ' load the picture
    Set Image = LoadPicture(stPath)
End Sub
```

Last, we'll set the ID for the item. We aren't doing anything with the ID in this example, but if you were to later handle the `onAction` callback for items in the gallery you might need the ID. The ID is assigned using the `getitemID` callback, as follows:

```
Sub OnGetItemId(ctl As IRibbonControl, Index As Integer, ByRef ID)
    'Debug.Print ctl.ID, Index, "OnGetItemId"
    ID = m_rs("ControlID")
    m_rs.MoveNext

    If (m_rs.EOF) Then
        m_rs.Close
        Set m_rs = Nothing
    End If
End Sub
```

Part III: Interacting with the Application

When you select the gallery and hover over an image, you should see the images with a tooltip, as shown in Figure 11-21.



Figure 11-21

Loading Images from an Attachment Field

Access 2007 includes a new data type called Attachment that allows you to embed files inside a database. This new data type, which is available only with the new .ACCDB file format, also compresses certain file formats such as bitmaps in a database. Bitmap images inside an attachment field can be used as the source for an image in a ribbon customization in Access. The trick to using these fields is to have a form that is bound to the attachment field in the table. Let's see how this works.

Create the Table

Start by creating a table that contains an attachment field. The table should have the following schema. Save the table as `tblAttachments`.

Field Name	Data Type
ID	AutoNumber
Attachments	Attachment

After you've created the table, add three bitmap (.bmp images into the first record of the attachment field). This example only handles cases where the attachments are in the first record. Make a note of the file name for each file.

Create the Form

You need a form to contain the attachments from the table because the images in the customization require the attachment control of a form to render the images. Create a new form using the `tblAttachments` table and save it as `frmAttachments`.

Create the Customization

Now, we need to add the customization. Use the following XML to define the customization. Notice that we've used the `tag` attribute to store the file name of the image as it appeared when you added it to the attachment field:

```
<group id="grpAttachmentImages" label="Access Attachment Field">
    <button id="btnAttachment1" getImage="OnGetAttachmentImage"
            size="large" label="Attachment1" tag="Image1.bmp"/>
    <button id="btnAttachment2" getImage="OnGetAttachmentImage"
            size="large" label="Attachment2" tag="Image2.bmp"/>
    <button id="btnAttachment3" getImage="OnGetAttachmentImage"
            size="large" label="Attachment3" tag="Image3.bmp"/>
</group>
```

Add the Callback

Last, add the `OnGetAttachmentImage` callback for the customization. This callback is used in the `getImage` callback for the button, as follows:

```
Sub OnGetAttachmentImage(ctl As IRibbonControl, ByRef Image)
    ' open the attachment form (hidden)
    If (Not CurrentProject.AllForms("frmAttachments").IsLoaded) Then
        DoCmd.OpenForm "frmAttachments", , , , acHidden
    End If

    ' bind the image which has the image name in the tag attribute
    Set Image = Forms("frmAttachments")!Attachments.PictureDisp(ctl.Tag)

    ' close the form
    DoCmd.Close acForm, "frmAttachments"
End Sub
```

You'll notice that this code opens the `frmAttachments` form in hidden mode and uses the hidden `PictureDisp` property of the attachment control to retrieve the image. The name of the image is specified in the `Tag` property of the `IRibbonControl` object that you defined earlier. The `PictureDisp` property returns an `IPictureDisp` object similarly to the `LoadPicture` function in VBA.

Moving Functionality into the Ribbon

Now that we've gone through the available controls and programming the Ribbon, it's time to put these pieces to use with some more scenarios.

Part III: Interacting with the Application

The code for the samples that follow is contained in the sample file DynamicRibbons.accdb, which is available for download on the Web site for this book.

The NotInList Event — Ribbon Style

Let's start with a pretty common scenario in Access. The NotInList event is used in combo boxes to add values to the underlying row source for the combo box. We can simulate the same effect using a combo box in a ribbon customization with just a few callbacks.

Let's say that we have a list of categories that is used as a lookup field. To manage the lookup field, we add a combo box to the Ribbon. There are two parts to this example. First, we need to be able to dynamically fill the combo box. Second, we need to be able to update it. To do this, we implement the following callbacks:

- Fill — getItemCount, getItemLabel
- Update — onChange

Add the following XML for the customization to the USysRibbons table:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
    onLoad="OnRibbonLoad">
    <ribbon startFromScratch="true">
        <tabs>
            <tab id="tabNotInList" label="NotInList Example">
                <group id="grpNotInList" label="NotInList Example">
                    <comboBox id="cboTest" label="Categories"
                        getItemCount="OnGetItemCount"
                        getItemLabel="OnGetItemLabel"
                        onChange="OnNotInListRibbon"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```

We'll need a recordset to fill the combo box. Add the following declaration to the top of a new module:

```
Private m_rs As DAO.Recordset
```

The first callback that the Ribbon asks for is the getItemCount callback, so let's start there. The getItemCount callback is used to tell the combo box how many items are in the list. To fill the list, we'll use a recordset that points to the Categories table (from previous versions of Northwind). As with some of the other callbacks that we've seen, we return the number of items in an argument that is defined in the callback, as follows:

```
Public Sub OnGetItemCount(ctl As IRibbonControl, ByRef Count)
    ' open the recordset
    Dim stSQL As String
    stSQL = "SELECT * FROM tblCategories ORDER BY CategoryName"
```

Chapter 11: Creating Dynamic Ribbon Customizations

```
Set m_rs = CurrentDb.OpenRecordset(stSQL)
m_rs.MoveLast
m_rs.MoveFirst

' return the count
Count = m_rs.RecordCount
End Sub
```

Next, we need to fill the items in the combo box using the `getItemLabel` callback. Having set the number of items in the combo box, the `getItemLabel` callback is called for each item. As a result, we need to keep track of where we are in the recordset.

```
Public Sub OnGetItemLabel(ctl As IRibbonControl, index As Integer, ByRef Label)
    ' set the label
    Label = m_rs("CategoryName")
    m_rs.MoveNext

    If (m_rs.EOF) Then
        m_rs.Close
        Set m_rs = Nothing
    End If
End Sub
```

The callback starts by returning the label and moving to the next record in the recordset. This advances the cursor in the recordset and provides the next category to the combo box. When we reach end-of-file (EOF) in the recordset, it's safe to do cleanup. That's all that is required to fill a combo box in a ribbon customization using items in a table!

This is a relatively straightforward scenario. There may be times when you need to handle selection of items in the list and want to assign ID values to each item in the list. To dynamically assign IDs, you should handle the `getitemId` callback.

All that's remaining now is the actual `NotInList` implementation. To do this, we'll handle the `onChange` callback for the combo box. This callback is fired when the text inside the combo box is changed. The signature for the callback includes an argument called `Text` that represents the data that was entered in the combo box. This is analogous to the `NewData` argument to the `NotInList` event in Access.

Remember that we closed and destroyed the recordset earlier so we need to re-open it. This time, however, we'll open it with a filter for the item that was entered. If there are no matching records, we add it to the recordset. As you might imagine, we then invalidate the combo box to refresh the items in the list that reflect the new record that was added to the `Categories` table, as follows:

```
Public Sub OnNotInListRibbon(ctl As IRibbonControl, Text As String)
    ' open the recordset
    Dim stSQL As String
    stSQL = "SELECT * FROM tblCategories WHERE CategoryName = '" & Text & "'"

    Set m_rs = CurrentDb().OpenRecordset(stSQL)

    If (m_rs.BOF And m_rs.EOF) Then
        ' add the item
        m_rs.AddNew
```

Part III: Interacting with the Application

```
m_rs("CategoryName") = Text
m_rs.Update

    ' invalidate the combo box to refresh
    gobjRibbon.InvalidateControl "cboTest"
End If

    ' close
m_rs.Close
Set m_rs = Nothing
End Sub
```

Form Navigation

Have you ever written your own navigation form because you wanted a different look than the navigation buttons provided by Access? If so, this next example is for you. Here's how you can move form navigation into the Ribbon.

We've started with a form based on the `Customers` table from Northwind again, but any form should work. You need to add code to the form for some of the requirements, which are listed here:

- Move between first, last, previous, and next records
- Populate a label control with the current position in the form
- Jump to a particular record in the form
- Navigation controls should be kept up to date when you navigate through the form directly without using the controls
- Disable the navigation buttons in the ribbon customization depending on where we are in the form

Let's get started with the XML for the customization. Because we need to refresh a label and buttons, we need to invalidate controls. This means we need to handle the `onLoad` callback, as follows:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
    onLoad="OnRibbonLoad">
```

Next, set up the ribbon, the tab, and the group, as follows:

```
<ribbon startFromScratch="true">
    <tabs>
        <tab id="tabNav" label="Navigation">
            <group id="grpNav" label="Navigation">
```

Add a button to open the sample form. This requires the `OnOpenForm` callback, as defined earlier. As you can see, our sample form is called `frmCustomers`. This is simply a shortcut to help get into the sample.

```
<button id="btnSample" label="Open Sample Form"
    imageMso="AccessFormModalDialog"
    tag="frmCustomers" onAction="OnOpenForm" />
```

Chapter 11: Creating Dynamic Ribbon Customizations

Time to start adding navigation controls. We want a layout that is linear rather than vertical, so we're using a `box` control to define a horizontal layout. Start with the navigation label inside the box. To set the text for the label, we handle the `getLabel` callback, as follows:

```
<box id="boxNavLbl" boxStyle="horizontal">
    <labelControl id="lblNav" getLabel="OnGetLabel"/>
</box>
```

Next, we want to lay out the buttons and edit box in a horizontal layout, so we create another `box` node. The buttons should appear to be grouped together, so we're using a `buttonGroup` control to define a particular appearance.

```
<box id="boxNav" boxStyle="horizontal">
    <buttonGroup id="bg1">
```

Time to add the individual buttons. Notice that we are using built-in images specified with the `imageMso` attribute. Each button calls the same callback named `OnNavigateRecord`. To determine whether a control should be enabled, we handle the `getEnabled` callback, as follows:

```
<button id="btnNavFirst" imageMso="MailMergeGoToFirstRecord"
        onAction="OnNavigateRecord"
        getEnabled="OnGetNavEnabled"/>
<button id="btnNavPrev" imageMso="MailMergeGoToPreviousRecord"
        onAction="OnNavigateRecord"
        getEnabled="OnGetNavEnabled"/>
</buttonGroup>
```

Add the edit box for the navigation. We're handling the `onChange` callback so that the user can enter a number and jump to a particular record. We're handling the `getText` callback to put an empty string into the edit box for invalid data. The `editBox` control also defines an attribute called `sizeString` that is used to define the width of the edit box. The width of the string in this attribute determines the width of the control.

```
<editBox id="txtJump" sizeString="000000"
        onChange="OnChangeRecord"
        getText="OnGetText"/>
```

Finish the customization by adding two more buttons inside a `buttonGroup`, and close the nodes for the ribbon, as follows:

```
<buttonGroup id="bg2">
    <button id="btnNavNext" imageMso="MailMergeGoToNextRecord"
            onAction="OnNavigateRecord"
            getEnabled="OnGetNavEnabled"/>
    <button id="btnNavLast" imageMso="MailMergeGotToLastRecord"
            onAction="OnNavigateRecord"
            getEnabled="OnGetNavEnabled"/>
</buttonGroup>
</box>
</group>
</tab>
</tabs>
</ribbon>
</customUI>
```

Part III: Interacting with the Application

When you put this into a USysRibbons table, the customization should look something like the Ribbon shown in Figure 11-22.

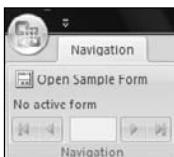


Figure 11-22

To enable the functionality, we need to start writing the callbacks. Create a new module called basFormNavigationCallbacks for this code. Let's start by writing the OnGetText callback, as follows. This is used to simply write an empty string into the edit box for invalid entry cases.

```
Public Sub OnGetText(ctl As IRibbonControl, ByRef Text)
    If (ctl.Id = "txtJump") Then
        Text = ""
    End If
End Sub
```

Next, add the OnGetLabel callback. This routine is called to set the text for the navigation label. The following code sets the text for the label. If there are no forms open, we set the label to "No active form." When there is an open form, we set the label to something such as "Record 1 of 10." If the active form is filtered, we append the string "(Filtered)."

```
Public Sub OnGetLabel(ctl As IRibbonControl, ByRef Label)
    Dim f As Form

    If (ctl.Id = "lblNav") Then
        If (Forms.Count = 0) Then
            Label = "No active form"
        Else
            Set f = Screen.ActiveForm
            Label = "Record " & f.CurrentRecord & " of " &
                f.RecordsetClone.RecordCount

            If (f.FilterOn) Then
                Label = Label & " (Filtered)"
            End If
        End If
    End If
End Sub
```

Next, add the OnNavigateRecord callback, as follows. This routine is called by each of the navigation buttons. To do navigation, we're simply calling DoCmd.GotoRecord and passing the appropriate value based on the button that was clicked.

```
Public Sub OnNavigateRecord(ctl As IRibbonControl)
    Dim lRecord As AcRecord
```

Chapter 11: Creating Dynamic Ribbon Customizations

```
Select Case ctl.Id
    Case "btnNavFirst": lRecord = acFirst
    Case "btnNavPrev": lRecord = acPrevious
    Case "btnNavNext": lRecord = acNext
    Case "btnNavLast": lRecord = acLast
End Select

' do the navigation
DoCmd.GoToRecord , , lRecord

If (Not gobjRibbon Is Nothing) Then
    ' invalidate the nav label
    gobjRibbon.InvalidateControl "lblNav"

    ' invalidate the button to enable/disable
    gobjRibbon.InvalidateControl ctl.Id
End If
End Sub
```

Because we've moved records, we also need to invalidate the navigation label. We do this by calling `InvalidateControl` for `lblNav`.

Next, add the `OnChangeRecord` callback that is fired when the user enters a value in the edit box. To prevent the user from doing something invalid, there are some additional checks in this code. We first ensure that the user entered a number. If they didn't, we alert the user and invalidate the edit box. This, in turn, calls the `OnGetText` callback and writes an empty string.

To move the record, we're manipulating the underlying Recordset for the form. Again, to prevent the user from doing something invalid (much as Access itself does), we've added some checks. If you enter a value that is greater than the number of records or less than zero, we alert the user and invalidate the edit box.

```
Public Sub OnChangeRecord(ctl As IRibbonControl, Text)
    If (Not IsNumeric(Text)) Then
        MsgBox "Please enter a number", vbExclamation, "Cannot Move Record"
        gobjRibbon.InvalidateControl "txtJump"
    Else
        ' move to the specified record
        With Screen.ActiveForm.Recordset
            .MoveFirst
            If (CLng(Text) > .RecordCount Or CLng(Text) < 1) Then
                MsgBox "Cannot move to specified record", vbInformation
                gobjRibbon.InvalidateControl "txtJump"
            Else
                .Move CLng(Text) - 1
            End If
        End With
    End If
End Sub
```

Nice job so far. One callback left — `OnGetNavEnabled`. Remember that we want to disable controls when they are not available. In other words, if you're on the first record, you shouldn't be able to move

Part III: Interacting with the Application

to the first or previous records. So why not disable the controls? The following code sets the enabled attribute, depending on the selected control and the CurrentRecord property of the active form:

```
Public Sub OnGetNavEnabled(ctl As IRibbonControl, ByRef Enabled)
    Dim f As Form

    If (Forms.Count > 0) Then
        Set f = Screen.ActiveForm

        Select Case ctl.Id
            Case "btnNavFirst"
                Enabled = (f.CurrentRecord > 1)
            Case "btnNavPrev"
                Enabled = (f.CurrentRecord > 1)
            Case "btnNavNext"
                Enabled = (f.CurrentRecord < f.Recordset.RecordCount)
            Case "btnNavLast"
                Enabled = (f.CurrentRecord < f.Recordset.RecordCount)
        End Select
    End If
End Sub
```

Great! Okay, hang on a minute — we're not quite done. We need to add some code to our form to keep the controls in sync if you navigate using the form instead of the controls. Keeping the controls in sync will mimic the behavior that Access provides in the navigation buttons for a form. To do this, add the following helper routine to the code behind the form. This simply invalidates the navigation controls.

```
Private Sub InvalidateNavControls()
    With gobjRibbon
        .InvalidateControl "lblNav"
        .InvalidateControl "btnNavFirst"
        .InvalidateControl "btnNavPrev"
        .InvalidateControl "btnNavNext"
        .InvalidateControl "btnNavLast"
    End With
End Sub
```

Last, we need to call this code as you move from record to record, and also when the form is closed. When the form is closed, the label resets to “No active form.”

```
Private Sub Form_Close()
    ' invalidate
    InvalidateNavControls
End Sub
Private Sub Form_Current()
    ' invalidate
    InvalidateNavControls
End Sub
```

To test the navigation, open the sample form and move from record to record. Also, try to use the first and last buttons and jump to a specific record.

Chapter 11: Creating Dynamic Ribbon Customizations

In this sample, we handled the getEnabled callback to disable a control by returning a Boolean value that indicates whether a control should be enabled. The general style guideline from Microsoft is that controls in the Ribbon should be disabled instead of hidden. If you need to hide controls, instead of disabling them, use the getVisible callback.

Managing Filters Using a Dynamic Menu

Let's say that you enable your users to save filters of a particular form so that they can reuse the filter later. You store information about the saved filters in a table. This information includes a friendly name of the filter and a description. Using a dynamic menu is an interesting way to display this information to the user because it permits changes from the user. To be truly dynamic, the Ribbon defines an attribute on the dynamicMenu node called invalidateContentOnDrop. Set this attribute to true to call the getContent callback every time the user drops down the menu.

To create the filter example, start by creating a table to save filters. The table should have the following fields defined. Save the table as `tblSavedFilters` when you're done. Create a form based on this table that the user can use to manage filters later on. Name the form `frmSavedFilters`.

Field Name	Data Type	Properties
FilterName	Text (255)	Primary Key
FilterDescription	Text (255)	
FilterString	Text (255)	

Next, we define the XML for the customization. This sample was created using the Northwind sample database from previous versions of Access but it should work for any form because the filter is being saved as it relates to the form that is currently open.

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
    <ribbon startFromScratch="true">
        <tabs>
            <tab id="tabSavedFilters" label="Saved Filters">
                <group id="grpFilter" label="Filters">
                    <dynamicMenu id="dmnu1" label="Saved Filters"
                        getContent="OnGetContent"
                        invalidateContentOnDrop="true"
                        size="normal" imageMso="Filter"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```

Next, add the `getContent` callback, starting with the declarations:

```
Public Sub OnGetContent(ctl As IRibbonControl, ByRef content)
    Dim stMenu As String           ' XML for the menu being filled
    Dim stID   As String           ' ID for buttons in the menu
```

Part III: Interacting with the Application

```
Dim rs      As DAO.Recordset2    ' recordset containing saved filters
Dim stSQL   As String
```

Open a recordset against the saved filter table. We're sorting the recordset based on the `FilterString` field in descending order so that when the menu is created the `FilterString` field is sorted ascending. As an alternative to opening the recordset sorted in descending order, you could iterate backwards through the recordset.

```
' open the recordset
stSQL = "SELECT * FROM tblSavedFilters ORDER BY FilterString DESC"
Set rs = CurrentDb.OpenRecordset(stSQL)
```

We need to start building the menu. To do this, we define XML that will be used in the customization for the dynamic menu. Start with the menu node. Notice that we need to include the `customUI` namespace definition. We've also added a menu separator for aesthetics.

```
' create the menu node
stMenu = "<menu xmlns='http://schemas.microsoft.com/office/2006/01/customui'
itemSize='large'>"
stMenu = stMenu & "<menuSeparator id='msMyFilters1' title='Filters' />"
```

It's time to fill the menu with the list of saved filters. To do this, we walk through the recordset. We're going to fill the menu with buttons so we've written a helper function called `GetButtonXml` to generate the XML for a button node given some parameters. This function appears in a moment.

One of the arguments to the `GetButtonXml` helper function is called `strAction`. This argument is used as the `onAction` callback for the button in the menu. We're going to create a callback called `DoApplyFilter` that applies the selected filter based on the user selection.

```
' build the XML for the menu
While (Not rs.EOF)
    ' get the ID for the button by replacing any spaces with empty spaces
    stID = Replace(rs("FilterName"), " ", "")
    ' Append the button node
    stMenu = stMenu & GetButtonXml(stID, _
        rs("FilterName"), _
        "DoApplyFilter", _
        rs("FilterDescription"), _
        rs("FilterName"))
    rs.MoveNext
Wend
```

Before we close the menu node, we'd like to add a couple of static buttons to our menu to manage filters. The first is a Save button that has its own callback. The second is a button that opens the `frmSavedFilters` form that you created earlier. Add the buttons as follows:

```
' add buttons to manage filters
stMenu = stMenu & "<menuSeparator id='msMyFilters2' title='Manage Filters' />"
stMenu = stMenu & "<button id='btnSaveFilter' label='Save Filter' " & _
    "onAction='OnSaveFilter' imageMso='FileSave' />"
stMenu = stMenu & "<button id='btnClearFilter' label='Clear Filter' " & _
    "onAction='OnClearFilter' />"
stMenu = stMenu & "<button id='btnFilters' label='Manage...' " & _
```

Chapter 11: Creating Dynamic Ribbon Customizations

```
"tag='frmSavedFilters' " & _  
"onAction='OnOpenForm'/'>"
```

Okay, now we close the menu node and the recordset:

```
' close the menu node  
stMenu = stMenu & "</menu>"  
  
' cleanup  
rs.Close  
Set rs = Nothing
```

And last, of course, we return the content to the customization and exit the routine:

```
' return the XML for the menu  
content = stMenu  
End Sub
```

So far so good. We have some helper functions and additional callbacks to write, so let's add those starting with GetButtonXml:

```
Private Function GetButtonXml(strID As String, _  
    strLabel As String, _  
    strAction As String, _  
    Optional strDescription As String = "", _  
    Optional strTag As String = "")  
    ' builds the XML for a button  
    GetButtonXml = "<button id=''' & strID & ''' & _  
        "         label=''' & strLabel & ''' & _  
        "         onAction=''' & strAction & '''  
    ' add the description attribute  
    If (Len(strDescription) > 0) Then  
        GetButtonXml = GetButtonXml & _  
            "             description=''' & strDescription & '''  
    End If  
  
    ' add the tag attribute  
    If Len(strTag) > 0 Then  
        GetButtonXml = GetButtonXml & "         tag=''' & strTag & '''  
    End If  
  
    ' close the node  
    GetButtonXml = GetButtonXml & "/>"  
End Function
```

As you can see, this function accepts arguments for the `id`, `label`, `onAction`, `description`, and `tag` attributes of a button node in a customization.

Next, add the `OnSaveFilter` callback, as follows. This procedure is called when the user clicks the Save Filter button in the dynamic menu.

```
Public Sub OnSaveFilter(ctl As IRibbonControl)  
    Dim stFilter As String
```

Part III: Interacting with the Application

```
' get the filter for the current filter
On Error GoTo SaveFilterErrors

stFilter = Screen.ActiveForm.Filter

' make sure there is a filter
If (Len(stFilter) = 0) Then
    MsgBox "Filter has not been set for the ActiveForm, cannot save.", _
        vbExclamation, "Cannot Save Filter"
    Exit Sub
End If

' open the form to save the filter
DoCmd.OpenForm "frmSavedFilters", , , , acFormAdd, , stFilter

Exit Sub

SaveFilterErrors:
If (Err = 2475) Then
    MsgBox "There is no open form", vbExclamation
    Exit Sub
Else
    Stop
End If
End Sub
```

For this callback, we're making sure that a form is open and if so, getting its filter using the `Filter` property of the `Form` object. If the `Filter` property is not empty, we open the `frmSavedFilters` form in data entry mode and pass the filter to the form in its `OpenArgs`.

Next, add the `OnClearFilter` callback. This procedure is called when the user clicks the Clear Filter button in the dynamic menu.

```
Public Sub OnClearFilter(ctl As IRibbonControl)
    Screen.ActiveForm.Filter = ""
    Screen.ActiveForm.FilterOn = False
End Sub
```

Next, we write the `OnOpenForm` callback:

```
Public Sub OnOpenForm(ctl As IRibbonControl)
    DoCmd.OpenForm ctl.Tag
End Sub
```

And last, the `DoApplyFilter` callback. When we created the button for the saved filter in the menu, we passed in the name of the filter in the `FilterName` field to the `tag` attribute of the button. This is used to query the `tblSavedFilters` table to ask for the filter string.

```
Public Sub DoApplyFilter(ctl As IRibbonControl)
    Dim stSQL      As String
    Dim rs         As DAO.Recordset2
```

Chapter 11: Creating Dynamic Ribbon Customizations

```
' build the SQL statement
stSQL = "SELECT FilterString FROM tblSavedFilters "
stSQL = stSQL & "WHERE FilterName = '" & ctl.Tag & "'"

' open the recordset
Set rs = CurrentDb.OpenRecordset(stSQL)

' set the filter
Screen.ActiveForm.Filter = rs("FilterString")
Screen.ActiveForm.FilterOn = True

' cleanup
rs.Close
Set rs = Nothing
End Sub
```

We need to add one more piece of code — to pick up the new filter in the frmSavedFilters form. Remember that we pass it a new filter via its `OpenArgs` property. Add the following code to the `Form_Load` event of frmSavedFilters:

```
Private Sub Form_Load()
    ' check openargs for a filter string
    If (Me.DataEntry And Not IsNull(Me.OpenArgs)) Then
        Me.FilterString = Me.OpenArgs
    End If
End Sub
```

To test this example, open a form and apply a filter. Then, click on the Save Filter button in the menu. The frmSavedFilters form should open where you can assign a filter name and description. Now, when you drop down the menu again, the saved filter should be listed, as shown in Figure 11-23.

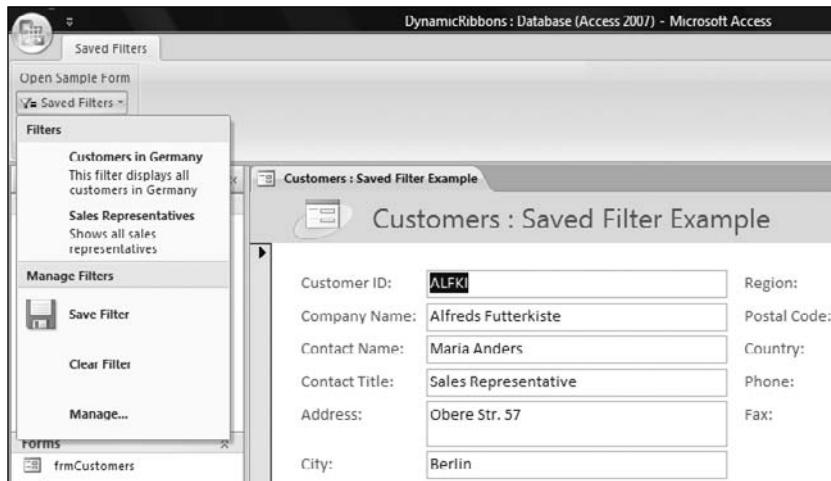


Figure 11-23

Part III: Interacting with the Application

Creating a Split Button That Sticks

Suppose you want to create an application launcher for your Access application. The application launcher gives the user a list of applications that they can launch easily from within *your* application. Here are the requirements:

- ❑ The launcher includes buttons for Word, Excel, Outlook, Calculator and Notepad.
- ❑ The launcher should stick — that is, the last application launched should stick in the button in the customization.

To meet this last requirement, we use a split button. The split button control contains a button and a menu of choices. When an application is launched, we update the button to reflect the last application that was launched. Start by setting up the customization, as follows:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
    onLoad="OnRibbonLoad">
    <ribbon startFromScratch="true">
        <tabs>
            <tab id="tabLauncher" label="Application Launcher">
                <group id="grpLauncher" label="Launcher">
```

Add the split button. The `splitButton` control contains a `button` control that is used as the actual button that is clicked in the Ribbon. This button control should reflect the last application that was launched so we handle the `getLabel` and `getImage` callbacks to update the label and the image respectively.

```
<splitButton id="sbLauncher">
    <button id="btnLauncher"
        getLabel="OnGetAppLabel" getImage="OnGetImage"
        onAction="OnLaunchApplication"/>
```

Now, let's add the menu and close the customization. This is the list of applications that can be launched from the customization. You'll notice that we are using images included with Office in the `imageMso` attribute. We're also storing the name of this image in the `tag` attribute as extra data. More on that in a moment.

```
<menu id="mnuLauncher" label="Launcher" itemSize="normal">
    <menuSeparator id="ms1" title="Office Applications"/>
    <button id="btnWord" label="Word" imageMso="FileSaveAsWordDotx"
        onAction="OnLaunchApplication"
        tag="FileSaveAsWordDotx"/>
    <button id="btnExcel" label="Excel" imageMso="MicrosoftExcel"
        onAction="OnLaunchApplication"
        tag="MicrosoftExcel"/>
    <button id="btnOutlook" label="Outlook" imageMso="MicrosoftOutlook"
        onAction="OnLaunchApplication"
        tag="MicrosoftOutlook"/>
    <menuSeparator id="ms2" title="Utilities"/>
    <button id="btnCalc" label="Calculator" imageMso="Calculator"
        onAction="OnLaunchApplication"
        tag="Calculator"/>
</menu>
</splitButton>
</group>
</tab>
```

Chapter 11: Creating Dynamic Ribbon Customizations

```
</tabs>
</ribbon>
</customUI>
```

The customization is complete, so let's add the callbacks. First, the `OnGetAppLabel` callback. This callback is defined in the launcher button that is updated when a selection is made. Because we're going to update labels and images, we need some private variables in the module to store this information between callbacks. Add the following code to the top of a new module called `basSplitButtonCallbacks`.

```
Private mstrLabel As String
Private mstrImage As String
```

Add the `OnGetAppLabel` callback. If the `mstrLabel` variable has not been set, we default to `Calculator`. As an alternative, you could also retrieve this from a table as a setting.

```
Sub OnGetAppLabel(ctl As IRibbonControl, ByRef Label)
    ' default to Calculator
    If (mstrLabel = "") Then mstrLabel = "Calculator"

    Label = mstrLabel
End Sub
```

Add the `OnGetImage` callback. This callback updates the image in the button to reflect the last application launched. When we defined our customization, we used images included with Office in the `imageMso` attribute. However, you may have noticed that there is a `getImage` callback, but not a `getImageMso` callback. So how can we load a built-in image dynamically? Well, it turns out that while there isn't a `getImageMso` callback, there is a `GetImageMso` method! And, this method was added on the `CommandBars` object of all places. This method returns the image for a named image that is included with Office as you would define in the `imageMso` attribute. So, we can simply use this method to retrieve the image from Office.

```
Sub OnGetImage(ctl As IRibbonControl, ByRef Image)
    ' default to calculator
    If (mstrImage = "") Then mstrImage = "Calculator"
    ' set the image
    Set Image = CommandBars.GetImageMso(mstrImage, 16, 16)
End Sub
```

Okay, the last callback we need is `OnLaunchApplication`, which is called when the user clicks any of the buttons. Remember from our customization that the name of the launcher button is called `btnLauncher`. The buttons in the menu have names specific to their applications. So, clicking the launcher button in the Ribbon opens the application that was most recently launched. However, you need to set that value in `mstrLabel`, which is retrieved from the other buttons. For demo purposes, we are simply showing a message box. You could, however, also store the path to an application to launch from a table based on the button that was clicked.

Earlier we also mentioned that we were storing the name of the `imageMso` attribute as extra data in the `tag` attribute of the button. We use the `tag` value now to set the name of the image in `mstrImage`:

```
Sub OnLaunchApplication(ctl As IRibbonControl)
    ' set the label and the image and invalidate
```

Part III: Interacting with the Application

```
If (ctl.ID <> "btnLauncher") Then
    mstrLabel = Mid(ctl.ID, 4)
    mstrImage = ctl.Tag
    gobjRibbon.InvalidateControl "btnLauncher"
End If

' launch the application
MsgBox "You are launching " & mstrLabel
End Sub
```

When you click the button, you should get a message that says Calculator is being launched. If you select an item in the menu, say Outlook, you should get a message that says Outlook is being launched and the button should update to the Outlook icon.

Other Ribbon Modifications

So far, we've taken an in depth look at creating ribbon customizations using controls such as gallery and button. However, there are other places in the Ribbon that allow for customization. Let's take a look at two of them — the Office menu and built-in commands.

Modifying the Office Menu

You've probably noticed that when you use the `startFromScratch` attribute, some of the controls in the Office menu are hidden, but not all of them. Luckily, you can modify the Office menu if you'd like by modifying controls in the `officeMenu` node of a customization. You can even add your own controls. The following XML defines a customization that adds a button and a menu to the Office menu:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
    loadImage="OnLoadImage">
<ribbon startFromScratch="false">
    <officeMenu>
        <!-- About button -->
        <button id="btnAbout" label="About My Application..." 
            insertBeforeMso="FileNewDatabase" imageMso="Info" 
            onAction="=MsgBox('My Application - copyright 2007')"/>
        <!-- Menu -->
        <menu id="mnu1" label="Application Launcher">
            insertBeforeMso="FileNewDatabase">
                <checkBox id="chk1" label="Disable Launcher"/>
                <menuSeparator id="ms1" title="Office Applications"/>
                <button id="btnWord" label="Project" imageMso="MicrosoftProject"/>
                <button id="btnExcel" label="Excel" imageMso="MicrosoftExcel"/>
                <button id="btnOutlook" label="Outlook" imageMso="MicrosoftOutlook"/>
                <menuSeparator id="ms2" title="Utilities"/>
                <button id="btnCalc" label="Calculator" imageMso="Calculator"/>
                <button id="btnNotepad" label="Notepad" imageMso="ReviewEditComment"/>
                <button id="btnPhotos" label="My Pictures" imageMso="Camera"/>
            </menu>
        </officeMenu>
    </ribbon>
</customUI>
```

Chapter 11: Creating Dynamic Ribbon Customizations

The modified Office menu is shown in Figure 11-24.



Figure 11-24

Overriding Commands and Repurposing Controls

The Access Ribbon provides a wealth of built-in functionality. For example, there are buttons in the Office menu to create new databases or open an existing database. Other commands are available in the Ribbon itself, such as the new Encrypt with Password button. With all of these commands, there may be times when you want to include behavior or controls that Access provides in your applications. There may be other times when you want to override the default behavior of a control that Access provides. This is called *repurposing* a control. For example, consider a basic Contacts form in a contact tracking application. If you are deploying this application using the Access runtime you will likely want to create a ribbon customization for the application so users can interact with it. To prevent from implementing your own functionality for filtering or finding records, you can include the groups that are defined by Access, which provides these capabilities.

To include or modify existing controls, you need the `idMso` attribute. For commands that are not in the Office menu, use the `commands` node under the `customUI` root node of the ribbon customization, as shown in the code that follows. For instance, say that we want to disable the `New` button in the Office menu so that users cannot create new databases when they are using your application. The following customization will do this:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
    <commands>
        <command idMso="FileNewDatabase" enabled="false"/>
    </commands>
</customUI>
```

Part III: Interacting with the Application

Disabling controls is one way to change functionality, but the Ribbon also enables you to write your own code when a built-in control is clicked. The signature for callback routines changes when you repurpose a built-in control. Let's say that you wanted to enable users to encrypt their databases by adding a database password. Access provides this functionality using the Encrypt with Password button for the ACCDB file format so you don't have to create this yourself. However, Access gives an error when the database is not opened exclusively. Thus, we want to replace the Access error message. So, let's repurpose the control.

Again, start with the customization, as follows:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
    <commands>
        <command idMso="SetDatabasePassword" onAction="OnSetPassword"/>
    </commands>
</customUI>
```

To repurpose an existing control, specify an `onAction` callback with an `idMso` control. Now, add the `OnSetPassword` callback. Notice that the `onAction` callback includes an extra parameter called `CancelDefault`.

```
Public Sub OnSetPassword(control As IRibbonControl, ByRef CancelDefault)
    If (CurrentProject.Connection.Properties!Mode = 12) Then
        CancelDefault = False
    Else
        MsgBox "You must open the database exclusively to set the password.", _
            vbExclamation, "Cannot Set Password"
        CancelDefault = True
    End If
End Sub
```

We are using the `Mode` property of the ADO `Connection` object for the database that is currently open. When the value of this property is 12, the database is open exclusively. We set the `CancelDefault` argument to `False` to let the built-in command run. When the property is not 12, we display a message and cancel the built-in command by setting the `CancelDefault` argument to `True`.

Now, if you click the Encrypt with Password button on the Database Tools tab in the Access Ribbon, you should receive the message box when the database is not opened exclusively.

Summary

The Ribbon, introduced in Office 2007, will undoubtedly change the way we look at user interface development in the future. It creates opportunities for us as developers to put the things in front of our users that really matter and help them do their jobs effectively. Sure, the model has changed, but we feel this model creates enough new opportunities for development and once you have learned them you might even be able to add new skill sets under your belt such as XML!

Chapter 11: Creating Dynamic Ribbon Customizations

Here are some of the key points in this chapter:

- The programming model for the Ribbon has changed. This will likely have an effect on how we perform user interface development.
- There are many different controls that can be used to create rich user experiences.
- Many resources are available for Ribbon development in Office 2007.
- Using callback functions, you can provide the same functionality for several Access events in a ribbon customization.
- Programming the Ribbon is different, but fun!

In the next chapter, we discuss configuration and extensibility, and go into the details of creating Access applications that can be configured by users, and even localized!

Part IV: Finalizing the Application

Chapter 12: Configuration and Extensibility

Chapter 13: Personalization and Security

Chapter 14: Deployment

Chapter 15: Help and Documentation

12

Configuration and Extensibility

If you're writing an off-the-shelf application or even an in-house application, it might not always be clear how all of your customers will use your product. Many times, users need to determine how they work best. To enable users to work in a way they feel most comfortable, you can add the capability to customize your application. In this chapter, we explore how to create applications that are flexible and that give users the ability to configure them to meet their needs. More specifically, in this chapter you learn:

- How to create localized Access applications that can be used in locales other than your own
- How to create options for your applications to provide users with choices for configuration that meet their needs
- How to create styles for your forms and reports to provide a uniform look throughout your application

Localization

The marketplace is constantly expanding and opportunities for development can sometimes arise in countries other than your own. When such opportunities present themselves, you should keep in mind that users tend to prefer to work in their native languages. Localization is the process of adapting the interface for a given software product to a language other than the one it was authored in. For Access applications, this could mean providing alternate translations for forms and reports, and changing the way that dates and numbers are represented for a given location.

Part IV: Finalizing the Application

Locale Settings

Two different locale settings are used in Windows. The first is called the *user locale*, which represents the settings specified by a given user on a computer. These settings include the display formats for numbers, dates, and currency. The second locale setting is called the *system locale*. The system locale is used to determine how text will be displayed by non-Unicode applications. Generally speaking, when you see the term *locale*, the user locale is what is being referred to.

Using the API to Determine the Locale

Windows provides information about locale settings in a few API functions. One function that we use frequently is `GetUserDefaultLCID`. This function returns the locale identifier (LCID) of the user locale for the current user.

```
Public Declare Function GetUserDefaultLCID Lib "kernel32" () As Long
```

The real workhorse of locale API functions, however, is `GetLocaleInfo`. This function returns specific information about a given locale such as the decimal separator, localized locale name, or currency symbol. We'll show you how to use this function with some of the pieces of data that we retrieve frequently. Refer to the `WinNls.h` header file in the Windows SDK for a complete listing of possible values to retrieve.

In order to tell the `GetLocaleInfo` function the piece of locale information that you are interested in, you'll need to define some constants. We've defined these as an `Enum` to provide IntelliSense for our wrapper function. These are some of the more common constants that we use in our applications:

```
Public Enum LCTYPE
    LOCALE_SLIST = &HC          ' // list item separator
    LOCALE_SDECIMAL = &HE        ' // decimal separator
    LOCALE_SLANGUAGE = &H2       ' // localized name of language
    LOCALE_SENGLANGUAGE = &H1001 ' // English name of language
    LOCALE_SNATIVELANGNAME = &H4 ' // native name of language
    LOCALE_ICOUNTRY = &H5         ' // country code
    LOCALE_SCOUNTRY = &H6        ' // localized name of country
    LOCALE_SENGCOUNTRY = &H1002  ' // English name of country
    LOCALE_SNATIVECTRYNAME = &H8 ' // native name of country
    LOCALE_SCURRENCY = &H14      ' // local monetary symbol
    LOCALE_SSHORTDATE = &H1F     ' // short date format string
    LOCALE_SISO639LANGNAME = &H59 ' // ISO abbreviated language name
    LOCALE_SISO3166CTRYNAME = &H5A ' // ISO abbreviated country name
End Enum
```

Next, add the `Declare` statement:

```
Private Declare Function GetLocaleInfo Lib "kernel32" Alias "GetLocaleInfoA" _
    (ByVal Locale As Long, _
    ByVal lngLCTYPE As LCTYPE, _
    ByVal lpLCData As String, _
    ByVal cchData As Long) As Long
```

By default, the `GetLocaleInfo` function retrieves locale information as a string, which means that the return value of the function is the number of characters that it fills in the buffer.

And last, the wrapper function:

```
Public Function GetLocaleInformation(lcid As Long, lc As LCTYPE) As String
    Dim buffer As String
    Dim pos      As Long
    Dim cch      As Long

    buffer = Space(255)          ' allocate some space
    cch = Len(buffer) + 1        ' size of the buffer (cch = count of characters)

    pos = GetLocaleInfo(lcid, lc, buffer, cch)
    If (pos > 0 And cch > 0) Then
        GetLocaleInformation = Left(buffer, pos - 1)
    End If
End Function
```

To retrieve information about the current user locale, pass in the return value of `GetUserDefaultLCID` to the first argument of the `GetLocaleInformation` wrapper function. For example, to retrieve the ISO country code for the current locale, you could write a function such as:

```
Function GetISOCountryCode() As String
    Dim lngLCID As Long

    ' get the current LCID
    lngLCID = GetUserDefaultLCID()

    ' get the country code
    GetISOCountryCode = GetLocaleInformation(lngLCID, LOCALE_SISO639LANGNAME) & _
                        " - " & _
                        GetLocaleInformation(lngLCID, LOCALE_SISO3166CTRYNAME)
End Function
```

On a US English computer, this function returns:

en-US

Changing the User Locale

To change the user locale for testing, launch the Regional and Language Options dialog box in Windows Vista as follows:

1. Open the Control Panel.
2. Choose Clock, Language, and Region, and then select Regional and Language Options. If you are using Classic view, choose Regional and Language Options.
3. This should open the Regional and Language Options dialog box, as shown in Figure 12-1.
4. Change the Current format setting for your computer on the Formats tab of the Regional and Language Options dialog box in Windows Vista to change the user locale.

Part IV: Finalizing the Application



Figure 12-1

Determining the Current Language of Office

Office provides an object in the object model for each application called `LanguageSettings`. This object provides information about the language currently being used by an Office application. When you are considering localization of your Access application, you should typically use the user interface (UI) language of the current version of Access. This can be determined using the following code:

```
Application.LanguageSettings.LanguageID(msoLanguageIDUI)
```

This requires a reference to the Microsoft Office 12.0 Object Library to work.

The `LanguageID` property returns the locale identifier (LCID) for the current UI language of the Office application. Use the LCID value to determine the language that the user is currently using. The list of LCIDs used by Windows can be found on MSDN at <http://msdn.microsoft.com/library/en-us/script56/html/882caleb-81b6-4a73-839d-154c6440bf70.asp>.

How to Create Localized Access Applications

Depending on what you want to localize, there are a few different techniques for creating localized Access applications.

VBA Intrinsic Functions

Starting with VBA6 that was released with Access 2000, VBA has provided functions that help you create global applications:

- MonthName
- WeekdayName
- FormatCurrency
- FormatDateTime
- FormatNumber

Each of these functions uses the regional settings of the computer in order to return its data. For example, let's say that you have a form that displays the months of the year in a list box named `lstMonths`. This might look something like the form shown in Figure 12-2.

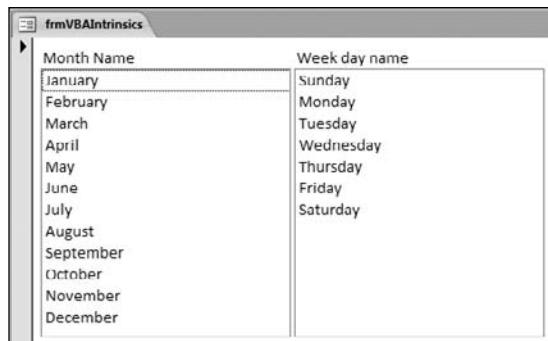


Figure 12-2

Because the months of the year don't change, you might choose to make this a static list using a Value List row source type of the list box control. But, what if you were selling your application in a different locale and wanted the names of the months to appear in the language of the user? A static property would present problems so instead we'll use the `MonthName` function in VBA to fill the list dynamically. Add the following code to the Load event of the form to fill in the list of months:

```
Private Sub Form_Load()
    ' add the months
    Dim i As Integer

    ' clear the listbox
    Me.lstMonths.RowSource = ""

    ' add the months
    For i = 1 To 12
        Me.lstMonths.AddItem MonthName(i)
    Next
End Sub
```

Part IV: Finalizing the Application

The cool thing about the `MonthName` function is that it returns the localized name of the month given the user locale. For example, if you change the user locale to Spanish (Spain) using the Regional and Language Options dialog box, the form should now display the names of the months in Spanish, as shown in Figure 12-3.



Figure 12-3

Localized Text Using Tables

Of course, you are more likely to want to localize a larger portion of your UI than the names of months or days of the week. Chances are you'll want to localize label captions or other text in your application such as error messages. To do this, you'll need to store the localized strings somewhere. The easiest place to store these strings is in a table in your database, but of course this adds to the size of the application. To prevent the database from bloating, use an external database to store the localized text.

Creating the String Table

This table is essentially a table of strings, or a string table. Let's assume for our example that we want to localize our application in English, German, and Spanish. To do this, we'll create the following table and name it `USysTblStrings` so that Access treats it as a system table in our application.

Field Name	Data Type	Description
ID	AutoNumber	Primary Key
FormName	Text (64)	
ControlName	Text (255)	
1033	Memo	English (United States)
1031	Memo	German
1034	Memo	Spanish (Spain)

The FormName and ControlName fields will be used to localize a particular control on a given form. Notice that we've created separate fields for each LCID to localize. When we retrieve the strings from this table, we include the given LCID field.

This table will not be related to other tables so we have purposefully created a de-normalized table structure to reduce the amount of records in the table.

Retrieving Strings from the String Table

When you store localized strings in a table in a database, retrieving strings is as simple as using a Recordset object.

This example is available for download on the Web page for this book at www.wrox.com. The sample file is called LocalizationTest.accdb and the sample form is frmContacts. This database contains additional utility code to fill the string table.

For this example, create a new table based on the Contacts table template in Access 2007. Name the table tblContacts. Next, create a form based on this table using the Form button on the Create tab in the Access Ribbon. Name this form frmContacts. This should give you a form similar to the one shown in Figure 12-4.

Figure 12-4

Next, fill the string table with captions from the form. This table contains the control names that we have renamed as well to correspond with the captions. Before you test the code, be sure to rename the labels to match those shown in the table that follows.

Part IV: Finalizing the Application

FormName	ControlName	1033	1031	1034
frmContacts	lblCompany	Company	Firma	Compañía
frmContacts	lblLastName	Last Name	Nachname	Nombre Pasado
frmContacts	lblFirstName	First Name	Vorname	Nombre
frmContacts	lblEmailAddress	e-mail Address	e-mail Adresse	Dirección Del e-mail
frmContacts	lblJobTitle	Job Title	Arbeitstitel	Título Del Trabajo
frmContacts	lblBusinessPhone	Business Phone	Geschäft Telefon	Teléfono Del Negocio
frmContacts	lblHomePhone	Home Phone	Haustelefon	Teléfono Casero
frmContacts	lblMobilePhone	Mobile Phone	Handy	Teléfono De La Célula
frmContacts	lblFaxNumber	Fax Number	Fax-Nummer	Número De Fax
frmContacts	lblAddress	Address	Adresse	Dirección
frmContacts	lblCity	City	Stadt	Ciudad
frmContacts	lblProvince	State/Province	Provinz	Provincia
frmContacts	lblPostalCode	Postal Code	Postleitzahl	Código Postal
frmContacts	lblCountry	Country	Land	País
frmContacts	lblNotes	Notes	Anmerkungen	Notas

Once you've filled the table, you should have a string table that looks something like Figure 12-5.

Now that there is data in the string table, you need to retrieve the strings for a given form. To do this, you'll create a procedure in a new standard module called `basLocalize` that accepts a `Form` object as a parameter and the path to an external database that contains the `usystblStrings` table. Include the declarations for the procedure as follows:

```
Public Sub Localize(objForm As Form, Optional stLocalizedDb As String)
    Dim db      As DAO.Database
    Dim rs      As DAO.Recordset
    Dim ctl     As Label
    Dim strSQL  As String
    Dim lngLCID As Long
```

ID	FormName	ControlName	1033	1031	1034
1	frmContacts	lblCompany	Company	Firma	Compañía
2	frmContacts	lblLastName	Last Name	Nachname	Nombre Pasado
3	frmContacts	lblFirstName	First Name	Vorname	Nombre
4	frmContacts	lblEmailAddress	E-mail Address	E-mail Adresse	Dirección Del E-mail
5	frmContacts	lblJobTitle	Job Title	Arbeitstitel	Titulo Del Trabajo
6	frmContacts	lblBusinessPhone	Business Phone	Geschäft Telefon	Teléfono Del Negocio
7	frmContacts	lblHomePhone	Home Phone	Haustelefon	Teléfono Casero
8	frmContacts	lblMobilePhone	Mobile Phone	Handy	Teléfono De La Célula
9	frmContacts	lblFaxNumber	Fax Number	Fax Nummer	Número De Fax
10	frmContacts	lblAddress	Address	Adresse	Dirección
11	frmContacts	lblCity	City	Stadt	Ciudad
12	frmContacts	lblProvince	State/Province	Provinz	Provincia
13	frmContacts	lblPostalCode	Postal Code	Postleitzahl	Código Postal
14	frmContacts	lblCountry	Country	Land	País
15	frmContacts	lblNotes	Notes	Anmerkungen	Notas

Figure 12-5

You need to get the DAO Database object for the database that contains the string table. If there is no external database specified, you'll default to the current database.

```
' get the database
If (Len(stLocalizedDb) = 0) Then
    Set db = CurrentDb()
Else
    Set db = DBEngine.OpenDatabase(stLocalizedDb)
End If
```

Next, cache the current language using the LanguageID property shown earlier. This will return the LCID that will determine which field to use in the string table.

```
' get the LCID for the current language
' used to determine which field to select
lngLCID = LanguageSettings.LanguageID(msoLanguageIDUI)
```

Now, you need to get the strings. To do this, select the form name, control name, and the appropriate LCID field that contains the localized text. Restrict the recordset to the current form.

```
' get the strings for the specified form
strSQL = "SELECT [FormName], [ControlName], [" & lngLCID & "]"
strSQL = strSQL & " FROM usystblStrings"
strSQL = strSQL & " WHERE FormName = '" & objForm.Name & "'"
Set rs = db.OpenRecordset(strSQL)
```

Last, you need to fill the captions. This code assumes you are changing the Caption property for Label objects on the form that was passed as an argument to the Localize procedure. Because you are storing the control name in the string table, you can get the appropriate control using the Controls collection of the form. Again, you use the LCID as the field in the recordset for the string table that contains the actual caption.

```
' fill the label captions
While (Not rs.EOF)
```

Part IV: Finalizing the Application

```
' get the label and set the caption
Set ctl = objForm.Controls(rs("ControlName"))
ctl.Caption = rs.Fields("[ " & lngLCID & " ]")
rs.MoveNext
Wend
```

Last, add some cleanup code and end the procedure:

```
' cleanup
rs.Close
db.Close

Set ctl = Nothing
Set rs = Nothing
Set db = Nothing
End Sub
```

To call this routine, add code to the Load event of frmContacts that calls the Localize procedure:

```
Private Sub Form_Load()
    ' localize the label captions
    Localize Me
End Sub
```

To change the language for an Office application, you'll need to have a *language pack* installed. Language packs are add-on pieces of software available from Microsoft that contain localized strings for the Office applications themselves. Once you have installed a language pack, you can change the UI language for Office using the Microsoft Office 2007 Language Settings tool, as shown in Figure 12-6.

In this example, we have changed the UI for Office to German. This requires closing the application and re-opening. When you re-open Access and open the database, the label captions should appear in German, as shown in Figure 12-7.

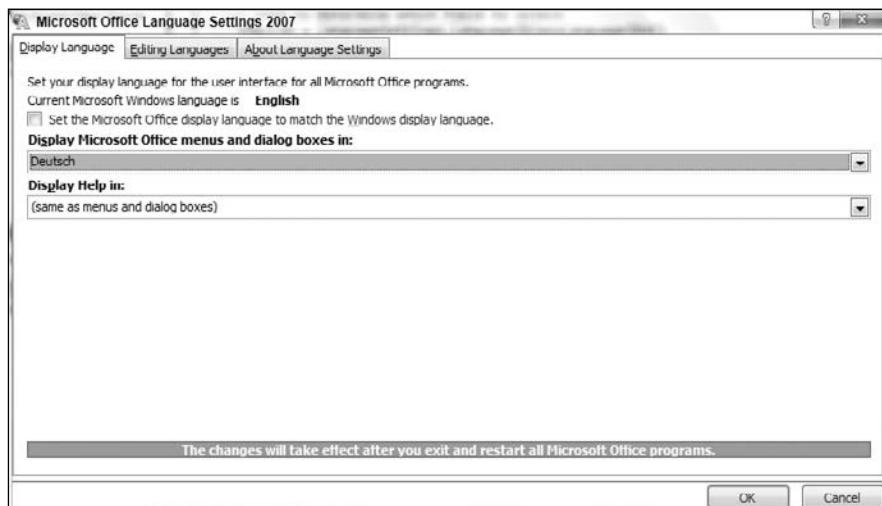


Figure 12-6

Label	Field
Firma	Adresse
Nachname	Stadt
Vorname	Provinz
E-mail Adresse	Postleitzahl
Arbeitstitel	Land
Geschäft Telefon	Anmerkungen
Haustelefon	
Handy	
Fax-Nummer	

Figure 12-7

When you are localizing applications, make sure that you have enough space for the localized text to avoid truncating text. The length of text in other languages is often not the same as the length in your native language.

Creating a Resource DLL

Creating a string table in a database is pretty easy but it does add some size to the database as we mentioned earlier. In addition, retrieving strings has a minimal amount of overhead because we are loading recordsets. You can avoid these issues altogether by creating a *resource DLL*. A resource is a piece of content such as a string, cursor, bitmap, or icon. A DLL that stores resources only is called a resource DLL. If you are using Visual Studio, you don't even need to know C++ to create one!

Access itself uses a resource DLL called msain.dll to retrieve its own strings. Resources are retrieved from a resource DLL using Windows API functions and are retrieved very quickly. When using resource DLLs, you typically will create one DLL for each localized version that you deploy. We'll discuss deployment of a resource DLL in a few moments. The tradeoff here is that while this technique reduces the overhead of objects and database storage, it requires that you deploy an extra file with your application.

Let's create the DLL using Visual Studio 2005. These steps require Visual C++ 2005 but knowledge of the C++ language is not required. You can also use Visual C++ 2005 Express Edition, which is available as a free download from the Microsoft Web site.

- 1.** Launch Visual Studio 2005 or Visual C++ 2005 and create a new project.
- 2.** In the New Project dialog box, choose Visual C++, and then select Win32 and choose Win32 Project. Name the project **MyAppResources**, as shown in Figure 12-8.

Part IV: Finalizing the Application

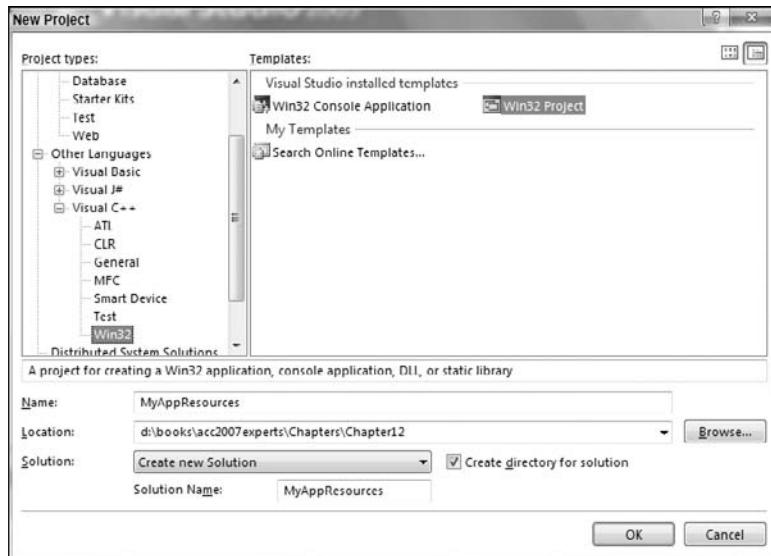


Figure 12-8

3. Click Next in the Win32 Application Wizard, and then select DLL as the Application type and click Finish. This should create the DllMain entry point function in C++. This function is called by Windows when a DLL is loaded.
4. You need to create a string table to store the strings. To do this, click Project, and then click Add Resource. This opens the Add Resource dialog box. Select String Table in the dialog box and click New, as shown in Figure 12-9.

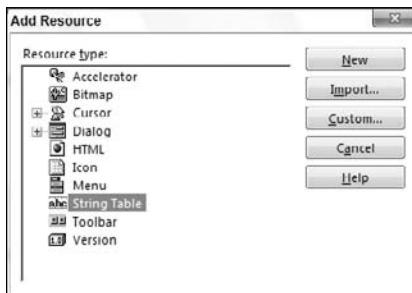


Figure 12-9

5. At this point, you should have a string table resource in the DLL so let's create some strings. This time, instead of localizing label captions you'll localize error messages. Suppose you are writing a contact tracking application and want to ensure that the user enters the first and last name. If these values are not supplied, you'll display an error message. Create two strings in the string table, as shown in Figure 12-10.

ID	Value	Caption
IDA_NOFIRSTNAME	101	Please enter the first name of the contact
IDA_NOLASTNAME	102	Please enter the last name of the contact

Figure 12-10

6. Select Build Solution from the Build menu to build the project. This will create the DLL file with your string table.

We are using a convention in naming the resource identifiers. The prefix `IDA` is used to identify errors and stands for “ID of an alert.” For strings in the application, we use the prefix `IDS`, which stands for “ID of a string.”

There is one more resource to add to this DLL and that is a version resource. The version resource is used to provide additional information about your product in the Properties dialog box for a file in Windows Explorer. One of the fields included in the version resource is the language.

Click Add Resource from the Project menu in Visual Studio to add a version resource. In the version resource editor, change the fields as you’d like them to appear. The specified language of the DLL is included in the Block Header field, as shown in Figure 12-11. Choose the appropriate language for your DLL.

After you build the DLL, open the folder where the DLL resides in Windows Explorer and display the Properties dialog box for the DLL file. You should see something like the dialog box shown in Figure 12-12.

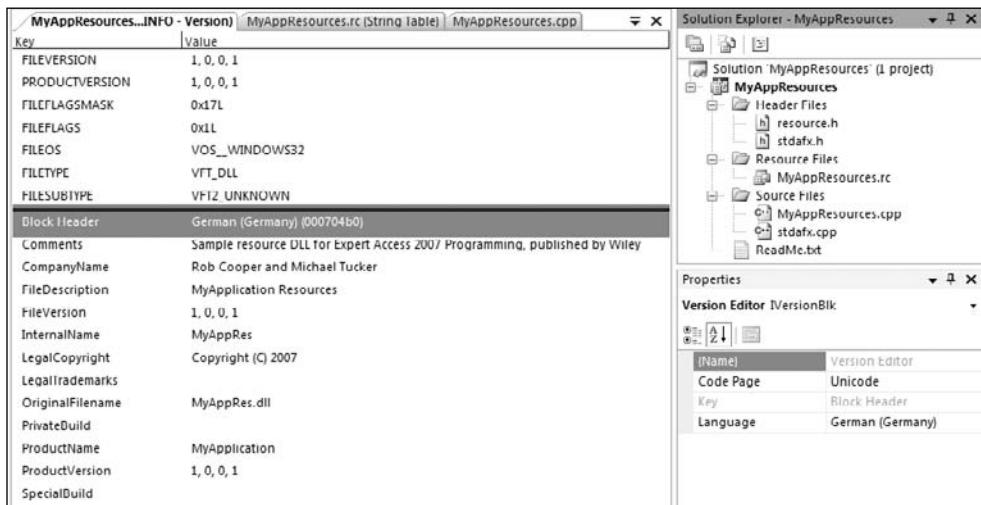


Figure 12-11

Part IV: Finalizing the Application

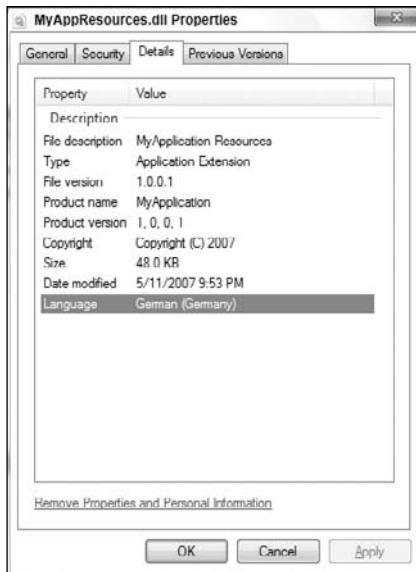


Figure 12-12

Deploying a Resource DLL

Earlier we mentioned that there is a deployment tradeoff when using a resource DLL. This is because you'll need to install an extra file along with your application. Furthermore, you'll also need to keep in mind the language of the DLL when you install it.

Access installs the `msain.dll` resource file in a subfolder of the `Office12` directory where Access itself is installed. This subfolder is the LCID for the language version of the DLL. We will do the same for the code example in the next section. Unless you are using an automated build process, you'll need to manually copy each language DLL to a separate directory for testing. For the upcoming example, create two folders under the directory where the test database resides: 1031 for German (Germany), and 1033 for English (United States).

Reading Resources Using the `LoadString` API

Once you've built the resource DLL, you need to read the strings from the string table. That is after all why we created it! There are three Windows API functions to use to read a string from a resource DLL:

- ❑ `LoadLibrary` — Loads the DLL into memory
- ❑ `LoadString` — Reads a string from a string table
- ❑ `FreeLibrary` — Frees a DLL from memory that was loaded using `LoadLibrary`

For this example we're going to use the `frmContacts` form that was created earlier in the section "Localized Text Using Tables." These functions are very straightforward but you'll need some supporting events and code to enable the scenario. Start with the API declarations:

```
Private Declare Function LoadLibrary Lib "kernel32.dll" Alias "LoadLibraryA" _  
(ByVal sFile As String) As Long
```

```
Private Declare Sub FreeLibrary Lib "kernel32.dll" (ByVal hInstance As Long)
Private Declare Function LoadString Lib "user32.dll" Alias "LoadStringA" _
    (ByVal hInstance As Long, _
     ByVal uID As Long, _
     ByVal lpBuffer As String, _
     ByVal nBufferMax As Long) As Long
```

Next, declare a Long integer value to store the handle to the DLL. This is known as an *instance handle* or a *module handle*:

```
' handle to the DLL
Private hModule As Long
```

VBA cannot use the named resource identifiers directly so we'll re-create them using an `Enum`. This is to avoid magic numbers in our code.

```
' error constants
Private Enum ResourceIdentifiers
    IDA_NOFIRSTNAME = 101
    IDA_NOLASTNAME = 102
End Enum
```

To prevent multiple loads of the resource DLL, we'll load it once using the `Load` event of the form. Remember from the earlier example that this form also uses localized label captions.

```
Private Sub Form_Load()
    Dim sResourceDll As String

    ' localize the label captions
    Localize Me

    ' load the module
    sResourceDll = CurrentProject.Path & "\& _
        LanguageSettings.LanguageID(msoLanguageIDUI) & _
        "\MyAppResources.dll"

    hModule = LoadLibrary(sResourceDll)

    ' make sure the module loaded successfully
    Debug.Assert hModule <> 0
End Sub
```

The path to the resource DLL is defined as a subdirectory for the LCID under the directory where the database resides. This is determined using the `CurrentProject.Path` property. The `LoadLibrary` API function returns the handle to the DLL. If it fails to load the DLL, it will return zero.

Because we've loaded the DLL, we should free it. Do this in the `Close` event of the form by calling the `FreeLibrary` API function.

```
Private Sub Form_Close()
    ' close the DLL
```

Part IV: Finalizing the Application

```
    FreeLibrary hModule  
End Sub
```

To read the string we'll use the `LoadString` API function. The following function will allocate space to receive data from Windows and call the function. The `LoadString` function expects the numeric resource identifier values that you defined in the string table of the DLL. We mirrored these values using the `Enum`.

```
Private Function GetResourceString(lStringId As Long) As String  
    Dim buffer As String  
  
    ' allocate some space  
    buffer = Space(1024)  
  
    If (hModule) Then  
        If (LoadString(hModule, lStringId, buffer, Len(buffer) + 1)) Then  
            GetResourceString = Left(buffer, InStr(buffer, vbNullChar) - 1)  
        End If  
    End If  
End Function
```

Last, because you want to display an error message to the user, add the following code in the `BeforeUpdate` event of the form. This event will check the first name and last name fields and display an error message using the `GetResourceString` function with the specified resource identifier value.

```
Private Sub Form_BeforeUpdate(Cancel As Integer)  
    ' make sure that the first name and last name have been entered  
    ' loads error strings from a resource DLL  
    If (IsNull(Me.[First Name])) Then  
        MsgBox GetResourceString(IDA_NOFIRSTNAME), vbExclamation  
        Cancel = True  
    ElseIf (IsNull(Me.[Last Name])) Then  
        MsgBox GetResourceString(IDA_NOLASTNAME), vbExclamation  
        Cancel = True  
    End If  
End Sub
```

To test this routine in another language, you need a language pack installed for Office. You can test it using your native language provided that you have copied the resource DLL to a subdirectory of the folder where the database resides that is the LCID for your current UI language. When you save a record without setting the first name you should see an error, as shown in Figure 12-13.

Application Options

Adding options to your applications enables your users to customize the application to best suit them. Say, for example, that you wanted to allow the user to change the look of the application by using a theme or a skin. Using options gives the user a place to change the theme. Office 2007 includes a new Color scheme option, which provides this functionality, as shown in the Access Options dialog box in Figure 12-14.

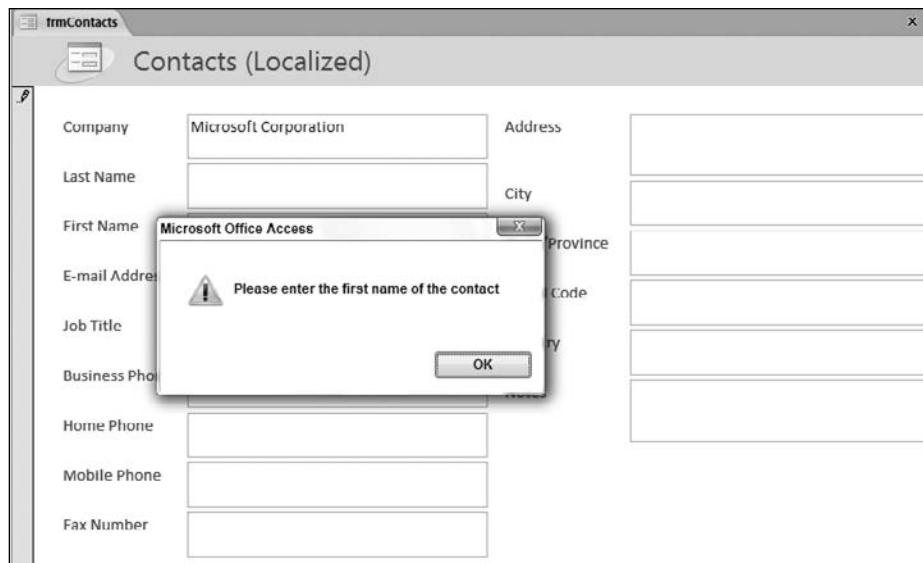


Figure 12-13

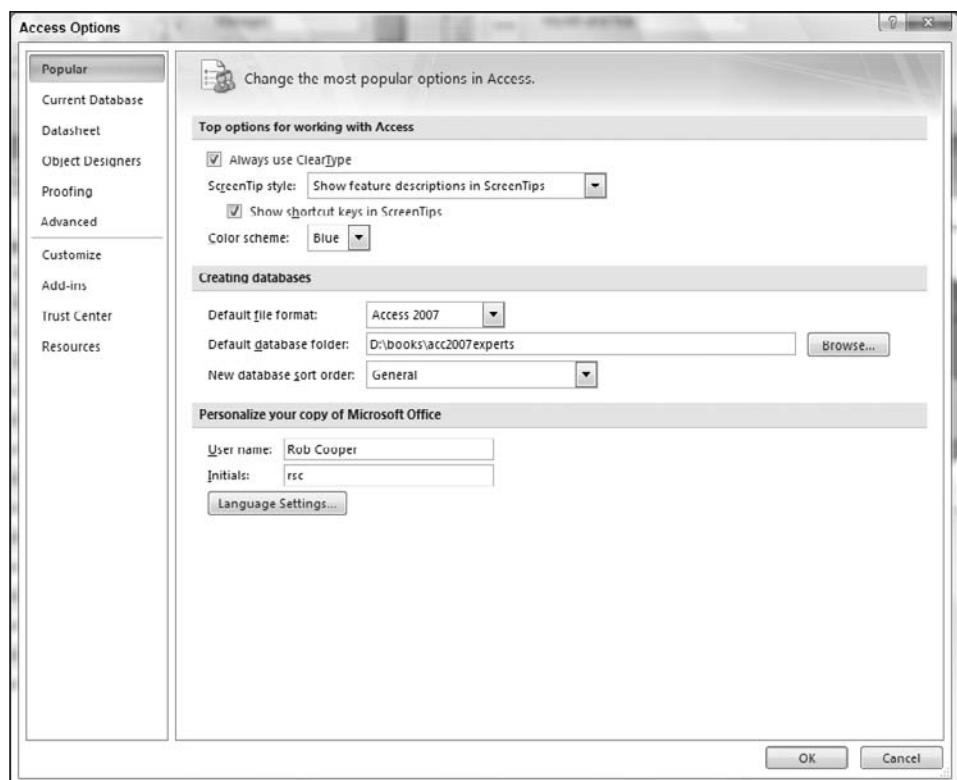


Figure 12-14

Part IV: Finalizing the Application

You might also use application options for the following purposes:

- To include a personalized message to the customer or the signature line for the user in an application that lets the user send e-mail to customers.
- To give the user a choice to maximize reports in print preview.
- To open the last form that was opened on startup of the application. The name of the form that was last opened is also stored but not displayed to the user.
- To include charts on a dashboard form that might take a long time to load.
- To provide the ability to turn off the tracking feature in a report manager feature that tracks when each report is opened.

As you can see, there are many opportunities to provide options to your users. Using options can increase user productivity to standardize common tasks and provides a personal feel to an application. Let's look at two ways to incorporate options into your applications.

Storing Options in Tables

As was the case with localizing Access applications, using a table in the database is the easiest way to create options. Application options are typically key/value pairs so the table is very straightforward. We also add a category to the options table, which is used to group options together. Create a new table called `USysTblOptions` as follows:

Field Name	Field Size	Notes
ID	AutoNumber	Primary Key
OptionName	Text (50)	
OptionCategory	Text (255)	
OptionValue	Text (255)	

This table is included in the download files that correspond to this book at www.wrox.com. The name of the sample file is OptionsAndThemesTest.accdb.

Storing Options in the Registry

If you are creating more options than you'd like to store in a table, you might consider storing application options in the Registry. There are three ways to write to the Registry from an application. The first is to use the `SaveSetting` statement in VBA. This statement is easy to use but is limited to one section of the Registry: `HKEY_CURRENT_USER\Software\VB and VBA Program Settings`. In addition, this method only writes strings to the Registry, it does not write numbers. For more control over the branding of your application or the values that are written, you'll need to use another technique. The second approach you can use is to use the `WshShell` object defined in the Windows Scripting Host. This object defines a method called `RegWrite` that is used to write to the Registry. The third method used to write to the Registry is the Windows API. Windows provides many functions that allow you to write to the Registry.

For the examples in the next section, we'll use a table and the Windows Scripting Host because it is available on most computers. To avoid an additional reference in our application, we'll use late-binding to the `WshShell` object.

For detailed information about the Registry API functions, please refer to the Access 2007 VBA Programmer's Reference, Wrox, 2007.

Creating a Class to Work with Options

As you've seen throughout this book, using a class module can simplify programming tasks, and application options or settings are no exception. We'll use a class module to hide the details of retrieving and setting options in the application. This allows you to change the mechanism for storing settings without changing the code that calls it. To help facilitate this, we use constants in our code that represent the option names. This serves two purposes:

- ❑ It helps to prevent hard-coded strings in the application.
- ❑ It provides flexibility of option names. When you store options in the Registry, unless you store all option names in the same hive, you might have to change the option name to designate a hive under your main hive in the Registry.

If you had a complex application where options were stored in multiple locations (for example user and system options), you might abstract these classes further into an interface as described in Chapter 3.

Creating the Option Constants

Before we get into the details of creating the class, let's create the constants for the option names. Constants for the option names are stored in a standard module named `basConstants`. Add the following declarations to this module. The values listed here appear in `USysTblOptions` in the `OptionName` field. We use these constants in code instead of hard-coded strings.

```
' option name constants
Public Const AEP_OPT_LASTSUBFORMOBJ As String = "LastFormObject"
Public Const AEP_OPT_SHOWLASTSFRM As String = "ShowLastFormOnStartup"
Public Const AEP_OPT_MAXIMIZERPTS As String = "MaximizeReports"
Public Const AEP_OPT_APPTHEME As String = "ApplicationTheme"
```

The AEP prefix in these constants stands for Access Expert Programming.

Managing Options from Tables

Once you've created the table described earlier, create a new class module called `Settings`. This class will contain three methods. The first method is called `OptionExists` and it is used to determine whether an option exists in the table. For simplicity, we'll use the `DLookup` function in Access to make this determination. Before adding this method, add the following constant declaration to the `Settings` class module. This stores the name of the options table in the database and is used throughout the class.

```
Private Const CON_SETTINGS_TABLE As String = "USysTblOptions"
```

Part IV: Finalizing the Application

Next, add the OptionExists method.

```
Public Function OptionExists(stOptName As String) As Boolean
    ' Determines if an option exists in USysTblOptions
    OptionExists = Not IsNull(DLookup("OptionName", CON_SETTINGS_TABLE, _
        "OptionName=' " & stOptName & "'"))
End Function
```

Notice that you've defined this function as `Public`, but it can be marked as `Private` if you didn't want to expose it to the rest of your application. The next method is used to write an option value and is called `SetOptionValue`. This method accepts the name and value of an option to set.

```
Public Sub SetOptionValue(stOptName As String, stOptValue As String)
    ' Sets a property in USysTblOptions
    Dim stSQL As String

    ' If the option exists, update it, otherwise add it
    If (OptionExists(stOptName)) Then
        stSQL = "UPDATE " & CON_SETTINGS_TABLE
        stSQL = stSQL & " SET OptionValue = '" & stOptValue
        stSQL = stSQL & "' WHERE OptionName=' " & stOptName & "'"
    Else
        stSQL = "INSERT INTO " & CON_SETTINGS_TABLE
        stSQL = stSQL & " (OptionName, OptionValue)"
        stSQL = stSQL & " VALUES ('" & stOptName & "', "
        stSQL = stSQL & "'" & stOptValue & "')"
    End If

    CurrentProject.Connection.Execute stSQL
End Sub
```

The `SetOptionValue` method uses the `OptionExists` method to determine if an option exists. If the option exists, you update it using the `UPDATE` statement in SQL. If the option does not exist, you add it to the table using an `INSERT INTO` statement.

The third and final method in the class is called `GetOptionValue` and is used to retrieve an option value from the `USysTblOptions` table. Again for simplicity we've used the `DLookup` function.

```
Public Function GetOptionValue(stOptName As String) As Variant
    ' Gets a property from USysTblOptions
    GetOptionValue = DLookup("OptionValue", CON_SETTINGS_TABLE, _
        "OptionName=' " & stOptName & "'")
End Function
```

Managing Options in the Registry

To manage options in the Registry, we'll use the `wshShell` object defined by the Windows Scripting Host. This gives you flexibility without the complexity of the Registry API functions. To prevent breaking client code, we'll create the same methods defined in the previous section: `OptionExists`, `SetOptionValue`, and `GetOptionValue`. Our keys in the Registry will be defined in a given hive. If you use keys under the root for the settings you'll need to change the parameters used in these methods. We recommend using constants to store the option names to limit the amount of changes required to client code for this scenario.

Chapter 12: Configuration and Extensibility

Create a new class called `SettingsRegistry`. You'll need to define the hive that will contain the root for your settings so define the following constants in the declarations section:

```
Private Const SETTINGS_COMPANY_ROOT As String = "HKCU\Software\MyCompany"
Private Const SETTINGS_APP_ROOT      As String = SETTINGS_COMPANY_ROOT & _
                                         "\MyApplication"
```

Earlier we mentioned that we would use late-binding to the `WshShell` object to prevent an additional reference in our application. To do this, add the following variable in the declarations section of the class.

```
Dim objWshShell As Object
```

Because we've declared an object, let's use the `Initialize` event of the class to create the `WshShell` object and then the `Terminate` event to destroy it. In the `Initialize` event we'll also make sure that our root keys are available. To create a key using the `RegWrite` method, add an extra trailing backslash (\) to the end of the value name, as shown in the following code:

```
Private Sub Class_Initialize()
    Set objWshShell = CreateObject("WScript.Shell")

    ' create the root key for our company
    If (Not OptionExists(SETTINGS_COMPANY_ROOT & "\")) Then
        objWshShell.RegWrite SETTINGS_COMPANY_ROOT & "\", ""
    End If

    ' create the root key for the application
    If (Not OptionExists(SETTINGS_APP_ROOT & "\")) Then
        objWshShell.RegWrite SETTINGS_APP_ROOT & "\", ""
    End If
End Sub

Private Sub Class_Terminate()
    Set objWshShell = Nothing
End Sub
```

Okay, time to start adding the methods. Start with the `OptionExists` method. The `RegRead` method of the `WshShell` object throws a runtime error if the specified value does not exist. We'll trap this error in the `OptionExists` method.

```
Public Function OptionExists(stOptName As String) As Boolean
    ' Determines if a setting exists in the Registry
    Dim rv As Variant

    ' try to read a value
    On Error Resume Next
    rv = GetOptionValue(stOptName)
    OptionExists = (Err = 0)

    On Error GoTo 0
End Function
```

Part IV: Finalizing the Application

To write a value to the Registry in the `SetOptionValue` method, we'll use the `RegWrite` method of the `WshShell` object:

```
Public Sub SetOptionValue(stOptName As String, stOptValue As String)
    ' Sets an option value in the Registry
    objWshShell.RegWrite SETTINGS_APP_ROOT & "\" & stOptName, stOptValue
End Sub
```

Last, we'll use the `RegRead` method of the `WshShell` object to retrieve a value from the Registry.

```
Public Function GetOptionValue(stOptName As String) As Variant
    ' Gets an option value from the Registry
    GetOptionValue = objWshShell.RegRead(SETTINGS_APP_ROOT & "\" & stOptName)
End Function
```

Let's see how to test this code. Add a new standard module and add the following routine and declaration:

```
Sub TestSettingsRegistry()
    Dim r As SettingsRegistry
    Set r = New SettingsRegistry
```

When you create a new instance of the `SettingsRegistry` class, the `Initialize` event is called to create the root keys for the company and the application.

Next, let's set the default value for the root key. This is done by passing in an empty string for the option name and a value:

```
' Set the default value for the root key
r.SetOptionValue "", "MyDefaultValue"
```

Now, set a named value under the root key. This is done by passing in the option name and value. If the value does not exist, the `RegWrite` method will create it for you.

```
' Set a named value
r.SetOptionValue "MyNamedValue", "TestNamedValue"
```

Create a new section with a new value by specifying a section name followed by a backslash and then the value name.

```
' Create a section with a new value
r.SetOptionValue "TestSection\TestValue", "123"
```

Now, retrieve the values as follows:

```
' get named values
If (r.OptionExists("MyNamedValue")) Then
    Debug.Print r.GetOptionValue("MyNamedValue")
Else
    Debug.Print "Cannot find named value"
End If

If (r.OptionExists("TestSection\TestValue")) Then
```

```
    Debug.Print r.GetOptionValue("TestSection\TestValue")
Else
    Debug.Print "Cannot find named value in a different section"
End If

    Set r = Nothing
End Sub
```

When you run this code, you should have the following output:

```
TestNamedValue
123
```

Creating Options Forms

To manage options and let your users change them, you should create an options form. Previous versions of Access used a tab control to organize application options. Applications such as Access 2007 and Visual Studio still group options together but don't display them in a tab control. Or do they? We'll use a tab control in Access 2007 but hide the tabs to simulate the appearance of the Options dialog box in Visual Studio and make the presentation of options more appealing.

Creating the Form

Start by creating a new form in design view. Save the form as USysFrmOptions. Set the properties listed in the table that follows for the form.

Property Name	Property Value
Caption	Settings
Pop Up	Yes
Modal	Yes

Add the following controls with their properties. In the table that follows, the Page control is a page in the tab control.

The cboThemes combo box defined here uses a table called USysTblThemes as its row source. This table is defined later in this chapter in the section "Theming Forms."

Control Type	Control Name	Property Name	Property Value
List Box	lstCategories	Row Source	SELECT DISTINCT OptionCategory FROM USysTblOptions WHERE OptionCategory Is Not Null ORDER BY OptionCategory;

Continued on next page

Part IV: Finalizing the Application

Control Type	Control Name	Property Name	Property Value
Tab Control	tabOptions	Multi Row	No
		Style	None
Page	pgAppearance		
Page	pgReports		
Combo Box			
(place on pgAppearance)	cboThemes	Column Count	2
		Column Widths	0"
		Row Source	SELECT ThemeID, ThemeName FROM USysTblThemes ORDER BY ThemeName;
		Tag	ApplicationTheme
Check Box (place on pgAppearance)	chkShowLastForm	Tag	ShowLastFormOnStartup
Check Box (Place on pgReports)	chkMaximizeReports	Tag	MaximizeReports
Command Button (in Footer section)	cmdOk	Caption	OK
Command Button (in Footer section)	cmdCancel	Caption	Cancel

After you've created the form, you should have something that looks like the form shown in Figure 12-15. You'll notice that we've filled in some options that have the OptionCategory field set to both Appearance or Report.

Adding Events

To drive the form to set and retrieve option values, you need to add some events to the form and the controls. Start with the Load event of the form. We'll use this event to retrieve the settings from USysTblOptions. We're using the Tag property of the controls to "bind" the controls to records in the table. This is done by iterating through the controls of the form and using the Tag property as the option name parameter in the GetOptionValue method of the Settings class.

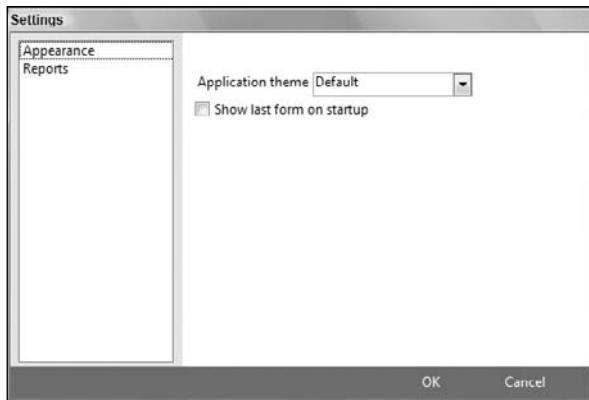


Figure 12-15

Before you add the code to the form, add the following code to the top of a new standard module:

```
Public gobjSettings As New Settings
```

Okay, now add the code to the Load event of the form:

```
Private Sub Form_Load()
    Dim c As Access.Control

    ' load the properties
    For Each c In Me.Controls
        If (Len(c.Tag) > 0) Then
            c.Value = gobjSettings.GetOptionValue(c.Tag)
        End If
    Next
End Sub
```

If the user clicks the Cancel button, the form should close and changes should not be written back. Because the form is unbound, we simply need to close the form without writing to the table.

```
Private Sub cmdCancel_Click()
    DoCmd.Close acForm, Me.Name
End Sub
```

When the user clicks OK, however, we need to write the option values back to the table and close the form. This is done by enumerating the controls and checking the Tag property, as shown earlier. This time we'll also check the value of the control to write back to the table using the SetOptionValue method of the Settings class:

```
Private Sub cmdOK_Click()
    ' save settings
    Dim c As Access.Control
    For Each c In Me.Controls
        If (Len(c.Tag) > 0) Then
```

Part IV: Finalizing the Application

```
        gobjSettings.SetOptionValue c.Tag, Nz(c.Value)
    End If
Next
' close the form
DoCmd.Close acForm, Me.Name
End Sub
```

Last, we hid the tabs in the tab control to provide a nice appearance, which leaves the user without a way to change to different pages! To fix this, add code to the `AfterUpdate` event of the `lstCategories` list box. Remember that the list of categories is sorted in the list box. By sorting the pages in the tab control to match, changing pages is simple:

```
Private Sub lstCategories_AfterUpdate()
    ' display the option category
    Me.tabOptions = Me.lstCategories.ListIndex
End Sub
```

Tab controls have a `Value` property that represents the current page selected as indicated by the `PageIndex` property of the `Page` control. Because the pages are sorted in the same manner as the items in the list box, the `ListIndex` of the item selected in the list box corresponds to a given page in the tab control.

When you run this form, any controls that have the `Tag` property set should have values set from the table. If you change a value and click OK, the values should be written back to the table.

Displaying Options in the Ribbon

Certain options might be so common that you want to show them predominantly in the application. A ribbon customization provides a nice mechanism for doing this. We are working with a limited number of options in our example, but we'll demonstrate how to tie the `USysFrmOptions` form with a ribbon customization to make sure that the Ribbon reflects the latest state of the options in the table.

Creating the Customization

Before you create the customization, create the `USysRibbons` table, as shown in Chapter 11. Add a record to this table with a ribbon named `rbnSettings` and the following XML for the customization. This customization includes two check boxes and a dialog box launcher, which is used to open the `USysFrmOptions` form to display more options:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
 onLoad="OnRibbonLoad">
<ribbon>
<tabs>
<tab id="tabSettings" label="Settings">
<group id="grpSettings" label="Settings">
<checkBox id="chkMaximizeReports"
          label="Maximize Reports"
          getPressed="OnGetOption"
          onAction="OnSetOption"
          tag="MaximizeReports"/>
<checkBox id="chkShowLastForm"
          label="Show last form on startup"
          getPressed="OnGetOption"
          onAction="OnSetOption"
```

```
        tag="ShowLastFormOnStartup" />
    <dialogBoxLauncher>
        <button id="btnSettings" onAction="OnOpenForm" />
    </dialogBoxLauncher>
</group>
</tab>
</tabs>
</ribbon>
</customUI>
```

Set the Ribbon Name property for the database to rbnSettings. You should have a Ribbon that looks something like the Ribbon displayed in Figure 12-16.



Figure 12-16

Adding the Callbacks

To make this work, you'll need some callbacks. Remember from the previous chapter that a callback is the routine that is called by the Ribbon when the user performs some action or when it needs to determine the state of a control.

Because you will refresh the Ribbon at runtime, you'll add the `onLoad` callback for the Ribbon. Create a new module called `basRibbonCallbacks` and add the following declarations:

```
Public gobjSettings As New Settings
Public gobjRibbon As IRibbonUI
```

Add the callback:

```
' onLoad callback for the Ribbon
Public Sub OnRibbonLoad(obj As IRibbonUI)
    Set gobjRibbon = obj
End Sub
```

Next, add the routine that is used in the `getPressed` callback. This callback is called when the Ribbon needs to determine whether a check box is checked. We're using the `Settings` class to determine the value in the `USysTblOptions` table.

```
' getPressed callback for checkBox control
Public Sub OnGetOption(ctl As IRibbonControl, ByRef pressed)
    ' get the option from the table
    If (Len(ctl.Tag) > 0) Then
        pressed = gobjSettings.GetOptionValue(ctl.Tag)
    End If
End Sub
```

Part IV: Finalizing the Application

Next, add the routine for the `onAction` callback of the check boxes. Again, we're using the `Settings` class to set the option values.

```
' onAction callback for checkBox control
Public Sub OnSetOption(ctl As IRibbonControl, pressed)
    ' set the option in the table
    If (Len(ctl.Tag) > 0) Then
        gobjSettings.SetOptionValue ctl.Tag, CLng(pressed)
    End If
End Sub
```

Last, add the `onAction` callback for the button in the `dialogBoxLauncher` control:

```
' onAction callback for button control
Public Sub OnOpenForm(ctl As IRibbonControl)
    If (ctl.ID = "btnSettings") Then
        DoCmd.OpenForm "USysFrmOptions"
    End If
End Sub
```

When you open the database, the check boxes in the Settings tab of the Ribbon should reflect the values stored in the options table. If you change the values, the new values should be written to the table.

Modifying the Options Form

If a user changes these options in the options form, the Ribbon should also be updated. To do this, you need to invalidate the check box controls. Add the following code to the `Click` event of the OK button on `USysFrmOptions` to invalidate the controls.

```
' invalidate the ribbon
If (Not (gobjRibbon Is Nothing)) Then
    gobjRibbon.InvalidateControl "chkMaximizeReports"
    gobjRibbon.InvalidateControl "chkShowLastForm"
End If
```

Now, when the user changes an option in the options form, the check box controls in the Ribbon should update as well.

Creating Form Themes

It doesn't necessarily make sense for all applications, but the ability to "skin" an application is becoming increasingly common. With applications such as WinAmp in the mid-90s all the way to Office 2007 itself, certain applications provide the ability to change the visual appearance of the application to suit the preference of the user. For example, Office 2007 enables you to change the appearance of the application in a new option called Color scheme. Possible values for this option are Blue, Silver, and Black. Using a table and a little code, Access applications can be themed as well.

Theming Forms

The AutoFormat Wizard in Access has always stored information about auto formats in a database that is included with Access. We've created a similar mechanism to apply themes to our own Access applications.

Creating a Table to Store Theme Data

Themes can be simple or complex, depending on the number of visual elements you want to modify. For this example, you'll store data in a table that will allow you to change the following visual elements of forms:

- Header section: back color, fore color of labels
- Footer section: back color, fore color of labels
- Detail section: back color, fore color of labels
- Labels: Caption font name and size
- Border color of text boxes, list boxes, and combo boxes

Start with a table named `USysTblThemes`.

This table is included with the download files for this book on www.wrox.com to make it easier to create. The sample file is called OptionsAndThemesTest.accdb.

Field Name	Field Type	Description
ThemeID	AutoNumber	Primary key
ThemeName	Text (50)	Name of the theme
HeaderBackColor	Long Integer	Back color for the header section of forms
HeaderForeColor	Long Integer	Fore color for label text in the form header
FooterBackColor	Long Integer	Fore color for label text in the form footer
FooterForeColor	Long Integer	Back color for the footer section of forms
DetailBackColor	Long Integer	Back color for the detail section of forms
DetailForeColor	Long Integer	Fore color for label text in the detail section
LabelCaptionFontSize	Long Integer	Font size of the label caption for labels in a form
LabelCaptionFontName	Text (50)	Name of the font used in label captions for labels in a form
ControlBorderColor	Long Integer	Border color for text boxes, list boxes, and combo boxes

You will use the sample data included with the database, which is available for download with this book for testing.

Part IV: Finalizing the Application

Applying Themes at Runtime

Once you have defined some themes, you need to write code to apply the theme at runtime. The theme of the application can be represented as an object that wraps a record in the table, so as you can probably imagine by now, you're going to use a class module to represent the theme. The class module will focus on applying themes and therefore be fairly simple, but can be expanded to manage themes in great detail.

Start by creating a new class module called `Theme`. This class will use the `Settings` class created earlier in this chapter in the section "Managing Options from Tables." This assumes you have a table created to store application options.

Add the following declarations to the `Theme` class. The first variable stores the current theme setting in `USysTblOptions` and the second will contain the recordset for the themes table:

```
Private m_currentThemeId As Long  
Private m_rsThemes As DAO.Recordset2
```

Use the `Initialize` event of the class to retrieve the current theme and then open a recordset against `USysTblThemes`:

```
Private Sub Class_Initialize()  
    Dim opt As New Settings  
    Dim stSQL As String  
  
    ' get the current theme id  
    m_currentThemeId = opt.GetOptionValue(AEP_OPT_APPTHEME)  
  
    ' open the theme recordset for the current theme  
    stSQL = "SELECT * FROM USysTblThemes WHERE ThemeID = " & m_currentThemeId  
    Set m_rsThemes = CurrentDb().OpenRecordset(stSQL)  
  
    ' cleanup  
    Set opt = Nothing  
End Sub
```

Use the `Terminate` event of the class to close the themes recordset:

```
Private Sub Class_Terminate()  
    m_rsThemes.Close  
    Set m_rsThemes = Nothing  
End Sub
```

You've exposed one property in the class called `Recordset` that returns the recordset of the current theme. This is to provide some insight into how this class can be expanded in the future. This property is not used in this example.

```
Public Property Get Recordset() As DAO.Recordset2  
    Set Recordset = m_rsThemes  
End Property
```

Chapter 12: Configuration and Extensibility

Last, create a method called `Apply` that accepts a `Form` object as a parameter. This method does the work of applying a theme using the values in `USysTblThemes` to a given form. A fair amount of work is going on in this method so let's take a closer look.

```
Public Sub Apply(frm As Form)
    ' applies the current theme to the specified form
    Dim ctl As Control
```

The first thing you'll do in this method is to apply the back color to the sections. If the value for a given back color attribute has not been set default it to white using the `Nz` function in Access:

```
With frm
    ' STEP 1 - section back colors - default to vbWhite
    .Section(acHeader).BackColor = Nz(m_rsThemes!HeaderBackColor, vbWhite)
    .Section(acFooter).BackColor = Nz(m_rsThemes!FooterBackColor, vbWhite)
    .Section(acDetail).BackColor = Nz(m_rsThemes!DetailBackColor, vbWhite)
```

Next, change the `ForeColor` property for label controls in the header section.

```
' STEP 2 - header: Label.ForeColor
For Each ctl In .Section(acHeader).Controls
    If (ctl.ControlType = acLabel) Then
        ctl.ForeColor = m_rsThemes!HeaderForeColor
    End If
Next
```

Change the `ForeColor` property for label controls in the footer section.

```
' STEP 3 - footer: Label.ForeColor
For Each ctl In .Section(acFooter).Controls
    If (ctl.ControlType = acLabel) Then
        ctl.ForeColor = m_rsThemes!FooterForeColor
    End If
Next
```

Almost done. You need to do the same to the labels in the `Detail` section. You'll also change the `BorderColor` property of text box, combo box, and list box controls in the `Detail` section.

```
' STEP 4 - detail section:
' - Label.ForeColor
' - BorderColor: TextBox, ComboBox, ListBox
For Each ctl In .Section(acDetail).Controls
    ' Label.ForeColor
    If (ctl.ControlType = acLabel) Then
        ctl.ForeColor = m_rsThemes!DetailForeColor
    ElseIf (ctl.ControlType = acTextBox Or _
            ctl.ControlType = acComboBox Or _
            ctl.ControlType = acListBox) Then
        ctl.BorderColor = Nz(m_rsThemes!ControlBorderColor, 0)
    End If
Next
```

Part IV: Finalizing the Application

Last, change the `FontName` and `FontSize` property of labels in all sections and exit the routine:

```
' STEP 5 - all sections:  
' - Label.FontName  
' - Label.FontSize  
For Each ctl In .Controls  
    If (ctl.ControlType = acLabel) Then  
        ctl.FontName = m_rsThemes!LabelCaptionFontName  
        ctl.FontSize = m_rsThemes!LabelCaptionFontSize  
    End If  
Next  
End With  
End Sub
```

To use this code, create a new form based on a table in your database. We've used the `USysTblOptions` table that was defined earlier, as shown in Figure 12-17.

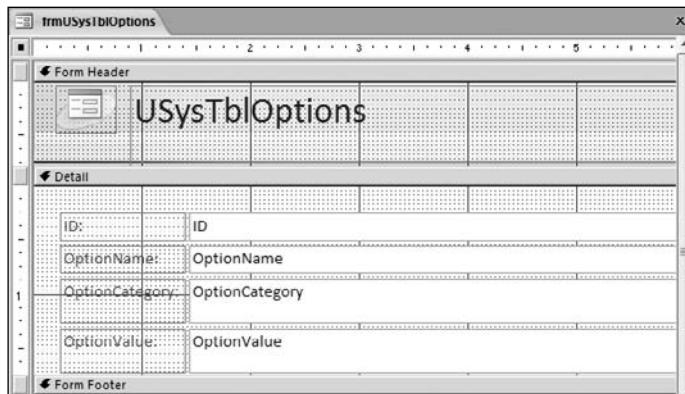


Figure 12-17

Add the following code to the `Load` event of the form. This code creates a new instance of the `Theme` class and then calls the `Apply` method to apply the theme to the given form.

```
Private Sub Form_Load()  
    Dim objTheme As New Theme  
    objTheme.Apply Me  
  
    Set objTheme = Nothing  
End Sub
```

When you open this form, you should have a form that looks something like the form in Figure 12-18. Notice that the label captions have changed to a smaller font and that the form header has a different back color than when it was in design view.

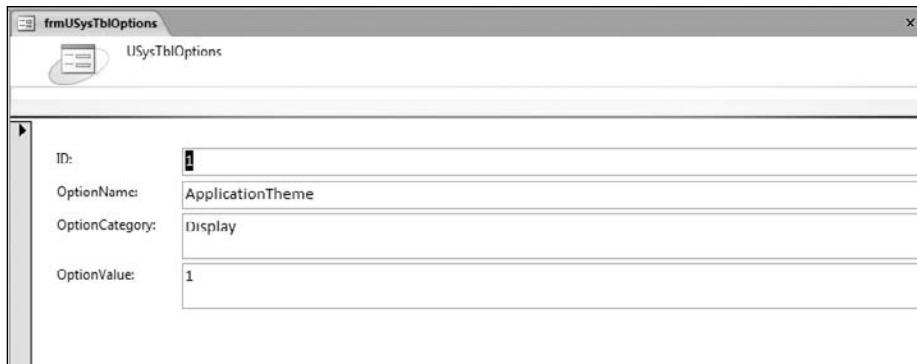


Figure 12-18

Summary

By providing customization for your users, you give them a chance to personalize the application to suit their needs or preferences. This helps to create pleasant user experiences that will keep them coming back to your application. In this chapter, you learned how to:

- Create resource DLLs using Visual Studio that can be used as the source of localized strings for Access applications
- Retrieve locale settings such as the localized name of a language or country
- Write a class module to manage application settings or options
- Integrate application settings into a ribbon customization

The spirit of this chapter continues in Chapter 13 where you will look at ways to identify the user and to create a personalized experience for the Navigation Pane in Access 2007.

13

Personalization and Security

Very few topics can stimulate discussion like security. Whether you're working with security for the first time or the hundredth time, the questions and discussions are likely to be the same — who has access, who does not have access, who can do what? When it comes to security, understanding the types of questions to ask is just as important as the security itself.

Security needs can vary from one application to the next and, in fact, some applications don't require any security at all. Sometimes security is added to an application after it has been developed. This can result in difficult changes and decisions, occasionally at the architectural level. For this reason, we recommend determining your security needs up front wherever possible. We find that designing with security in mind from the beginning makes the development process much, much simpler.

In this chapter we discuss:

- New security features available in Access 2007
- Adding personalization to your applications using Windows
- Password-protecting objects such as forms
- Restricting the number of login attempts
- Using the Windows API to lock the computer and receive notifications when the computer is locked or unlocked
- Using the Windows logon name to simulate record-level security
- Some best practices related to security for your applications

What Is Security?

This may seem like a pretty basic question, but what *is* security? The truth is that security can mean different things to different people. In most cases when we talk about security, we are talking about restricting, limiting, or preventing something from happening. We tend to think about

Part IV: Finalizing the Application

security from the perspective of some asset that we are trying to protect. For database applications, this is usually the data, intellectual property (IP) such as code or logic, or even structure such as the schema of a table. Security should not be confused with *personalization*, which we discuss in a few moments.

Security is a very complex subject. It has many different aspects such as authentication and authorization, which as a developer you should have at least a cursory understanding of. Depending on the needs of an application, users may have very high expectations with regard to the security of an application, or perhaps security is less of a concern for a given application. If you're storing contact information then you might find that a minimum amount of security is required. However, if you're storing sensitive information such as a credit card number, bank account number, or social security number, then you should be very concerned about security. Breaches of data make the news constantly these days and negative publicity is not a good thing.

Information such as a credit card number, social security number, or health information may be used to identify an individual. Such information is said to be *personally identifiable* and governments may have restrictions on how such data is stored. Privacy laws can vary greatly between countries so we strongly recommend that you are aware of the laws in the country you are deploying to before you consider storing personally identifiable information.

Before we get into the difference between security and personalization, let's talk about a couple aspects of security, namely, *authentication* and *authorization*.

Authentication

Authentication, sometimes abbreviated as AuthN, is the process by which users are identified within a system. This often requires some user database such as Active Directory in newer versions of Windows or the Security Account Manager (SAM) in earlier versions of Windows such as Windows NT.

There are several different types of authentication supported on Windows-based desktops or servers:

- Windows Authentication
- Basic Authentication
- Digest Authentication
- Kerberos
- Smartcard

In addition to these mechanisms supported by Windows, there are others such as:

- Microsoft Passport
- Single Sign On

For more information about Windows security, please refer to Microsoft Technet, which is available online at <http://technet.microsoft.com>.

The user-level security feature of the Access database engine uses the system database (MDW) file for authentication. Access 2007 does not use Windows security but we'll examine ways of using Windows authentication in your applications.

By now you are probably aware that Access 2007 does not support user-level security for the new ACCDB file format. User-level security is still supported on databases created in previous versions in the MDB file format. If you need to use user-level security for your application in Access 2007, you need to use the previous format.

Authorization

Authorization, sometimes abbreviated as AuthZ, is the process of determining the level of access that a given identity has within a system. The level of access usually applies to a particular resource or object. In other words, it answers the question, "Who can do what with a given object?"

Windows manages authorization on objects such as files or folders using access control entries (ACE) in an access control list (ACL).

The user-level security feature of the Access database engine uses both the system database and the individual database for authorization.

What Is Personalization?

Personalization is the process of tailoring an application to a particular user. For Access applications, this can mean showing a different form to a user such as a manager, or hiding a control based on membership of a group. Because personalization often involves identifying the user or group, it is sometimes confused with security.

Many people used the user-level security feature from previous versions of Access to apply personalization features to an application. For such an application, personalization is still possible but you'll need to come up with your own means of identifying users and taking action for the user. This chapter discusses how to implement personalization using the Windows user name.

Overview of Security in Access 2007

As a developer, you should be aware of a few new security features available in Access 2007. Although not all of these are developer features per se, they do affect your users. As a result, they may affect the way that you build or deploy applications.

Database Encryption

Access has had the ability to add a password to a database for quite a while; however, this password never affected the actual data on the database — that is until now. When you set the database password

Part IV: Finalizing the Application

on a database in the new ACCDB file format in Access 2007, the Access database engine actually encrypts the data stored in the database. When a user opens the file, the database engine detects that the database is encrypted and asks Access to get the password from the user. Access displays the same password prompt as it has in previous versions. The prompt is shown in Figure 13-1.



Figure 13-1

The same rules apply to passwords as they have in previous versions:

- The password is case-sensitive and can be a maximum of 20 characters long
- You must have the database opened exclusively to set the password

Access encrypts the file using the cryptography functionality included with Windows.

The MDB file format continues to use the previous mechanism for the database password for backward compatibility.

Disabled Mode

Another feature you're likely to run across in Access 2007 is Disabled mode. Disabled mode is a new feature in Access 2007 that prevents potentially malicious objects from being executed. These objects are:

- VBA code
- Action queries
- DDL queries
- SQL pass-through queries
- Certain macro actions
- ActiveX controls on forms and reports

As a developer, you may receive calls from end users saying that an application isn't working. This is likely because VBA code is disabled. In Disabled mode, your users may see a bar below the Ribbon called the message bar, which is shown in Figure 13-2.

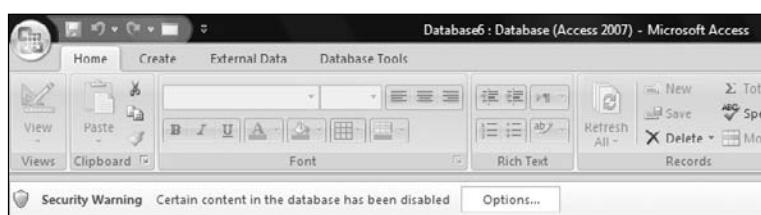


Figure 13-2

A new property on the CurrentProject object called `IsTrusted` helps notify users that something is not running as expected. When this property is `True`, code is running and you can proceed to launch the application. If this property is `False`, code is disabled and you may consider showing a different user interface to indicate that the application is disabled.

Because you cannot run code to determine if code is running, you need to use this property in a macro, as shown in Figure 13-3. Notice that when the property is `True` we open our startup form; otherwise we open a different form. Also note that we're calling this in the `autoexec` macro to make sure that our startup code runs when the database opens.

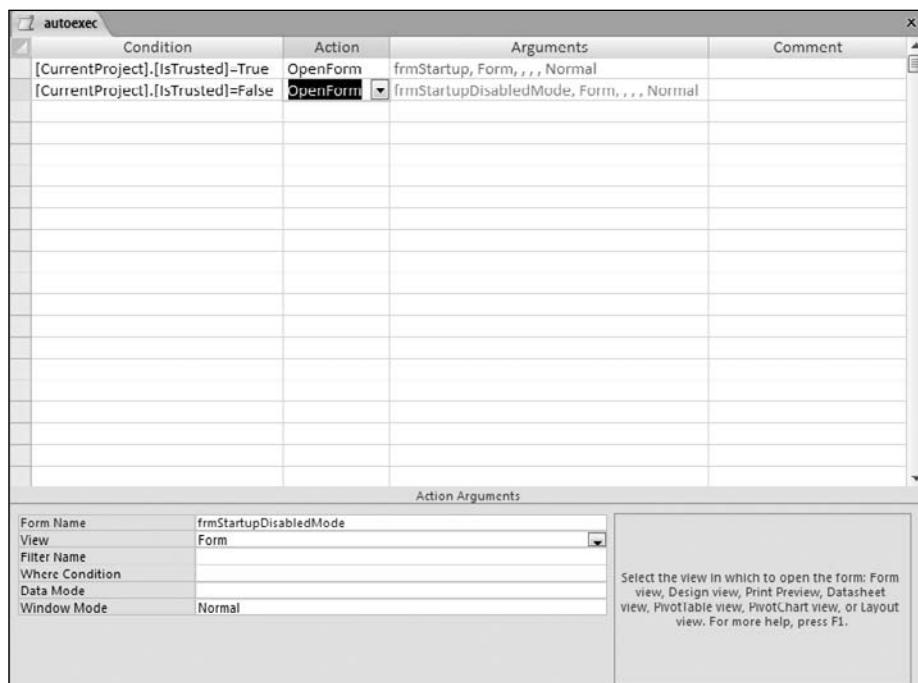


Figure 13-3

Earlier we mentioned that only certain macro actions are blocked in Disabled mode. Macros in Access 2007 run in a sandboxed environment where some actions are allowed to run in Disabled mode. The actions that are blocked include actions that can touch resources outside of the database, such as `TransferDatabase` or `TransferText`.

Office Trust Center

Another new feature for security in Office 2007 is the Office Trust Center. The Trust Center provides a single location for working with security settings for Office applications. To view the Trust Center from Access, click the Office menu and select the Access Options button. Then, select the Trust Center page in the options dialog box and click the Trust Center Settings button. This opens the Trust Center, as shown in Figure 13-4.

Part IV: Finalizing the Application

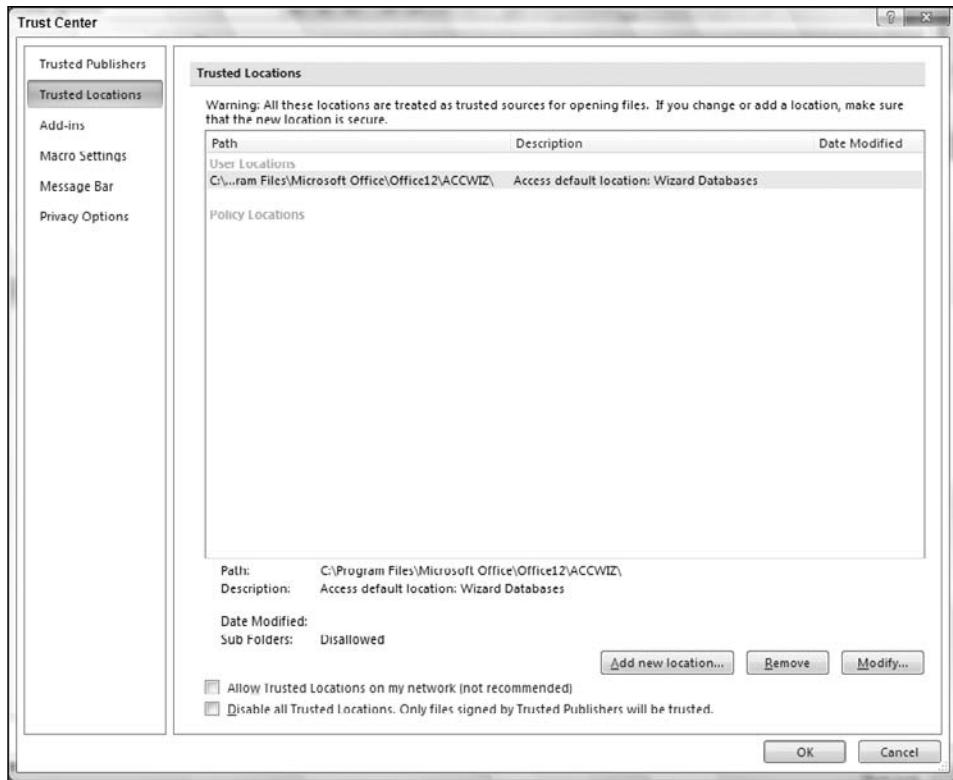


Figure 13-4

Let's take a closer look at the features in the Trust Center that are likely to affect you as a developer.

Macro Settings

If you are upgrading to Access 2007 from Access 2003, you'll probably recognize macro settings as similar to the macro security settings in Access 2003. The settings have been replaced to be a little more descriptive, but the behaviors are similar. The following table lists the Access 2007 setting and its Access 2003 equivalent.

Access 2007	Access 2003	Description
Disable all macros without notification	No corresponding setting in Access 2003	All databases open in disabled mode without notification in the message bar.
Disable all macros with notification	Medium	The message bar is displayed for all databases that are not in a trusted location. Digitally signed databases from a trusted publisher executes without notification.

Access 2007	Access 2003	Description
Disable all macros except digitally signed macros	High	Databases open in disabled mode unless they are digitally signed from a trusted publisher.
Enable all macros	Low	All databases are opened in Enabled mode without warnings. This is the least secure setting.

Access 2003 did not have an equivalent of the Disable all macros without notification setting, but the other applications in Office did. In Word and Excel, this setting was the equivalent of the Very High macro security setting. Access 2003 did not include the Very High setting because databases could not be opened in a mode where code was disabled. This is the most secure setting.

Trusted Locations

The Trusted Locations page in the Trust Center allows you to designate a given folder as trusted. When you mark a location as trusted, any database in that directory opens in Enabled mode, regardless of macro setting. Trusted locations in Access 2007 can significantly help developers who can specify the install directory of their applications within an organization. Digital signatures are no longer required if you can specify a trusted location.

Digital Signatures

Digital signatures were added to databases in Access 2003 and require a certificate to digitally sign a file. The digital signature covers VBA code, Access macros, and certain types of queries. Access 2007 supports this type of digital signature for MDB files but not ACCDB files. If you try to add a digital signature to an ACCDB file you will receive an error.

For more information about digital signatures in Access 2007, please refer to the Access 2007 VBA Programmer's Reference, ISBN 978-0470047033.

Signed Packages

Access 2007 includes a new type of file called a signed package. This feature is both a security feature and a deployment feature. Its functionality as a deployment feature lies in the fact that the package file, which ends with an ACCDC file extension, is a compressed version of an ACCDB or ACCDE file. The smaller file results in something that is easier to deploy. Its functionality as a security lies in the fact that the package is signed using a digital signature.

Creating a Signed Package

You'll need a digital certificate to create a signed package file. Digital certificates can be obtained from third-party certificate authorities such as VeriSign or Thawte, from a Certificate Services server running inside of an organization, or by creating a self-signed certificate. For the purpose of our demo, we use a self-signed certificate.

Part IV: Finalizing the Application

Creating the Self-Signed Certificate

Follow these steps to create a self-signed certificate. By default, files signed with a self-signed certificate cannot be trusted on another computer.

1. On Windows XP, click Start \Rightarrow Programs \Rightarrow Microsoft Office \Rightarrow Microsoft Office Tools, and select Digital Certificate for VBA Projects. On Windows Vista, click the Windows logo, select All Programs \Rightarrow Microsoft Office \Rightarrow Microsoft Office Tools, and then select Digital Certificate for VBA Projects.
2. Enter the name for the certificate when prompted and click OK.

Creating the Package File

Once you've created a certificate that you can use for signing, follow these steps to create a signed package file:

1. Launch Access 2007 and open an ACCDB file on your computer.
2. Click the Office menu, choose Publish, and then click Package and Sign. This should open the Select Certificate dialog box, as shown in Figure 13-5.



Figure 13-5

3. When prompted, select the path to the .ACCDC file to save the package.

You may need to re-run the Office 2007 setup if the Digital Certificate for VBA Projects tool does not appear in your Start menu.

Extracting the Signed Package

When you open a signed package, a couple of different things happen. First, the digital signature of the file is checked. Second, if the signature is valid, then you are prompted to extract the database to a location. Once the database is extracted from the package, it is not signed. Making changes to the database does not affect the package. In order to update the package, you must re-create it.

Access 2007 Navigation Pane

With some of the overview material behind us, we can get down to the business of customizing an application. We start with the navigation pane. The navigation pane in Access 2007 replaces the database window in previous versions. It provides a customizable view of objects in the database. We use this customization functionality to provide a personalized experience for users. This experience dictates how users interact with your applications.

Customizing the Navigation Pane

To start customizing the navigation pane, you need to create some custom groups and categories. Start by creating an instance of the Northwind 2007 database template. You'll notice that the navigation pane has already been customized, as shown in Figure 13-6.



Figure 13-6

In the figure, Northwind Traders is a *category*, and each item below the category is a *group*. To customize the navigation pane, we'll create our own categories and groups.

Let's say that you're developing an application for Northwind Traders. The application will be used by a few different departments within the company: Customer Service, which takes orders for the company; Purchasing, which manages inventory in the application; Sales Management, which tracks sales information for the company; and Human Resources, which is responsible for managing information about employees.

To start customizing the navigation pane, open the Navigation Options dialog box, as shown in Figure 13-7. The dialog box is available by using the right-click menu on the category name in the navigation pane.

Create categories and groups in the navigation pane as shown in the following table.

Category Name	Group Name
Northwind: Customer Service	Customer Service
Northwind: Human Resources	Human Resources
Northwind: Purchasing	Purchasing
Northwind: Sales Management	Sales Management

Part IV: Finalizing the Application

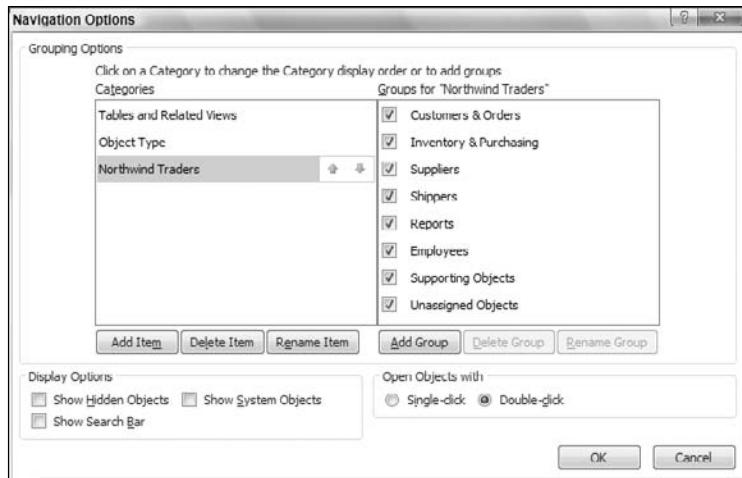


Figure 13-7

Leave the Unassigned Objects group for now and click OK to close the dialog box.

Once you've created the category and the three groups, you need to drag objects from the Unassigned Objects group to the other groups in the navigation pane. Drag the objects listed in the table that follows to the new groups. You need to switch categories to get to the appropriate group.

Object Type	Group Name	Object Name
Form	Customer Service	Customer Details
		Customer List
		Order Details
		Order List
		Product Details
		Shipper Details
		Shipper List
	Human Resources	Employee Details
		Employee List
	Purchasing	Inventory List
		Purchase Order Details

Object Type	Group Name	Object Name
Report	Sales Management	Purchase Order List
	Customer Service	Supplier Details
	Human Resources	Supplier List
	Purchasing	Sales Reports Dialog
	Sales Management	Customer Address Book
		Customer Phone Book
		Employee Address Book
		Employee Phone Book
		Supplier Address Book
		Supplier Phone Book
	Sales Management	Monthly Sales Report
		Product Category Sales by Month
		Product Sales by Category
		Product Sales by Total Revenue
		Product Sales Quantity by Employee
		Quarterly Sales Report
		Top Ten Biggest Orders
		Yearly Sales Report

Notice that we didn't add any tables or queries to the custom groups because we want to hide these objects from the users by default.

Adding objects to groups creates shortcuts to the objects. After you've added objects to the groups, we need to do a little cleanup. In each of the categories we created, right-click on the Unassigned Objects group in the navigation pane and choose Hide. Select the Northwind: Customer Service category. The navigation pane should now look something like the one shown in Figure 13-8.

Part IV: Finalizing the Application

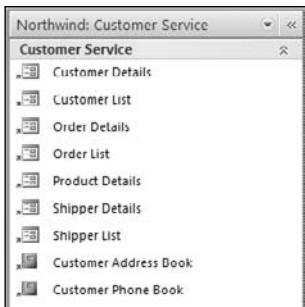


Figure 13-8

Restricting the View of the Navigation Pane

So far, we've customized the navigation pane to meet the needs of the four departments who will use our application. To complete the personalization experience we'll show the group in the navigation pane for the appropriate department. To do this, we need to add a field to the Employees table in the database to designate the department for the employee.

Modify the Employees Table

Open the Employees table in design view and add a new field called `Department`. Move the `Department` field so that it is below the `Job Title` field and save the table. Edit the data in the table to set the `Department` field, as shown in Figure 13-9. Notice that we've added two records to make sure that all departments are represented in the table.

ID	Company	Last Name	First Name	E-mail Address	Job Title	Department
1	Northwind Traders	Freehafer	Nancy	nancy@northwind	Sales Representative	Customer Service
2	Northwind Traders	Cencini	Andrew	andrew@northwir	Vice President, Sales	Sales Management
3	Northwind Traders	Kotas	Jan	jan@northwindra	Sales Representative	Customer Service
4	Northwind Traders	Sergienko	Mariya	mariya@northwin	Sales Representative	Customer Service
5	Northwind Traders	Thorpe	Steven	steven@northwin	Sales Manager	Sales Management
6	Northwind Traders	Neipper	Michael	michael@northwil	Sales Representative	Customer Service
7	Northwind Traders	Zare	Robert	robert@northwindc	Sales Representative	Customer Service
8	Northwind Traders	Giussani	Laura	laura@northwindt	Sales Coordinator	Customer Service
9	Northwind Traders	Hellung-Larser	Anne	anne@northwindt	Sales Representative	Customer Service
10	Northwind Traders	Cooper	Rob		HR Representative	Human Resources
11	Northwind Traders	Tucker	Michael		Purchasing Agent	Purchasing

Figure 13-9

Create a Login Form

The Northwind 2007 template database includes a form called `Login Dialog` that gives the user a chance to log in to the application. We could modify this form but it uses an embedded macro to specify the login information. We're going to use code for this scenario, but suffice it to say that you could use a macro if you so choose.

Start by creating a new form in design view. Add a combo box and name it `cboLogin`. Add two command buttons named `cmdLogin` and `cmdExit`. Set the properties of the form listed in the table that follows.

Property Name	Property Value
Caption	Login
Border Style	Dialog
Pop Up	Yes
Modal	Yes

After creating the form, set the Row Source of the combo box to the following SQL statement. This retrieves the employee ID, name, and department from the Employees table.

```
SELECT ID,
       [First Name] & " " & [Last Name] AS EmployeeName,
       Department
  FROM Employees
 ORDER BY [First Name] & " " & [Last Name];
```

Set the properties of the cboLogin combo box shown in the table that follows.

Property Name	Property Value
Column Count	3
Column Widths	0";1";0"

Next, you need to add some code. Start with the Click event of the cmdExit button:

```
Private Sub cmdExit_Click()
    Application.Quit
End Sub
```

Last, add code to the Click event of the cmdLogin button. Start with the event handler and one variable:

```
Private Sub cmdLogin_Click()
    ' get the department name
    Dim stDept As String
```

Make sure the combo box is not Null and concatenate the name of the department that is stored in the Column property of the combo box to our category names, which begin with the string Northwind:

```
If (Not IsNull(Me.cboLogin)) Then
    stDept = "Northwind: " & Me.cboLogin.Column(2)
```

The first thing you do is hide all categories in the navigation pane. To do this, call the SetDisplayedCategories method on the DoCmd object as shown:

```
' hide all categories
DoCmd.SetDisplayedCategories False
```

Part IV: Finalizing the Application

The `SetDisplayedCategories` method takes two arguments. The first is required and indicates whether or not to show the specified categories. The second argument is optional and indicates the category being shown. If you leave this argument blank, all categories are shown or hidden.

Now that you've hidden all the categories, you need to show the category for the department based on the user's login selection. To do that, you call the `SetDisplayedCategories` method again to show the category. This method does not update the display of the navigation pane, so we need to call the `NavigateTo` method of the `DoCmd` object. This method accepts arguments for the name of the category and group to select in the navigation pane. Our categories have only one group so we won't specify this argument, as follows:

```
' show the category for the selected department
DoCmd.SetDisplayedCategories True, stDept
DoCmd.NavigateTo stDept
```

To prevent users from changing the view of the navigation pane, call the `LockNavigationPane` method of the `DoCmd` object and specify `True` as the parameter, as follows. In addition to preventing users from changing the view of the pane, it also prevents them from opening the Navigation Options dialog box.

```
' lock the navigation pane
DoCmd.LockNavigationPane True
```

Last, close the login form, as follows. If the combo box was `Null`, display a message to the user.

```
' close
DoCmd.Close acForm, Me.Name
Else
    MsgBox "Please select your login name", vbExclamation
End If
End Sub
```

When you run the form, choose the name Andrew Cencini. The navigation pane should switch to the Northwind: Sales Management category, as shown in Figure 13-10.

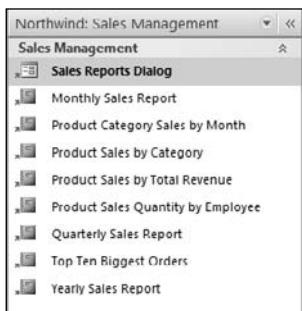


Figure 13-10

Remember, personalization is not security. The steps in this section describe how to present a majority of users with a different user experience based on some piece of information such as their user name. It will not prevent a determined user from viewing other objects in the database.

Hiding the Design Surface of Objects

Earlier we mentioned that the items you added to a custom group in the navigation pane are actually shortcuts to the object. Because user-level security is not supported for the new ACCDB file format, this begs the question about how you can prevent users from viewing your objects in design view? Well, to help out with this, object shortcuts have an option that enables you to prevent users from opening objects in design view. The caveat is that this option applies to shortcuts only. If a user can see a default view of the navigation pane, such as the Object Type category, they will be able to open objects in design view.

To set this option, right-click on a shortcut in a custom group and select View Properties. In the Properties dialog box for the object, check Disable Design View shortcuts, as shown in Figure 13-11.

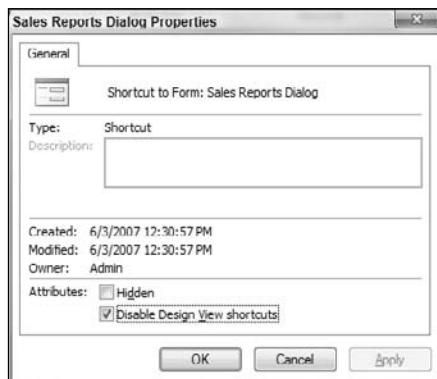


Figure 13-11

To prevent users from viewing the design of forms and reports, use an MDE or ACCDE file.

Password-Protecting Objects

One piece of functionality that we see people asking for quite a bit is the ability to prompt a user for a password to open a particular object, such as a form or report. User-level security in previous versions of Access doesn't provide this capability so we need to implement a custom solution. In order to check the password, we store a user name and password in a table and create a form to ask for this information.

Storing user names and passwords in a table is not secure, but it meets the needs of this example. If you need something more secure, we'd recommend using a client-server based database system, such as SQL Server or Oracle.

Part IV: Finalizing the Application

Creating the User Table

Let's get started by creating a table to store the user names and passwords. Create a new table with two fields: `UserName` and `UserPassword`. To further hide the password, set the `Input Mask` property of the `UserPassword` field to `Password`. Save the table with the name `USysTblUsers`. The `USys` prefix tells Access to treat this table like a system table and hide it in the navigation pane by default. Add some records to the table.

Creating the Login Form

We'll create a login form similar to the one created earlier, except this time we're adding a password. Create a new form in design view and save it with the name `USysFrmLogin`. Set the properties listed in the table that follows on the form.

Property Name	Property Value
Caption	Login
Allow Datasheet View	No
Allow PivotTable View	No
Allow PivotChart View	No
Allow Layout View	No
Auto Center	Yes
Border Style	Dialog
Pop Up	Yes
Modal	Yes

Now, add two text boxes and two command buttons to the form and set the properties listed in the table that follows on the controls.

Control	Property Name	Property Value
Text box 1	Name	<code>txtUserName</code>
Text box 2	Name	<code>txtUserPassword</code>
	Input Mask	<code>Password</code>
Command Button 1	Name	<code>cmdLogin</code>
	Caption	Login

Control	Property Name	Property Value
	Picture Caption Arrangement	General
	Picture	Use the Picture Builder to select the Lock (security) image
	Back Style	Transparent
	Cursor on Hover	Hyperlink Hand
	Default	Yes
Command Button 2	Name	cmdCancel
	Caption	Cancel
	Back Style	Transparent
	Cursor on Hover	Hyperlink Hand
	Cancel	Yes

When you view the form, you should have something that looks like the form shown in Figure 13-12.

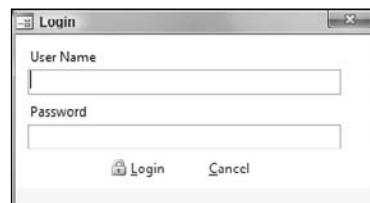


Figure 13-12

Before we start adding code to the command buttons, we have some additional requirements for the login form.

- ❑ The form should return a property to determine whether the login is valid.
- ❑ The password check should be case-sensitive.
- ❑ A user can enter a maximum of three invalid passwords.
- ❑ Users must enter both a user name and a password — no blank passwords allowed.
- ❑ Regardless of whether the user enters a valid or invalid password, we simply hide the form so that the caller can take the appropriate action based on the valid login property.

Part IV: Finalizing the Application

To meet the first requirement, we need a variable in the declarations section of the form. Add the following code to the form:

```
Private fValidLogin As Boolean
Public Property Get ValidLogin() As Boolean
    ValidLogin = fValidLogin
End Property
```

This property is available when you use a variable that is declared as the type of the form. More on that later.

Let's add some code to the command buttons, beginning with the cmdCancel button. If the user presses the Cancel button on a login form, we don't want to let them into the application so we'll exit.

```
Private Sub cmdCancel_Click()
    Application.Quit
End Sub
```

For the cmdLogin button, we need to verify the password that the user enters for the given user name compared to the password in the table. Start with the event handler and some declarations. We've defined the constant MAX_LOGINS to determine the maximum number of login attempts. This is measured using a Static variable in the routine.

```
Private Sub cmdLogin_Click()
    ' maximum number of login attempts
    Const MAX_LOGINS As Long = 3

    Dim rs             As DAO.Recordset
    Dim stSQL          As String
    Static iLoginCount As Integer
```

We want to enforce the rule that the user must enter a password so make sure that both the user name and password have been filled in.

```
' make sure both controls have been filled in
If (IsNull(Me.txtUserName)) Then
    MsgBox "Please enter the user name", vbExclamation
    Exit Sub
End If

If (IsNull(Me.txtUserPassword)) Then
    MsgBox "Please enter the password", vbExclamation
    Exit Sub
End If
```

Next, we'll increment the login count, as follows:

```
' increment the login attempts
iLoginCount = iLoginCount + 1
```

Chapter 13: Personalization and Security

When the user enters their user name, we retrieve their record from the USysTblUsers table using a recordset, as follows:

```
' get the recordset for the security table
' for the user entered in the form
stSQL = "SELECT * FROM USysTblUsers WHERE UserName = '" & Me.txtUserName & "'"
Set rs = CurrentDb().OpenRecordset(stSQL)
```

If the user entered the wrong user name, the EOF property of the Recordset object will be set to True. Check this property and alert the user if they entered an incorrect name, as follows:

```
If (rs.EOF) Then
    MsgBox "Invalid user name: " & Me.txtUserName, vbExclamation

    ' clear the user name and password
    Me.txtUserName = Null
    Me.txtUserPassword = Null
    Me.txtUserName.SetFocus
Else
```

If the user entered a valid user name, we need to validate the password. Because we've specified (in no uncertain terms) that passwords should be case-sensitive, we use the StrComp function in VBA with the vbBinaryCompare option. If this function returns zero, the strings being compared are equal.

```
' now validate the password
' passwords should be case-sensitive so do a binary comparison
If (StrComp(Me.txtUserPassword, _
            rs("UserPassword"), _
            vbBinaryCompare) = 0) Then

    ' set the valid flag
    fValidLogin = True
    ' password succeeded - hide the login form
    Me.Visible = False
Else
```

A binary comparison compares a string byte by byte. In its binary representation, the lowercase letter "a" has a value of 97, whereas an uppercase "A" is 65 and therefore not equal.

If the password is not valid, we alert the user and clear the password text box, as follows:

```
' password failed
MsgBox "Invalid password. " & vbCrLf & _
        "Please make sure CAPS lock is not on and try again.", _
        vbExclamation

        ' clear the password
Me.txtUserPassword = Null
Me.txtUserPassword.SetFocus
```

Part IV: Finalizing the Application

We also need to verify the number of login attempts. If the number of attempts is greater than or equal to the maximum we've allowed, we give the user an alert and hide the form, as follows:

```
' check the login count
If (iLoginCount >= MAX_LOGINS) Then
    MsgBox "You have exceeded the number of login attempts. " & _
        "Click OK to exit.", vbCritical

    ' cleanup and close
    rs.Close
    Set rs = Nothing
    ' hide the form
    Me.Visible = False
End If
```

And last, close out the statements and the routine.

```
End If
End If

' cleanup
If (Not rs Is Nothing) Then
    rs.Close
    Set rs = Nothing
End If
End Sub
```

We'll test this form in the next section.

Prompting for Password with Forms and Reports

To use the shiny new login form, which contains the additional functionality, we need to hook it into our forms. Say that you have an Employee form called `frmEmployees` that you'd like to ask the user for a password prior to opening. We've used the `Contacts` table template in Access 2007 to create the table for demo purposes and a form based on this table.

Add the following code to the declarations section of the `frmEmployees` form:

```
Dim m_LoginForm As Form_usysfrmLogin
```

Because we've declared a variable of the type of the Login form (`Form_usysfrmLogin`), the `ValidLogin` property is available to us.

We're relying on a Boolean value to be returned to us that we'll use to make a decision of whether to open the form. To use this property, we need to use the `Open` event of the `frmEmployees` form, which enables us to cancel it from opening. Add the following code for the event handler:

```
Private Sub Form_Open(Cancel As Integer)
```

Display the login form using the `OpenForm` method. Be sure to specify the `acDialog` option to prevent the remaining code in the event handler from running.

```
' login in dialog mode so the remaining code waits
DoCmd.OpenForm "USysFrmLogin", , , , acDialog
```

When the login form is closed (or in our case hidden), code execution returns to the caller. We get the hidden instance of the login form from the Forms collection and check the `ValidLogin` property. If this property is `False`, we alert the user that they cannot open the form and cancel the `Open` event.

```
' get the login form and check the ValidLogin property
Set m_LoginForm = Forms("USysFrmLogin")
If (Not m_LoginForm.ValidLogin) Then
    Cancel = True
    MsgBox "Invalid login attempt exceeded - you cannot open this form", _
        vbExclamation
End If
```

We didn't close the login form so we should do that now:

```
' close the login form
DoCmd.Close acForm, "USysFrmLogin"
End Sub
```

To test both forms, open the `frmEmployees` form. You should be prompted to enter login information. Be sure to enter invalid information to verify that you cannot open the `frmEmployees` form.

Creating a Password Protected Form Object

This is a convenient way to provide a password prompt for a single form, but what if you wanted to add this functionality for multiple forms? In Chapter 3, you learned about subclassing as a technique to share code among multiple forms. By moving the code we just wrote to a separate class module, we can do this easily.

There is one gotcha, however, if you remember — you can't add an event sink for the `Open` event for a form so we need to modify our code slightly.

Start by creating a new class module called `PasswordProtectedForm`. Add the following code to the declarations section of the class:

```
Dim WithEvents m_form As Access.Form
Dim m_LoginForm As Form_usysfrmLogin
```

We use the `m_form` variable to store the form instance that we're subclassing. The `m_LoginForm` variable is an instance of the login form.

To enable subclassing, we add a `Form` property to the class. Remember from Chapter 3 that you need to set the event property that you're subclassing to the string `[Event Procedure]`. We do that in the

Part IV: Finalizing the Application

Property Set routine, as shown in the following code. As you can see, we're adding an event sink for the Load event.

```
Public Property Get Form() As Access.Form
    Set Form = m_form
End Property
Public Property Set Form(oForm As Access.Form)
    Set m_form = oForm

    ' sink the Load event
    m_form.OnLoad = "[Event Procedure]"
End Property
```

Next, we want to move our code into the Load event. Because you cannot cancel the Load event, we simply close the specified form if the ValidLogin property is False.

```
Private Sub m_form_Load()
    ' login in dialog mode so the remaining code waits
    DoCmd.OpenForm "USysFrmLogin", , , , acDialog

    ' get the login form and check the ValidLogin property
    Set m_LoginForm = Forms("USysFrmLogin")
    If (Not m_LoginForm.ValidLogin) Then
        MsgBox "Invalid login attempt exceeded - you cannot open this form", _
            vbExclamation
        DoCmd.Close acForm, m_form.Name
    End If

    ' close the login form
    DoCmd.Close acForm, "USysFrmLogin"
End Sub
```

To use the new PasswordProtectedForm class, create a new form based on your Employees table called frmEmployeesPPF. Add the following code to the form:

```
Dim ppf As PasswordProtectedForm
Private Sub Form_Open(Cancel As Integer)
    Set ppf = New PasswordProtectedForm
    Set ppf.Form = Me
End Sub
```

When you open the form you should be prompted for the user name and password.

Windows Integration

As mentioned earlier, the user-level security feature in Access does not integrate with Windows. However, that doesn't mean that you can't simulate this functionality yourself using the techniques we've already discussed.

A simple solution is to use the USysTblUsers table you created earlier. Instead of storing custom user names and passwords, simply add Windows user names to the table without the password and use this table for making decisions, such as whether users can open a particular object or the database.

Determining the Logon Name

The key to simulating Windows integration is to determine the name of the user who is currently logged on to the computer. There are a few different ways to do this, but we'll use the GetUserNameEx function that was declared in Chapter 2.

```
Private Declare Function GetUserNameEx Lib "Secur32.dll" Alias "GetUserNameExA" ( _  
    ByVal NameFormat As EXTENDED_NAME_FORMAT, _  
    ByVal lpNameBuffer As String, _  
    ByRef lpnSize As Long) As Long
```

You'll remember that this function returns the name of the user in one of several different formats, which are defined in the following enumeration:

```
Public Enum EXTENDED_NAME_FORMAT  
    NameUnknown = 0  
    NameFullyQualifiedDN = 1  
    NameSamCompatible = 2  
    NameDisplay = 3  
    NameUniqueId = 6  
    NameCanonical = 7  
    NameUserPrincipal = 8  
    NameCanonicalEx = 9  
    NameServicePrincipal = 10  
    NameDnsDomain = 12  
End Enum
```

And of course, we need the helper function. We've modified it in the following to make the argument optional:

```
Function GetUserNameHelper( _  
    Optional NameFormat As EXTENDED_NAME_FORMAT = NameSamCompatible) As String  
    Dim buffer As String      ' space to receive data  
    Dim rc      As Long       ' return code  
    Dim pSize   As Long       ' pointer to a long  
                            ' this is the number of characters which is  
                            ' returned by the function  
    ' allocate some space  
    buffer = Space(255)  
    pSize = Len(buffer)  
    ' return data...  
    If (GetUserNameEx(NameFormat, buffer, pSize)) Then  
        GetUserNameHelper = Trim(Left(buffer, pSize))  
    End If  
End Function
```

Part IV: Finalizing the Application

Locking the Computer

Locking the computer is an easy way to keep prying eyes away from potentially sensitive information, whether it is in a database application, e-mail, or a document. If your users aren't already in the habit of doing this, the following one-line API function can be used to add this functionality to your applications.

```
Private Declare Function LockWorkStation Lib "user32" () As Long
```

Simply call this function on a form to lock the computer.

Receiving Notifications When the Computer Is Locked or Unlocked

Let's say that your users are now in the habit of locking their machines (or you're locking it for them using the *LockWorkstation* API), and now you want to take an action in the database when the computer is locked or unlocked. For example, when the computer is locked you might want to start a long running process such as an update to records on a server, or printing a group of reports. When the computer is unlocked, you might choose to ask users to log in to the database using the login form created earlier.

The notifications sent by Windows when the computer is locked or unlocked are called *session notifications*. As you might imagine, there is an API function you can call to receive these notifications.

When Windows sends a notification, it does so through a special routine called a *window procedure*, or *wndproc*. The window procedure is the main function running for a given window that processes messages and notifications from the operating system.

To receive notification that the computer is being locked, you need to write your own window procedure. This can also be done in Access by calling API functions. Writing your own window procedure is (also) called *subclassing*. The steps for subclassing a window are as follows:

1. Write the window procedure and listen for the message that you're interested in.
2. Tell Windows that you want to replace the existing window procedure with your own. Save a pointer to the previous window procedure so that you can use it later.
3. When you receive the specified message from the system, take the appropriate action.
4. If you receive a message other than the one you are listening for, pass it along to the previous window procedure. Windows uses messages as a means of communication for certain tasks such as painting windows so if you fail to pass other messages you end up with a mess!

When you're finished subclassing, be sure to tell Windows that you're no longer subclassing by replacing your window procedure with the previous one.

Let's take a look at how you can receive notifications when the computer is locked or unlocked. Start by creating a new module called *bassSessionNotification*. Add the following API declarations to the module:

```
' Session Notification APIs
Private Declare Function WTSRegisterSessionNotification Lib "wtsapi32.dll" _
(ByVal Hwnd As Long, _
```

Chapter 13: Personalization and Security

```
    ByVal dwFlags As Long) As Long ' call on Form_Load  
  
Private Declare Function WTSUnRegisterSessionNotification Lib "wtsapi32.dll" _  
    (ByVal Hwnd As Long) As Long ' call on Form_Unload
```

The first function tells Windows that you want to receive session notification messages. The second function tells Windows that you want to stop receiving session notification messages.

Next, we need some API functions to work with window procedures, as follows:

```
' WndProc APIs  
Private Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" _  
    (ByVal Hwnd As Long, _  
     ByVal nIndex As Long, _  
     ByVal wNewWord As Long) As Long  
Private Declare Function CallWindowProc Lib "user32" Alias "CallWindowProcA" _  
    (ByVal lpPrevWndFunc As Long, _  
     ByVal Hwnd As Long, _  
     ByVal Msg As Long, _  
     ByVal wParam As Long, _  
     ByVal lParam As Long)
```

Here, the first function sets an attribute on a window. The attribute we'll set is the pointer to a routine in our database that acts as the window procedure. The second function calls a window procedure given some parameters. We use this function for all messages except for the one we're listening for.

We also need to define some constants that represent the messages we're looking for. Also, we need to store the pointer to the previous window procedure so we can reset it later. Add the following constants and variable declaration to the module.

```
' constants  
Private Const WM_WTSSESSION_CHANGE As Long = &H2B1 ' Session change message  
Private Const WTS_SESSION_LOCK As Long = &H7 ' Session was locked  
Private Const WTS_SESSION_UNLOCK As Long = &H8 ' Session was unlocked  
Private Const GWL_WNDPROC As Long = (-4) ' Sets a new address for the  
    wndproc  
' pointer to the previous wndproc  
Private lpPrevWndProc As Long
```

With the declarations out of the way, it's time to subclass a window. Add the following code to the module. This code calls the `WTSRegisterSessionNotification` function to tell Windows that we want to receive session notifications, and then calls the `SetWindowLong` API function to subclass the specified window.

```
Sub SetWndProcOfHwnd(Hwnd As Long)  
    ' receive session notifications  
    WTSRegisterSessionNotification Hwnd, 0  
  
    ' subclass HWnd by specifying our WndProc  
    lpPrevWndProc = SetWindowLong(Hwnd, GWL_WNDPROC, AddressOf MyWndProc)  
End Sub
```

Part IV: Finalizing the Application

The `AddressOf` operator was added in VBA6. This operator returns the address of a function in a standard module in VBA. If you are familiar with C++, you use this operator when you need to pass a function pointer as an argument to an API function.

To reset the window procedure when we're done, add the following code. Notice that we're also calling the `WTSUnRegisterSessionNotification` API to stop receiving session notification messages.

```
Sub RestoreWndProc(Hwnd As Long)
    Dim rc As Long

    ' stop receiving session notifications
    WTSUnRegisterSessionNotification Hwnd

    ' restore the wndproc
    rc = SetWindowLong(Hwnd, GWL_WNDPROC, lpPrevWndProc)
    lpPrevWndProc = 0
End Sub
```

Next, we need to define the window procedure named `MyWndProc` that we supplied to the `AddressOf` operator in `SetWndProcOfHwnd`. You may recognize the signature of this routine as similar to the `SendMessage` API function defined in Chapter 2. Functions that work with messages typically use arguments such as these for passing data:

```
Public Sub MyWndProc(ByVal Hwnd As Long, _
    ByVal Msg As Long, _
    ByVal wParam As Long, _
    ByVal lParam As Long)

    Dim rc As Long

    ' listen for the WM_WTSSESSION_CHANGE message
    If (Msg = WM_WTSSESSION_CHANGE) Then
        If (wParam = WTS_SESSION_LOCK) Then
            LockDatabase
        ElseIf (wParam = WTS_SESSION_UNLOCK) Then
            UnlockDatabase
        End If
    Else
        ' call the base wndproc
        rc = CallWindowProc(ByVal lpPrevWndProc, _
            ByVal Hwnd, _
            ByVal Msg, _
            ByVal wParam, _
            ByVal lParam)
    End If
End Sub
```

When the computer is locked, we'll call a routine called `LockDatabase`. This routine is defined as follows. For demo purposes, we're simply writing to the Immediate window.

```
Public Function LockDatabase() As Long
    ' do something when the session is locked...
```

```
    Debug.Print "Session locked!", Now()
End Function
```

When the computer is unlocked, we'll call a routine called `UnlockDatabase`. This routine opens the `USysFrmLogin` form defined earlier in this chapter and is defined as follows:

```
Public Function UnlockDatabase() As Long
    ' show the logon form when the database is unlocked
    DoCmd.OpenForm "USysFrmLogin", , , , acDialog
End Function
```

Almost done. We need to call these functions. To make sure that the routine is called when the database opens, we'll create a form. The form opens and is hidden from an `autoexec` macro. Create a new form called `USysFrmSessionNotification`. Add code to the `Load` and `Unload` events of the form to set and restore the window procedure, as follows:

```
Private Sub Form_Load()
    ' subclass the form
    basSessionNotification.SetWndProcOfHwnd Me.Hwnd
End Sub
Private Sub Form_Unload(Cancel As Integer)
    ' reset the window procedure
    basSessionNotification.RestoreWndProc Me.Hwnd
End Sub
```

Last, create a new macro named `autoexec` that runs when the database opens. Add the `OpenForm` action to the macro and set the `Form Name` argument to `USysFrmSessionNotification`. Set the `Window Mode` argument of the macro to `Hidden`.

To test this, close and re-open the database. Next, lock and unlock the computer. When the computer is unlocked, the login form created earlier should be displayed.

Simulating Record Level Security

Let's say that you are developing a customer relationship management database and are tracking customer information. For the sales employees using the application, these customers are leads in the application so they are vital to their survival. We'd like to impose some form of security at the record level for this information.

Client-server databases such as SQL Server often include low level security on tables. Although you can't add this functionality on an Access table directly, you can simulate it using code. For each table, you need to add a field to indicate who the owner is for a given record. Using the `Customer` table in the Northwind 2007 template database, add a new text field called `RecordOwner`. Save the table and fill it with values including your own Windows login name.

Limiting the View to a Particular User

To limit the view to the user who is currently logged on the computer, you simply change the record source of the form at runtime. Create a new form based on the `Customers` table in Northwind 2007 and

Part IV: Finalizing the Application

add the following code to the Load event of the form. You need the GetUserNameHelper function defined earlier in the chapter.

```
Private Sub Form_Load()
    ' filter by user
    Me.RecordSource = "SELECT * FROM Customers WHERE RecordOwner = '" & _
        GetUserNameHelper() & "'"
End Sub
```

When the form opens, the records should be filtered by the user who is currently logged on to the computer, as illustrated in Figure 13-13.



The screenshot shows a Microsoft Access form titled "frmCustomers" with a subCaption "Customers". The form contains a table with the following data:

ID	Company	Last Name	First Name	Job Title	Address	City	State/Province	RecordOwner
1	Company A	Bedecks	Anna	Owner	123 1st Street	Seattle	WA	Rob
3	Company C	Axen	Thomas	Purchasing Representative	123 3rd Street	Los Angeles	CA	Rob
4	Company D	Lee	Christina	Purchasing Manager	123 4th Street	New York	NY	Rob
8	Company H	Andersen	Elizabeth	Purchasing Representative	123 8th Street	Portland	OR	Rob
16	Company P	Goldschmidt	Daniel	Purchasing Representative	456 16th Street	San Francisco	CA	Rob
17	Company Q	Bagel	Jean Philippe	Owner	456 17th Street	Seattle	WA	Rob
19	Company S	Eggerer	Alexander	Accounting Assistant	789 19th Street	Los Angeles	CA	Rob
20	Company T	Li	George	Purchasing Manager	789 20th Street	New York	NY	Rob
23	Company W	Entin	Michael	Purchasing Manager	789 23th Street	Portland	OR	Rob

Figure 13-13

Consider using the GetUserNameHelper function as the Default Value of the RecordOwner field on the form to ensure that new records set the field correctly.

Locking Records Based on Logon

Limiting the view is one possible technique for restricting data for users, but what if you want to let users see all data, but only make updates to their records? To do this, we lock records on the form using the Current event, as shown in the following code:

```
Private Sub Form_Current()
    Dim stUser As String

    ' get the user name
    stUser = GetUserNameHelper()

    ' lock records based on user name
    With Me
        .AllowDeletions = (.RecordOwner = stUser) Or IsNull(Me.RecordOwner)
```

```
.AllowEdits = (.RecordOwner = stUser) Or IsNull(.RecordOwner)
End With
End Sub
```

In this scenario, if the RecordOwner field has not been set, or if the RecordOwner is the user who is currently logged on to the computer, we allow deletions and edits. We're always allowing the user to add records in this case.

As you can see, once you have the user name, you can personalize the user experience in a number of ways.

Database Encryption with DAO and ADO

The Data Access Objects (DAO) API in the Access database engine (formerly Jet) has supported encoding databases for a number of years. It has been updated for Access 2007 to enable database encryption. Let's look at some scenarios where you can use both DAO methods and ActiveX Data Objects (ADO) methods to encrypt databases or work with database passwords.

Creating an Encrypted Database with DAO

The first thing we do is create an encrypted database using the CreateDatabase method in DAO. By default, this method creates a database in the ACCDB file format for Access 2007. This is done using the following code:

```
' creates an encrypted database using DAO
Public Sub CreateEncryptedDbDAO(stPath As String, stPassword As String)
    Dim stLocale As String ' locale string
    Dim db      As DAO.Database

    ' the locale string includes the password
    stLocale = dbLangGeneral & ";PWD=" & stPassword

    ' create the database
    Set db = DBEngine.CreateDatabase(stPath, stLocale)
    db.Close

    Set db = Nothing
End Sub
```

The database password is specified as a part of the locale string for a database in DAO, which includes the language. In our case, we're using the dbLangGeneral constant and appending the password, which is specified using the PWD text in the string.

Changing the Database Password with DAO

If you have an existing database, you might want to change the password for the database. This is done with the NewPassword method of the Database object in DAO as demonstrated with the following code:

```
' uses DAO to change a database password
Public Sub ChangePasswordDAO(stDatabase As String, _
```

Part IV: Finalizing the Application

```
stOldPassword As String, _
stNewPassword As String)

On Error GoTo ChangePasswordErrors

' error constants
Const ERR_INVALID_PASSWORD As Long = 3031
Const ERR_DATABASE_OPEN      As Long = 3704

Dim stLocale As String  ' locale string
Dim db        As DAO.Database

' the locale string includes the password
stLocale = dbLangGeneral & ";PWD=" & stOldPassword

' open the database exclusively
Set db = DBEngine.OpenDatabase(stDatabase, True, False, stLocale)

' assign a password
db.NewPassword stOldPassword, stNewPassword

Cleanup:
If (Not (db Is Nothing)) Then
    db.Close
End If
Set db = Nothing
Exit Sub

ChangePasswordErrors:
Select Case Err
    Case ERR_INVALID_PASSWORD:
        MsgBox "Invalid password specified", vbExclamation
        Resume Cleanup
    Case ERR_DATABASE_OPEN:
        MsgBox "The database is already open", vbExclamation
        Resume Cleanup
    Case Else
        Debug.Assert Err = 0
        Resume
End Select
End Sub
```

This routine includes arguments for the database as well as the old and new passwords. As a result, you can use this procedure in your applications with minimal modification.

Adding a Database Password to a Database with DAO

The NewPassword method shown in the previous example can be used to set a password on a database that doesn't already have one. However, what if you wanted to maintain a copy of the database with a password and one without? It turns out that you can compact a database to assign a password, as shown in the following code:

```
' compacts a database to add a password using DAO
Public Sub EncryptDbWithCompactDAO(stOldDb As String, _
    stNewDb As String, _
    stPassword As String)
    Dim stLocale As String      ' locale string

    ' the locale string includes the password
    stLocale = dbLangGeneral & ";PWD=" & stPassword

    ' compact the old database database to encrypt it
    DBEngine.CompactDatabase stOldDb, _
        stNewDb, stLocale, dbVersion120, dbLangGeneral
End Sub
```

Changing the Database Password with ADO

Last, we change the password for a database using ADO. This is actually done using a SQL statement such as:

```
ALTER DATABASE PASSWORD [NewPassword] [OldPassword]
```

If either of the password arguments is not specified, you must use the string `NULL` in its place. The following code uses a `Connection` object in ADO to change the database password. Notice that if the user passes in an empty string for either of the password arguments, we replace it with the string `NULL`. The following code requires a reference to the ActiveX Data Objects Library.

```
' uses ADO to change a database password
Public Sub ChangePasswordADO(stDatabase As String, _
    stOldPassword As String, _
    stNewPassword As String)

    Dim cn      As ADODB.Connection
    Dim stSQL   As String      ' SQL statement

    ' set connection properties
    Set cn = New ADODB.Connection
    cn.Provider = "Microsoft.ACE.OLEDB.12.0"      ' OLE DB Provider
    cn.Properties("Data Source") = stDatabase          ' data source
    cn.Mode = adModeShareExclusive                  ' open exclusive

    ' add the existing database password if it already has one
    If (Len(stOldPassword) > 0) Then
        cn.Properties("Jet OLEDB:Database Password") = stOldPassword
    End If

    ' open the connection
    cn.Open
    ' change the password
    stSQL = "ALTER DATABASE PASSWORD "

    ' if either of the passwords are an empty string
    ' use the word "NULL"
```

Part IV: Finalizing the Application

```
If (Len(stOldPassword) = 0) Then stOldPassword = "NULL"
If (Len(stNewPassword) = 0) Then stNewPassword = "NULL"

' append the passwords and run the statement
stSQL = stSQL & stNewPassword & " " & stOldPassword
cn.Execute stSQL

cn.Close
Set cn = Nothing
End Sub
```

Any time you work with database passwords, the database must be opened exclusively. You do this in DAO by specifying an argument in the `CompactDatabase` or `OpenDatabase` methods. In ADO, this is done by setting the `Mode` property of the `Connection` object to `adModeShareExclusive`.

Best Practices

When designing an application, you consider many things; how you will protect your assets is one of the most important. The following are some things to consider when designing a database application.

Using a Client-Server Database to Protect Data

If you need to store sensitive information in your database, you should seriously consider using a client-server based database system such as SQL Server, Oracle, or Sybase. Because Access is a file-based database, it may become more difficult to secure than one of the client-server based systems. SQL Server 2005 Express Edition is available for free download on the Microsoft Web site and provides the same level of security as SQL Server 2005.

Access enables you to easily connect to remote data sources on client-server systems using ODBC drivers, thus making it a great front-end development tool for remote data.

AllowBypassKey Property

Holding down the Shift key is a nice way to get into your database for development without running startup code. However, you may not be so keen on your users finding out about this trick! So, in order to prevent users from using the Shift key, a.k.a. the *bypass* key, set the `AllowBypassKey` property of the database to `false`. The following code shows you how:

```
Public Sub SetBypassKey(fSet As Boolean)
On Error GoTo SetBypassKeyErrors

' constants
Const CON_PROPERTY_NAME      As String = "AllowBypassKey"
Const CON_ERR_PROPNOTFOUND As Long = 3270

' get the database
Dim db As DAO.Database
Set db = CurrentDb()
```

```
' try to set the property
db.Properties(CON_PROPERTY_NAME) = fSet

ExitRoutine:
    db.Close
    Set db = Nothing
    Exit Sub

SetBypassKeyErrors:
    If (Err = CON_ERR_PROPNOTFOUND) Then
        ' property doesn't exist - create it
        db.Properties.Append _
            db.CreateProperty(CON_PROPERTY_NAME, dbBoolean, fSet)

        ' exit
        Resume ExitRoutine
    Else
        Debug.Assert False
        Resume
    End If
End Sub
```

This code sets the property to the specified value and creates the property if it doesn't exist. Call this routine from any startup or shutdown code in your application to make sure the property is set.

Setting this property only keeps casual users from using the Shift key when opening your database. A determined user can reset the property and open the database using the Shift key.

Using an ACCDE or MDE File

Although it is not technically a security feature, using an ACCDE or MDE file is the best way to protect the intellectual property contained in an application.

Using an ACCDR File

Access 2007 includes a new file extension called .ACCDR. This extension launches Access with the /run-time switch, preventing users from seeing the navigation pane or the built-in Ribbon. Again, this is also not technically a security feature, but it will keep honest users honest and provide a runtime experience for your applications. Simply rename your accdb file to use the accdr file extension and open it from Windows Explorer.

File Format Selection

Whether you're starting a new application or upgrading an existing application, the choice of file format for your database is pretty important. In Access 2007, your choice of file format may result in a tradeoff between the availability for a new feature versus the removal of existing features.

Part IV: Finalizing the Application

The following table lists the security-related features available in Access over the years and its availability by file format in Access 2007.

Feature	ACCDB	MDB	New in Access 2007
Access database engine sandbox mode	Yes	Yes	No
Database encryption	Yes	No	Yes
Database password	Yes	Yes	No (Adds encryption in Access 2007)
Digital signatures	No	Yes	No
Disabled mode	Yes	Yes	Yes
Encoding and decoding	No	Yes	No
Office Trust Center	Yes	Yes	Yes
Package and Sign	Yes	No	Yes
User-level security	No	Yes	No

Summary

When you're developing an application, determining the correct amount of security is an important step in the overall process. You may even find that what you really need for the application is personalization as opposed to true security.

Some of the key points in this chapter are, as follows:

- ❑ Creating your own table of users enables you to create your own personalization and security mechanisms.
- ❑ Adding a little code enables forms to be customized to prompt for passwords.
- ❑ Assisting with security, Access 2007 offers new features that help developers and end users alike.
- ❑ Using the Windows API enables you do to some pretty cool stuff to extend your applications and integrate with the operating system.

The next chapter discusses how to distribute applications to end users and discusses using standard processes for rolling out applications.

14

Deployment

You've written your application, tested it, and it's time to get it out to your users. If the application you're deploying is within an organization, this might be as simple as copying it to a network share. Off-the-shelf applications, however, may have additional requirements. For example, do you take steps to try and prevent piracy? Do you require a registration process or provide a limited-use application?

Regardless of how your application is distributed, you may still have additional runtime requirements such as an external database server. Issues such as these are problems to solve as part of deployment because that's when problems may occur.

This chapter covers the following:

- Creating a build process for Access-based applications
- Ensuring that application dependencies are taken care of
- Updating an application
- Licensing and registration

Creating an Automated Build

When we talk about creating a build, we typically think of languages such as Visual Basic .NET, C#, or C++ that produce compiled code. In Access 2007, the equivalent of this, of course, is the ACCDE file. In addition to creating compiled code, development environments that use these languages do some other things as well, such as stamping a binary with the build number.

For an Access application, you might choose to stamp it with a build number to indicate the build of the database being distributed. The build number can also be used to help track when it was created. In addition to a build number, you may have other processes that can be automated such as removing test data.

Part IV: Finalizing the Application

Because this is an automated process, this is a good opportunity to create an external tool. We'll use Visual C# to create the tool in this section. A Visual Basic .NET version of the tool is included with the sample files for download with this book at www.wrox.com.

Creating the Visual Studio Project

To provide a user interface that allows you to select objects, you need to create a Windows Forms application. Start by launching Visual Studio and create a new C# Windows application. Name the application BuildAccApp.

We're not crazy about the default name of the form: Form1. Select form in the Solution Explorer and then select the **File Name** property for the form. Change the **File Name** of the form to MainForm.cs.

We are going to use code in some additional namespaces as well. Add the following lines of code to the top of the **MainForm.cs** source file:

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

This will prevent us from having to fully qualify the namespace — ultimately saving us some typing.

Adding References

In order to create an ACCDE or MDE file, we'll need a reference to the Access object model. Because we're working with tables and database properties, we also need a reference to the Access database engine object model (DAO).

Start with the Access reference. Click the Project menu and then click Add Reference. Choose the COM tab and select the Microsoft Access 12.0 Object Library, as shown in Figure 14-1.

The Access object model includes a dependency on the DAO 3.6 object model because of the **CurrentDb** function on the **Access Application** object. As a result, when you add a reference to the Access 12.0 object model, you get some additional references including the DAO reference. However, this version of DAO, which is used with Access 2000, 2002, and 2003, cannot open the newer ACCDB file format included with Access 2007. Therefore, there are a couple steps you need to take to add the correct DAO reference.

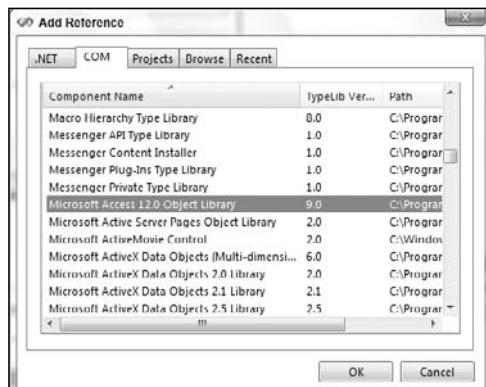


Figure 14-1

First, select the current DAO reference from the References folder in the Solution Explorer and delete it. Next, click the Project menu and then click Add Reference. Choose the COM tab and select the Microsoft Office 12.0 Access Database Engine Object Library. After updating the reference, the references in the Solution Explorer should look like those shown in Figure 14-2.

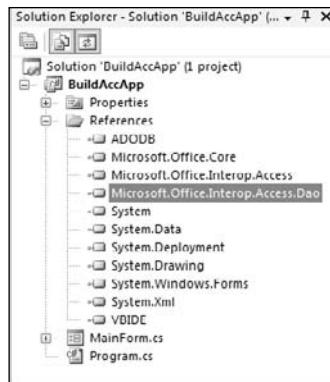


Figure 14-2

To make these object models easier to work with, add the following statements to the top of the `MainForm.cs` source file.

```
using OFF = Microsoft.Office.Core;
using ACC = Microsoft.Office.Interop.Access;
using DAO = Microsoft.Office.Interop.Access.Dao;
```

These will create shortcuts that you can use in code to avoid typing the full name of the namespace.

Private Data

We know that we're going to need a couple of instances of the `Database` object in DAO. When creating an ACCDE file, we also need an instance of the `Access Application` object. In addition, because we are automating the Access database engine from outside of Access, we'll need an instance of the `DBEngine` object. Add the following code to the form. This code should appear beneath the class definition as follows:

```
public partial class MainForm : Form
{
    // object data
    private DAO.Database sourceDatabase;
    private DAO.Database targetDatabase;
    private DAO.DBEngineClass myEngine;
    private ACC.ApplicationClass myApp;
```

We need some additional data in the class as well. Add the following code:

```
// scalar data
string errorText = null;
int major = 1;           // default to 1.0
int minor = 0;
int revision = 0;
```

Part IV: Finalizing the Application

Build properties in the source and the destination database will be stored in database properties. Add the following constants to define these properties:

```
// Source properties constants
private const string SRC_PRP_TARGET      = "BuildTarget";
private const string SRC_PRP_STARTDATE   = "AppStartDate";
private const string SRC_PRP_APPMAJOR    = "AppVersionMajor";
private const string SRC_PRP_APPMINOR    = "AppVersionMinor";
private const string SRC_PRP_APPREVISION = "AppVersionRevision";
// Destination properties constants
private const string DST_PRP_RELDATE    = "AppReleaseDate";
private const string DST_PRP_VERSION     = "AppVersion";
private const string DST_PRP_DEBUG       = "DEBUG";
```

There are a couple of exceptions that may be thrown from DAO so add constants for their COM error codes:

```
// error constants
private const int ERR_PROP_NOTFOUND      = -2146825018;
private const int ERR_PROP_ISNULL        = -2146824903;
```

Last, we need to create an instance of the DBEngine object. Modify the `MainForm` method of the form to instantiate an instance of the `DBEngineClass`, as shown here:

```
public MainForm()
{
    InitializeComponent();
    // create an instance of the Access database engine
    myEngine = new DAO.DBEngineClass();
}
```

The `MainForm` method is also known as the constructor. The constructor for a class runs when the object is instantiated. The `Initialize` event of a class module is the equivalent in VBA.

Helper Functions

We're tracking error messages through the variable called `errorText`. To provide consistent formatting of errors, add the following functions, called `ShowErrorMessage`, to the code:

```
private void ShowErrorMessage()
{
    ShowErrorMessage(errorText);
}
private void ShowErrorMessage(string errorMessage)
{
    MessageBox.Show(errorMessage,
                    "Build Failed",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Design the Form

With the references correctly set, we can start designing the form. To get an idea of where this is going, let's define the requirements for the application:

- Must be able to select the database to be built
- Must be able to specify the target database name
- Need an option to create an MDE or ACCDE file
- Should be able to stamp the build number and release date on the database
- Should be able to delete data from tables in the event that you have test data in the database

Adding Controls

In total, there are 19 controls we need to add to the form so let's get started.

Database Paths

To ask for the path to the database and the destination database we need a couple of text boxes. Add two labels and two text boxes to the form. Set their properties as shown in the table that follows.

Control	Property Name	Property Value
label1	Location	3, 9
	Text	Select database
textBox1	(Name)	inputDatabase
	Location	6, 25
	Size	215, 21
label2	Location	3, 49
	Text	Destination
textBox2	(Name)	outputDatabase
	Location	6, 65
	Size	215, 21

Build Properties

The build properties we'll set are the application start date, release date, and build number. The application start date is used to calculate the build number but is not required. Add a group box, three labels, two date time pickers, and one text box to the form and set their properties as shown in the table that follows.

Part IV: Finalizing the Application

Control	Property Name	Property Value
groupBox1	(Name)	propertiesGroup
	Location	6, 92
	Size	296, 105
	Text	Build Properties
label3	Location	6, 17
	Text	Application start date
label4	Location	6, 44
	Text	Release date
label5	Location	6, 71
	Text	Application version
dateTimePicker1	(Name)	appStartDate
	Format	Short
	Location	122, 20
	Size	168, 21
dateTimePicker2	(Name)	releaseDate
	Format	Short
	Location	122, 44
	Size	168, 21
textBox3	(Name)	appVersion
	Location	122, 71
	Size	168, 21

List of Tables

We're using a checked list box to display the list of tables to delete data from. Add one label and one checked list box to the form and set their properties as shown in the table that follows.

Control	Property Name	Property Value
label6	Location	3, 200
	Text	Delete data from tables
checkedListBox1	(Name)	deleteDataListbox
	CheckOnClick	True
	Location	6, 216
	Size	296, 100

Additional Options

Add two check box controls to the form and set its properties as shown in the table that follows.

Control	Property Name	Property Value
checkBox1	(Name)	createACCDE
	Location	6, 322
	Text	Create ACCDE/MDE
checkBox2	(Name)	debugBuild
	Location	6, 346
	Text	Create debug build

Buttons

Last, we'll add four buttons to the form. Two buttons will be used to browse for the source and destination files. Another will be used to create the actual build and the other to exit the application. Set their properties as shown in the table that follows.

Control	Property Name	Property Value
button1	(Name)	buildButton
	FlatStyle	System
	Location	146, 340
	Text	&Build

Continued on next page

Part IV: Finalizing the Application

Control	Property Name	Property Value
button2	(Name)	exitButton
	FlatStyle	System
	Location	227, 340
	Text	E&xit
button3	(Name)	browseSourceButton
	FlatStyle	System
	Location	227, 23
	Text	Br&owse...
button4	(Name)	browseTargetButton
	FlatStyle	System
	Location	227, 63
	Text	B&rowse...

Setting Form Properties

Once you've added all of the controls, you need to go back and set properties of the form, as shown in the table that follows

Property Name	Property Value
(Name)	MainForm
AcceptButton	buildButton
CancelButton	exitButton
Font	Tahoma
FormBorderStyle	FixedDialog
MaximizeBox	False
MinimizeBox	False
Size	320, 397

Property Name	Property Value
StartPosition	CenterScreen
Text	Build Access Application

After you've set all the properties, you should have a form that looks something like the one shown in the Visual Studio form designer in Figure 14-3.

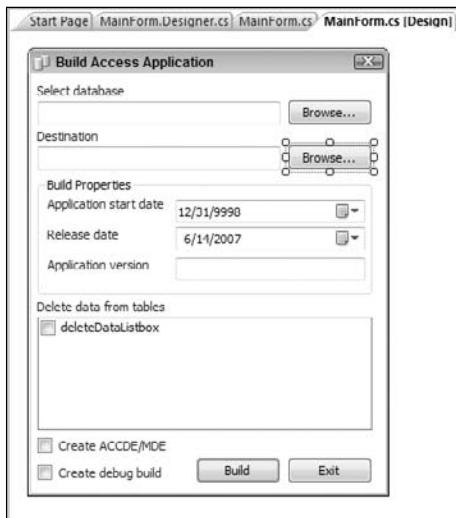


Figure 14-3

We didn't appear to use a naming convention for controls on the form but we're actually following the guidelines for .NET development. These guidelines differ greatly from the Access conventions of Hungarian notation, Reddick VBA (RVBA), or Leszynski Naming Convention (LNC). You can read more about the .NET guidelines from MSDN. More information can be found about these Access conventions in the Access 2007 VBA Programmer's Reference.

Control Events

To add functionality to the application, let's start adding code to the controls, beginning with the DateTimePicker controls.

Updating the Version Number

When the dates change in the form, we want to update the version number. The algorithm for this calculation will be described later in the section "Calculating the Version Number." For now, double-click on the appStartDate control to add an event handler for the ValueChanged event. Add the following code for the event:

```
private void appStartDate_ValueChanged(object sender, EventArgs e)
{
    // update the version
```

Part IV: Finalizing the Application

```
        appVersion.Text = CalculateVersion();
    }
```

Next, do the same thing for the `releaseDate` control. Add the following code to the event handler for the `ValueChanged` event:

```
private void releaseDate_ValueChanged(object sender, EventArgs e)
{
    appVersion.Text = CalculateVersion();
}
```

Browsing for Files

To make finding and saving files easier, we'll use the `Click` event of the `Browse` buttons to display a file dialog box. To add the code, double-click on the `browseSourceButton` control to add the event handler for the `Click` event. Add the following code to the event:

```
private void browseSourceButton_Click(object sender, EventArgs e)
{
    // prompt for a file name
    using (OpenFileDialog fd = new OpenFileDialog())
    {
        fd.CheckFileExists = true;
        fd.CheckPathExists = true;
        fd.Multiselect = false;
        fd.ShowHelp = false;
        fd.Filter = "Microsoft Access Databases (*.accdb;*.mdb) " +
            "| *.accdb;*.mdb";
        fd.Title = "Select Application to Build";
        if (DialogResult.OK == fd.ShowDialog())
        {
            inputDatabase.Text = fd.FileName;
            // get the information from the database
            GetDatabaseInfo(fd.FileName);
        }
    }
}
```

When you double-click in the control, you should see the standard Open dialog box. Next, repeat the process for the `Click` event of the `browseTargetButton` control. Add the following code to this event:

```
private void browseTargetButton_Click(object sender, EventArgs e)
{
    // prompt for the destination database
    using (SaveFileDialog fd = new SaveFileDialog())
    {
        fd.CheckFileExists = false;
        fd.CheckPathExists = true;
        fd.ShowHelp = false;
        fd.Filter = "Microsoft Access 2007 Database (*.accdb)|*.accdb";
        fd.Filter += "|Microsoft Access 2000 Database (*.mdb)|*.mdb";
        fd.Filter += "|ACCDE File (*.accde)|*.accde";
        fd.Filter += "|MDE File (*.mde)|*.mde";
        if (DialogResult.OK == fd.ShowDialog())
```

```
        {
            outputDatabase.Text = fd.FileName;
        }
    }
}
```

We've added more filters in this dialog box because we need the option to save in different formats.

The using block is different from the using statement in C#. The using block shown in the previous code is used to enforce garbage collection. Behind the scenes, this block of code generates a call to the Dispose method for the object in question.

Exit Button

As you would imagine, the Exit button enables you to exit the application. Add the following code to the Click event of the button:

```
private void exitButton_Click(object sender, EventArgs e)
{
    // exit
    Application.Exit();
}
```

Build Button

The Build button does the majority of the work to create the new database and set the properties. We'll use programming by intention here and define the code before it is implemented. Add the following code to Click event of the buildButton control:

```
private void buildButton_Click(object sender, EventArgs e)
{
    BuildApplication();
}
```

Retrieving Information from the Source Database

The browseSourceButton defined a moment ago calls the method `GetDatabaseInfo`, which retrieves information from the source database when you choose a file to build. The source database will store certain properties as well as the target database. In the source database, we'll store the following properties:

- `AppStartDate` — Used to calculate the version number
- `AppVersionMajor` — Major version number
- `AppVersionMinor` — Minor version number
- `AppVersionRevision` — Revision number of the last build
- `BuildTarget` — The destination database for the build

Write the following code to implement the `GetDatabaseInfo` method. First things first though — we need to open the database:

```
private void GetDatabaseInfo(string databaseName)
{
```

Part IV: Finalizing the Application

```
try
{
    // open the database
    sourceDatabase = myEngine.OpenDatabase(databaseName,
        false,
        false,
        Type.Missing);
```

Next, retrieve the properties from the database and set control values on the form. We've used the Parse methods to make sure that we have valid data:

```
// read database properties from the source database
try
{
    appStartDate.Value = DateTime.Parse(
        GetDatabaseProperty(sourceDatabase, SRC_PRP_STARTDATE));
    outputDatabase.Text = GetDatabaseProperty(
        sourceDatabase, SRC_PRP_TARGET);
    major = int.Parse(GetDatabaseProperty(
        sourceDatabase, SRC_PRP_APPMAJOR));
    minor = int.Parse(GetDatabaseProperty(
        sourceDatabase, SRC_PRP_APPMINOR));
    revision = int.Parse(GetDatabaseProperty(
        sourceDatabase, SRC_PRP_APPREVISION));
}
catch (COMException) { }
```

We need to list the tables in the database to delete data from. To do this, first clear the checked list box as shown:

```
// clear the listbox
deleteDataListbox.Items.Clear();
```

Now, enumerate the TableDefs collection in the source database and add non-system tables:

```
// list the tables
DAO.TableDefs tables = sourceDatabase.TableDefs;
foreach (DAO.TableDef td in tables)
{
    if (!IsSystemTable(td))
    {
        deleteDataListbox.Items.Add(td.Name);
    }
}
```

We also want to default the release date to the current date and then close the method:

```
// default the release date
releaseDate.Value = DateTime.Today;
// close the source database
sourceDatabase.Close();
}
catch (Exception ex)
```

```

    {
        // display an error
        MessageBox.Show(ex.ToString());
    }
}

```

This method includes a call to another method called `GetDatabaseProperty`. This method is used to read a property value from the specified database. In our case, we're creating user-defined properties that appear in the Properties dialog box for the database. Add the following code to implement the `GetDatabaseProperty` method:

```

private string GetDatabaseProperty(DAO.Database database, string propName)
{
    DAO.Container container;
    DAO.Document document = null;
    try
    {
        // get a database property
        container = database.Containers["Databases"];
        document = container.Documents["UserDefined"];
        return document.Properties[propName].Value.ToString();
    }
    catch (COMException ex)
    {
        throw ex;
    }
}

```

We also defined the method `IsSystemTable` to determine whether a `TableDef` object is a system table. Add the following code to implement this helper method:

```

private bool IsSystemTable(DAO.TableDef td)
{
    return (0 != (td.Attributes &
                  (int)DAO.TableDefAttributeEnum.dbSystemObject));
}

```

Building the Application

Follow these steps to write in the `BuildApplication` method to meet the requirements:

- 1.** Create the target database. (This is the end result of the build.)
- 2.** Set build properties on the target database.
- 3.** Set build properties on the source database.
- 4.** Delete data from tables in the target database.

Using programming by intention, write the `BuildApplication` method as follows. We'll fill in the dependent methods along the way. Start with the declaration and a call to the method

Part IV: Finalizing the Application

`CreateTargetDatabase`. We're wrapping the code in a `try/catch` block to handle any exceptions that are thrown:

```
private void BuildApplication()
{
    try
    {
        // get the target database
        targetDatabase = CreateTargetDatabase();
```

Next, we need to set properties on both the target and source databases. To do that, add the following code. These methods will be implemented in a few moments.

```
// set properties on the target and source databases
SetTargetProperties();
SetSourceProperties();
```

The next thing we need to do is delete the data from the selected tables. Add the following code to call a method that will do that:

```
// delete data from tables
DeleteTableData();
```

Next, close the source database and finish the try block:

```
// close the source database
sourceDatabase.Close();
// success!
MessageBox.Show("Application built successfully!",
    "Build Complete",
    MessageBoxButtons.OK, MessageBoxIcon.Information);
}
```

Two different exceptions may be thrown: a `COMException` if one of the user-defined properties is null, and `NullReferenceException` if the target database was not created. Handle both of these exceptions with the following `catch` blocks. We've added a `finally` block to make sure that the target database is closed:

```
catch (COMException ex)
{
    if (ERR_PROP_ISNULL == ex.ErrorCode)
    {
        errorText = "Custom database property cannot be null";
        ShowErrorMessage();
    }
}
catch (NullReferenceException)
{
    ShowErrorMessage();
}
finally
{
    // close target database in the finally block to ensure closure
    targetDatabase.Close();
```

```
    }
}
```

Creating the Target Database

The first thing you should do is create the file that you’re going to build. This is so that you can modify it without changing the original. Start by creating the `CreateTargetDatabase` method. If you check the option to create an ACCDE or MDE file, call another method called `CreateACCDEFfile` that will be defined in a moment.

```
private DAO.Database CreateTargetDatabase()
{
    // creates the destination database
    if (createACCDE.Checked)
    {
        CreateACCDEFfile();
    }
}
```

If you’re not creating an ACCDE or MDE file, you need to make a copy of the file to avoid overwriting the original:

```
else
{
    try
    {
        // copy the file
        File.Copy(inputDatabase.Text, outputDatabase.Text);
    }
}
```

If the copy fails, the .NET Framework will throw an `IOException`. The most common reason for this is that the file already exists. If we receive this exception, we’ll delete the file first and then copy it again.

```
catch (IOException)
{
    // delete the file first and then copy the file
    File.Delete(outputDatabase.Text);
    File.Copy(inputDatabase.Text, outputDatabase.Text);
}
}
```

At this point, the file should be created — be it an ACCDE or MDE file, or simply a copy of the original. Because the `CreateTargetDatabase` method returns a `Database` object in DAO, we need to open the database and return it to the caller.

```
// return the new database (opened)
try
{
    return myEngine.OpenDatabase(
        outputDatabase.Text,
        false,
        false,
        string.Empty);
}
catch (COMException)
```

Part IV: Finalizing the Application

```
    {
        return null;
    }
}
```

To create an ACCDE or MDE file, we're going to cheat. The `SysCmd` function in Access defines a hidden command (603) to create an ACCDE or MDE file. To use this function, we need to use the Access object model, as shown in the following code:

```
private void CreateACCDEFfile()
{
    // launch Access
    myApp = new ACC.ApplicationClass();
    // cannot create MDE files in disabled mode
    // set AutomationSecurity to Low
    myApp.AutomationSecurity =
        OFF.MsoAutomationSecurity.msoAutomationSecurityLow;
    // create an ACCDE/MDE
    ACC.AcSysCmdAction syscmd603 = (ACC.AcSysCmdAction)603;
    object rc = myApp.SysCmd(
        syscmd603,
        inputDatabase.Text,
        outputDatabase.Text);
    // close
    myApp.Quit(ACC.AcQuitOption.acQuitSaveNone);
    myApp = null;
}
```

The `SysCmd` 603 command is not documented by Microsoft. You should be aware that calling undocumented methods is not without risk. While this will work in Access 2007, there are no guarantees that it will be supported in future versions of Access.

Set Build Properties on the Target Database

The target database will contain three properties:

- `AppVersion` — The version number of the application
- `AppReleaseDate` — The release date of the application
- `DEBUG` — Indicates whether the database is a debug build

You can read more about creating debug builds of a database in Chapter 4.

The `BuildApplication` method calls the method `SetTargetProperties` to set properties in the target database. Define this method as follows.

```
private void SetTargetProperties()
{
    SetDatabaseProperty(targetDatabase, DST_PRP_VERSION,
        DAO.DataTypeEnum.dbText, appVersion.Text);
    SetDatabaseProperty(targetDatabase, DST_PRP_RELDATE,
        DAO.DataTypeEnum.dbDate, releaseDate.Value.ToShortDateString());
    SetDatabaseProperty(targetDatabase, DST_PRP_DEBUG,
```

```

        DAO.DataTypeEnum.dbBoolean, debugBuild.Checked);
    // delete properties that may be in the target after doing a copy
    DeleteDatabaseProperty(targetDatabase, SRC_PRP_APPMAJOR);
    DeleteDatabaseProperty(targetDatabase, SRC_PRP_APPMINOR);
    DeleteDatabaseProperty(targetDatabase, SRC_PRP_APPREVISION);
    DeleteDatabaseProperty(targetDatabase, SRC_PRP_STARTDATE);
    DeleteDatabaseProperty(targetDatabase, SRC_PRP_TARGET);
}

```

Okay, so you noticed that this doesn't really do much — other than call another method. The `SetDatabaseProperty` method is used to set the actual property values. This method accepts the `Database` object so it is used to set properties in both the target and source databases. Add the following code to implement the `SetDatabaseProperty` method:

```

private void SetDatabaseProperty(DAO.Database database, string propName,
DAO.DataTypeEnum propType, object propValue)
{
    DAO.Container container;
    DAO.Document document = null;
    try
    {
        // set a database property
        container = database.Containers["Databases"];
        document = container.Documents["UserDefined"];
        document.Properties[propName].Value = propValue;
    }
}

```

You'll notice here that we're setting a property in the `UserDefined` document of the `Databases` container in the database. Properties created here appear in the `Properties` dialog box for the database.

If the specified property is not found, we receive the `Property Not Found` error in DAO. In the .NET Framework, this translates to a `COMException`. If we receive this error, we need to create the property.

```

catch (COMException ex)
{
    switch (ex.ErrorCode)
    {
        case ERR_PROP_NOTFOUND:
            // property not found - create it
            if (!string.IsNullOrEmpty(propValue.ToString()))
            {
                DAO.Property prp = targetDatabase.CreateProperty(
                    propName,
                    propType,
                    propValue,
                    Type.Missing);
                document.Properties.Append(prp);
            }
    }
}

```

If the property value is null, we're going to raise a new `COMException` to the caller to handle:

```

else
{
}

```

Part IV: Finalizing the Application

```
        // raise this exception to the calling routine
        throw new COMException("", ERR_PROP_ISNULL);
    }
    break;
default:
    throw;
}
}
```

If the target database did not get created, our code will receive a `NullReferenceException` so let's handle that as well:

```
catch (NullReferenceException)
{
    // set the error text
    errorText = "Target database does not exist. " +
        "Creating ACCDE/MDE failed." +
        "\n\nVerify that you can compile the " +
        "source database and that there are no errors.";
}
}
```

If we made a copy of the file, the source database properties may be copied to the target database. The `DeleteDatabaseProperty` method is used to delete a property from the specified database.

```
private void DeleteDatabaseProperty(DAO.Database database, string propName)
{
    DAO.Container container;
    DAO.Document document = null;
    try
    {
        // delete a database property
        container = database.Containers["Databases"];
        document = container.Documents["UserDefined"];
        document.Properties.Delete(propName);
    }
    catch (COMException){ }
}
```

Setting Build Properties on the Source Database

Setting properties on the source database is straightforward at this point. We simply need to call the `SetDatabaseProperty` method for the specified properties. Add the following code to implement the `SetSourceProperties` method:

```
private void SetSourceProperties()
{
    // open the source database
    sourceDatabase = myEngine.OpenDatabase(
        inputDatabase.Text,
        false,
        false,
        Type.Missing);
    // set the properties
}
```

```

        SetDatabaseProperty(sourceDatabase, SRC_PRP_STARTDATE,
            DAO.DataTypeEnum.dbDate, appStartDate.Value.ToShortDateString());
        SetDatabaseProperty(sourceDatabase, SRC_PRP_TARGET,
            DAO.DataTypeEnum.dbText, outputDatabase.Text);
        SetDatabaseProperty(sourceDatabase, SRC_PRP_APPMAJOR,
            DAO.DataTypeEnum.dbLong, appVersion.Text.Split('.')[0]);
        SetDatabaseProperty(sourceDatabase, SRC_PRP_APPMINOR,
            DAO.DataTypeEnum.dbLong, appVersion.Text.Split('.')[1]);
        SetDatabaseProperty(sourceDatabase, SRC_PRP_APPREVISION,
            DAO.DataTypeEnum.dbLong, appVersion.Text.Split('.')[3]);
    }
}

```

Deleting Data from Tables

One of the last steps in the `BuildApplication` method is to delete data from the selected tables in the target database. For our user interface, we chose a checked list box in the form because Access doesn't have this control and we think it's cool. The checked list box control has a collection of checked items that we simply need to enumerate.

Add the following code to implement the `DeleteTableData` method. Again, this might throw a `NullReferenceException` if the target database does not exist.

```

private void DeleteTableData()
{
    string sql = null;
    try
    {
        foreach (string tableName in deleteDataListbox.CheckedItems)
        {
            // create the SQL statement and execute
            sql = string.Format("DELETE * FROM [{0}]", tableName);
            targetDatabase.Execute(sql, Type.Missing);
        }
    }
    catch (NullReferenceException)
    {
        // set the error text
        errorText = "Delete table data failed. " +
                    "Target database does not exist.";
    }
}

```

Calculating the Version Number

We're going to use a calculated version number to make it easier to enter. Because the property in the database will ultimately be text, however, you could choose to stamp your own version number in there. The algorithm for this calculation is as follows:

- 1.** Retrieve the major and minor versions from the source database.
- 2.** Using the start date of the application as defined in the source database, calculate the number of months the project has been active. This is the difference (in months) of the current date and the start date of the application.

Part IV: Finalizing the Application

3. Append the day the build is being created. In other words, if today were November 5, we'd append 05 to the build number.
4. Last, we'll append a revision number to the database. The revision number will be stored in the source database and then incremented.

Before we get into the code, let's look at an example. Say that the start date of your application was January 4, 2007 and today is November 5. For the purpose of example, we'll assume that the major and minor version is 1.0. The difference in months between November 5 and January 4 is 10. Because we've already said that today is November 5, we'll append 05. If this is the first revision, we'd leave it at 0. Therefore, the calculated version number is 1.0.1005.0.

Here's the code, beginning with the declarations:

```
private string CalculateVersion()
{
    string appVersion;
    DateTime startDate = appStartDate.Value;
    DateTime relDate = releaseDate.Value;
```

To calculate the number of months, we start by subtracting the start date from the release date, as shown, and then divide by 30. This is an approximation, but it's pretty close. We'll also append the day of the release.

```
// diff the months and append the day
TimeSpan diff = relDate.Subtract(startDate);
string build = (diff.Days / 30).ToString("00");
build += relDate.Day.ToString("00");
```

Last, we need to create the formatted string using the `Format` method of the `String` object.

```
appVersion = string.Format("{0}.{1}", major, minor); // major.minor
appVersion += string.Format(".{0}", build);           // build
appVersion += string.Format(".{0}", revision);        // revision
// return the formatted version
return appVersion;
}
```

The `ToString` method can be used to format data in a particular format. In this case, we've used it to force two characters for the day and number of months.

When you run the application, you should be able to select a database and specify the target database. After setting properties and options, click the `Build` button. Open the target database when complete. You should see the new build properties for the target database in the `Properties` dialog box, as shown in Figure 14-4.

Handling Application Dependencies

Configurations can vary from one machine to another, making application dependencies one of the more difficult aspects of development and deployment. Let's take a look at some ways you can deal with one dependency issue in particular — references.

The following examples tie into one another so begin by creating a new database for the code that follows.



Figure 14-4

Installing Files from an Attachment Field

Certain dependencies can also be stored and subsequently installed from an attachment field. This is particularly useful if you are distributing library databases with your application. Using an attachment field allows you to create setup routines that are built in to the database.

To use an attachment field to contain the dependent files, begin by creating a table with the following schema. Save the table as USysTblDependentFiles when you're done.

As the MDB file format does not support the Attachment data type, this will work only in the new ACCDB file format.

Field Name	Field Type (Size)	Description
ID	AutoNumber	Primary Key.
DependentFiles	Attachment	Contains the dependent files.
Component	Text (255)	Description of the attachment contents. Used in the WHERE clause of a recordset that retrieves setup files and allows you to define multiple files that make up a given component.

In our example, we will store a single ACCDB file in an attachment field. This file will be used as a reference in our application.

Part IV: Finalizing the Application

Create a new class module called `Installer` and add the following code:

```
' installs a file from tblAttachments
Public Function InstallFromAttachment(stComponent As String, _
    stDestPath As String) As Long
    On Error GoTo err_handler:
```

You'll remember from Chapter 7 that the Attachment data type in Access 2007 stores an embedded recordset, which stores each attachment file. Therefore, we'll need two Recordset objects.

```
Dim rs1 As DAO.Recordset2
Dim rs2 As DAO.Recordset2
Dim fld As DAO.Field2
Dim stSQL As String
Dim stDestFile As String
```

Next, we want to open the dependent files table for the selected component and get the recordset of attachments. This will be stored in the variable named `rs2`.

```
' open the attachment table and look for the specified attachment
stSQL = "SELECT * FROM USysTblDependentFiles " & _
    "WHERE Component = '" & stComponent & "'"
Set rs1 = CurrentDb().OpenRecordset(stSQL)
Set rs2 = rs1.Fields("DependentFiles").Value
```

Because the attachment field may store multiple attachments, we should walk through the entire recordset of files. We'll get the `FileData` field for each attachment and save it to the hard drive using the `SaveToFile` method. The `FileData` field in the attachment recordset contains the actual file.

```
If (Not (rs2.EOF)) Then
    ' save the attachment data - may be multiple attachments
    While (Not rs2.EOF)
        Set fld = rs2("FileData")
        stDestFile = stDestPath & "\" & rs2("FileName")
        fld.SaveToFile stDestFile
        ' next attachment
        rs2.MoveNext
    Wend
Else
    Err.Raise vbObjectError + 513, TypeName(Me), _
        "Cannot find files for component: " & stComponent
End If
```

We're raising an error here instead of displaying a message box because this is a class module.

The last thing we should do is clean up our objects and then exit the routine:

```
cleanup:
    DoEvents
    rs2.Close: Set rs2 = Nothing
    rs1.Close: Set rs1 = Nothing
    Set fld = Nothing
    Exit Function
```

```

err_handler:
    If (Err = 3839) Then
        ' file already exists - delete it
        VBA.Kill stDestFile
        Resume
    ElseIf (Err = 91) Then
        Err.Clear
        Exit Function
    ElseIf (Err <> 0) Then
        MsgBox "Unexpected error: " & Err.Description, vbCritical
        Resume cleanup
    End If
End Function

```

The 3839 error code shown in the error handler is raised when the file being saved already exists. In our case, we're deleting the file and updating it from the one in the attachment field.

To test this routine, insert multiple attachments into the USysTblDependentFiles table. Then, add the following test code.

```

Public Sub InstallAttachmentsTest()
    Dim objInstaller As New Installer
    objInstaller.InstallFromAttachment "<YourComponentName>", CurrentProject.Path
    Set objInstaller = Nothing
End Sub

```

This code will install files from the dependent files table to the directory where the database resides. Be sure to change the text *YourComponentName* to match that in your table.

Updating References

References in your application provide library code that can be reused. Whether it's VBA code in another database that is being used as a library, a separate DLL, or ActiveX control, you may need a way to determine whether a reference is missing and subsequently update the reference. Missing references can cause expressions in the application to break.

To determine whether a reference is missing, check the `IsBroken` property of the `Reference` object, as shown in the following code. The `Access Application` object includes a property called `BrokenReference` that can be used as an optimization when iterating through references. This property returns `True` if there is a broken reference in the database. Check this property first to determine whether you need to iterate.

The following code shows you how to iterate through and update a reference to another Access databases. This code includes a call to the `InstallFromAttachment` method shown in the previous example.

Create a new standard module and add the following routine:

```

Public Function EnsureReferences() As Boolean
    Dim ref      As Access.Reference
    Dim stFile As String
    Dim stPath As String
    Dim stName As String
    Dim objInstaller As Installer

```

Part IV: Finalizing the Application

Before we iterate through the references, let's check the `BrokenReference` property:

```
' if there are no missing references, return
If (Not Application.BrokenReference) Then
    EnsureReferences = True
    Exit Function
End If
```

If there are missing references, create a new instance of the `Installer` class.

```
' create the installer instance
Set objInstaller = New Installer
```

Begin a loop to iterate through the `References` collection. We'll check the `IsBroken` property of the `Reference` object to determine whether we need to take an action.

```
' iterate through references
For Each ref In Access.Application.References
    If (ref.IsBroken) Then
        ' save the file information
        stFile = VBA.Mid(ref.FullPath, VBA.InStrRev(ref.FullPath, "\") + 1)
        stPath = Access.Application.CurrentProject.Path
        stName = VBA.Left(stFile, VBA.InStr(stFile, ".") - 1)
```

After saving some information about the reference, remove it.

```
' remove the reference
Access.Application.References.Remove ref
```

Our sample database contains a copy of the library database so we'll install it using the `InstallFromAttachment` code shown earlier. We'll create the library database in the next section.

```
' install it from the attachment field
objInstaller.InstallFromAttachment stName, stPath
```

Next, we'll re-add the reference using the `AddFromFile` method and exit the routine.

```
' re-add the reference
Access.Application.References.AddFromFile stPath & "\" & stFile
End If
Next

' compile
If (Not Access.Application.IsCompiled) Then
    Access.Application.RunCommand acCmdCompileAllModules
End If

' cleanup
Set objInstaller = Nothing

' return
EnsureReferences = True
End Function
```

Testing Reference Fix-Up

Before we can test this routine, we need to do some setup work; namely, we need to create a library database and reference it from the database created in the previous section.

Create the Library Database

To create the library database that will be used by our reference fix-up code, create a new database named `LibraryTest.accdb` and add a new standard module. Save the module with the name `My`. Add the following code to the module in the library database:

```

Public Property Get Name() As String
    Name = Environ("USERNAME")
End Property
Public Property Get Computer() As String
    Computer = Environ("COMPUTERNAME")
End Property
Public Property Get CurrentForm() As Form
    Set CurrentForm = Screen.ActiveForm
End Property
Public Property Get Project() As CurrentProject
    Set Project = CurrentProject
End Property
Public Property Get Library() As CodeProject
    Set Library = CodeProject
End Property

```

Call Code in the Library Database

After you save the code in the library database, re-open the database that contains the `EnsureReferences` function written earlier. Open the `USysTblDependentFiles` table and add the `LibraryTest.accdb` file in the `Attachment` field, as shown in Figure 14-5.

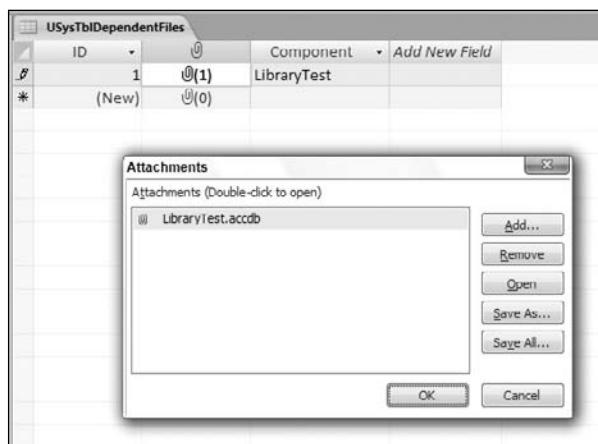


Figure 14-5

Part IV: Finalizing the Application

Insert a new standard module in the first database that will call code in the library database. Add a reference to the LibraryTest.accdb database, and then add the following code to the module:

```
Sub TestLibraryCode()
    ' calls code in the library
    Debug.Print My.Computer
    Debug.Print My.Name
    Debug.Print My.Library.FullName
    Debug.Print My.Project.FullName

End Sub
```

Run the code to verify that it works.

Break the Reference

To test that the `EnsureReferences` function works, close the database, and then rename the `LibraryTest.accdb` file you just created. Re-open the database and run the `TestLibraryCode` routine. The code should fail with a missing reference error. Now, run the `EnsureReferences` routine and re-run the `TestLibraryCode` routine. You should have a copy of the library database in the directory where the database resides and the routine should run successfully.

Any breakpoints in code will not work when you manipulate the VBA project programmatically. If you have a breakpoint set, you may see the message “Can’t enter break mode at this time” when the reference is removed.

Late Binding

During development, using early binding is easier because it provides IntelliSense. For deployment, however, late binding is often preferred because code continues to compile regardless of whether a reference is missing. In addition, a runtime error is displayed when you try to create an object if the object is not installed. It’s often said that there is a performance penalty when using late binding compared to early binding, however, we find that this is negligible with today’s modern hardware.

In late binding, variables are defined as `Object` instead of the strongly typed data type for an object. For example, if you were automating objects in the Excel 2007 object model, you might have code that looks something like this:

```
Dim objXL      As Excel.Application
Dim objBook    As Excel.Workbook
Dim objSheet  As Excel.Worksheet
' Launch Excel
Set objXL = New Excel.Application
```

With late binding, this becomes:

```
Dim objXL      As Object ' Excel.Application
Dim objBook    As Object ' Excel.Workbook
Dim objSheet  As Object ' Excel.Worksheet
' Launch Excel
Set objXL = CreateObject("Excel.Application")
```

Notice that we've changed the variable declarations of the objects to use the `Object` data type in VBA. We still like to include comments in the code for documentation so that we will always know what we're working with. If the `CreateObject` function fails, it will throw runtime error 429.

With late binding, you may also consider using the `IsNothing` function defined in Chapter 3 before calling a method on an object. For example, to open a workbook in Excel using late binding, you might write it something like this:

```
Dim objXL    As Object ' Excel.Application
Dim objBook  As Object ' Excel.Workbook
Dim objSheet As Object ' Excel.Worksheet
' Launch Excel
Set objXL = CreateObject("Excel.Application")
' Get the workbook
If (Not IsNothing(objXL)) Then
    Set objBook = objXL.Workbooks.Open("<PathToWorkbook.xls>")
End If
```

As a reminder, the `IsNothing` function is defined as follows:

```
Public Function IsNothing(obj As Object) As Boolean
    IsNothing = (obj Is Nothing)
End Function
```

Licensing Your Applications

For an off-the-shelf application, you may want to license the application in some manner or limit its functionality until someone has registered it with you. Limiting the functionality allows you to create trial versions of your application for distribution.

Even for internal applications with a large rollout, there may be beta testing of an application within an organization. These applications will likely provide the full functionality, but you may want to specify that the beta version is no longer used after a certain period of time.

Creating a Limited-Use Application

A limited-use application is one where you limit what the application can do, typically based on some registration. For a database application, there are a few different ways that you might reduce functionality. First, you might provide an application that provides full functionality but only until a certain date. Second, you might limit the number of records that a user can enter in a given table in the database. If your application is large and contains a number of functional components, a third option is to simply provide a subset of the functionality. Last, you might choose to restrict the number of times an application is launched.

Let's take a look at the different ways to do this in your applications.

Regardless of how you limit functionality, users should know that they're running in a reduced mode.

Part IV: Finalizing the Application

Expiration Date

The classic mechanism for limiting the use of an application is an expiration date. This practice is still going on today with beta versions of software such as Microsoft Office. Expiration dates have one major issue that you need to watch out for — whether a user can bypass the expiration date mechanism by adjusting the time on their computer. The solution shown here takes this into account.

To prevent users from finding the information easily, we're using the Registry to store information about when the application was used and its expiration date. Start by creating a new standard module and add the following declarations. We're using the same hive in the Registry used when we discussed the RegRead method in Chapter 12.

```
' Registry constants
Private Const REG_EXPDATE As String = _
    "HKCU\Software\MyCompany\MyApplication\ExpirationDate"
Private Const REG_LASTUSED As String = _
    "HKCU\Software\MyCompany\MyApplication>LastUsedDate"
Private Const REG_SZ As String = "REG_SZ"
Private Const EXPIRATION_LENGTH_DAYS As Long = 30
```

Create a routine called SetExpirationDate that sets the expiration date as a value in the Registry. The expiration date is defined by the current date and time plus 30 days.

```
Private Sub SetExpirationDate()
    ' update the last use date
    Dim objShell As Object
    Dim expDate As Date

    Set objShell = CreateObject("WScript.Shell")
```

We'll try to read the expiration date from the Registry. If the key does not exist, we'll call the RegWrite method of the WshShell object to write it. The error handling acts as a precaution to ensure that the expiration date is not updated unless the value has not been set.

```
On Error Resume Next
expDate = CDate(objShell.RegRead(REG_EXPDATE))

If (Err = &H80070002) Then
    ' key does not exist, create it
    objShell.RegWrite REG_EXPDATE, Now() + EXPIRATION_LENGTH_DAYS, REG_SZ
End If

On Error GoTo 0
Set objShell = Nothing
End Sub
```

In addition to storing the expiration date, we'll store the last date that the application was used. This is to prevent the user from tampering with their system clock to work around the expiration date. This routine sets the Registry key defined by the constant REG_LASTUSED to the current date and time.

```
Private Sub UpdateLastUsedDate()
    Dim objShell As Object
```

```

' write the key value
Set objShell = CreateObject("WScript.Shell")
objShell.RegWrite REG_LASTUSED, Now(), REG_SZ

Set objShell = Nothing
End Sub

```

The last function we need will determine whether the application is actually expired. To do this we're going to read both values from the Registry. If either of them doesn't exist, then this is a first boot scenario and we'll call one of the previous routines.

Start by declaring the routine:

```

Public Function IsExpired() As Boolean
    ' determines whether the expiration date has passed
    Dim expDate As Date
    Dim lastUsed As Date
    Dim objShell As Object

    Set objShell = CreateObject("WScript.Shell")

```

Next, try to read the expiration date. If this fails, call the `SetExpirationDate` routine defined earlier.

```

' read the values
On Error Resume Next
expDate = objShell.RegRead(REG_EXPDATE)

' first use
If (Err) Then
    SetExpirationDate
    IsExpired = False
    Exit Function
End If

```

Try to read the last used date in the Registry. If this fails, call the `UpdateLastUsedDate` routine defined earlier:

```

' check the last used date
lastUsed = objShell.RegRead(REG_LASTUSED)
If (Err) Then
    UpdateLastUsedDate
    IsExpired = False
    Exit Function
End If
On Error GoTo 0

```

We need to add the logic to determine whether the system clock has been tampered with and whether the application is expired. For starters, we know that if the current date and time is before the last used date and time (as stored in the Registry), the system clock has been altered. We'll return `True` in this case.

```

' if the current system date is less than the last use date then
' the user messed with their system clock

```

Part IV: Finalizing the Application

```
If (Now() < lastUsed) Then  
    IsExpired = True  
Else
```

If this check succeeds, then we need to compare the last used date against the expiration date. If the application is not expired we'll then update the last used date in the Registry.

```
' system clock is ok so check the expiration date  
' against the last use date  
IsExpired = (lastUsed >= expDate)  
  
' update the last usage date  
If (Not IsExpired) Then  
    UpdateLastUsedDate  
End If  
End If  
Set objShell = Nothing  
End Function
```

Number of Records

A nice approach for reducing functionality in a database application is to simply limit the number of records. After all, what good is a database where you can store only a few records? There are two ways to do this, one of which requires code and one that doesn't. The method that requires code is to use the BeforeUpdate event of a bound form. In this case, cancel the event when you exceed your pre-set maximum number of records, as shown in the following code:

```
' max number of records  
Private Const MAX_RECORDS As Long = 10  
Private Sub Form_BeforeUpdate(Cancel As Integer)  
    Dim n As Integer  
    n = Nz(DCount("*", "tblAssets"), 0)  
    If (n + 1 > MAX_RECORDS) Then  
        Cancel = True  
        MsgBox "Cannot add additional records in this edition of the database.", _  
            vbExclamation, "TRIAL VERSION"  
    End If  
End Sub
```

The second technique is to use the validation rule property of an AutoNumber field in a particular table. For example, say that you have a field in a table called ID, which is an AutoNumber. Set the validation rule property of this table to <= 10 to prevent adding more than 10 records to the table.

This last technique is not bulletproof. The AutoNumber value is incremented on first edit, so if you begin inserting a record but undo the commit, the AutoNumber seed is still incremented. As a result, users may not be able to store 10 records in the table.

Limited Functionality

One of the easier ways to limit the use of an application is to simply not ship all of it. This can be challenging to do if there are dependencies between features of your application. You might also consider leaving user interface entry points to the missing features as advertisement for the full version.

Restricting the Number of Times an Application is Launched

Instead of limiting an application by expiration date, you may restrict its overall usage by the number of times it is used. This gives users the flexibility to use the application as they need without a time limit as such. This scenario also has a drawback: Users who figure out what's happening might just launch the application and leave it open. You might work around this drawback by detecting idle time in the database and shutting it down after a period of time.

To track the number of times the application is launched, we're going to use a table. Depending on the amount of detail you wanted to display, there are a number of ways you could create the table. You could simply create a table that contains one field with one record that is incremented each time the database opens. We want to see a little more detail though so we're going to create a table that tracks the usage of the database. Start by creating a new table called `USysUsageInfo`. Add the fields shown in the following table.

Field Name	Data Type (Size)
ID	AutoNumber
UsageDate	Date/Time
UserName	Text (255)

Next, create a form called `USysFrmUsageInfo` and add the following code to the `Load` event of the form:

```

Private Const MAX_USAGE As Long = 10
Private Sub Form_Load()
    ' update the usage information
    Dim rs As DAO.Recordset
    Set rs = CurrentDb().OpenRecordset("USysUsageInfo")

    If (rs.RecordCount = MAX_USAGE) Then
        ' user has already reached the maximum
        MsgBox "You have reached your limit of " & MAX_USAGE & _
               " uses of this application. Please contact us for a full version.",
               vbExclamation, _
               "TRIAL VERSION"

        ' quit
        Application.Quit
    Else
        ' add a record to the usage table
        rs.AddNew
        rs!UsageDate = Now()
        rs!UserName = Environ("USERNAME")
        rs.Update
    End If

    ' close the form
    DoCmd.Close acForm, Me.Name
End Sub

```

Part IV: Finalizing the Application

In this code, when the maximum number of usages is reached, we display a message to the user and quit the application. Until this value is reached we add a record to the `USysUsageInfo` table. Because we don't need the form hanging around, we close it when we're done.

To test this code, create a new `autoexec` macro with an `OpenForm` action that opens this form. Open the application ten times. On the eleventh attempt to open the application, you should see the warning that you've exceeded the number of times it can be used.

Registering an Application

In order to unlock a limited-use application, or even to use an application for the first time, you may require users to register with you. Let's take a look at ways you can generate registration keys to be used by your application and then how to validate one of the keys.

Generating Registration Keys

A registration key can be as simple or as complex as you decide to make it. Registry keys can be formed with numbers, characters, or a mixture of the two. Using code, we can generate registration keys to be used by an application. In the next section, we take a look at how to verify whether a key is valid.

Our registration keys will use randomly generated ASCII characters within a range of valid characters. There are also certain characters that we don't want included in the key. Start by creating a new module and add the following constants to the module. These constants define the range of valid characters for the key. To exclude additional characters, change the `EXCLUSIONS` constant shown here:

```
' character constants
Private Const ASCII_0 As Long = 48      ' numeric 0
Private Const ASCII_9 As Long = 57      ' numeric 9
Private Const ASCII_A As Long = 65      ' upper-case A
Private Const ASCII_Z As Long = 122     ' lower-case z
Private Const EXCLUSIONS As String = "[\]^`?<>,.:'{}|/+=%$#@!~"""
```

Numeric Registration Keys

To generate a numeric registration key, we need to generate a string of numbers between the characters for 0 to 9. Add the following routine to the module to generate a random string of digits:

```
Public Function GetNumericKey(bLength As Byte) As String
    Dim i As Byte
    Dim s As String
    Dim ch As Integer

    For i = 1 To bLength
        ' get a random character
        ch = Int((ASCII_9 - ASCII_0 + 1) * Rnd + ASCII_0)

        ' append it to the string
        s = s & Chr(ch)
    Next

    GetNumericKey = s
End Function
```

Alpha-Numeric Registration Keys

An alpha-numeric key will contain both letters and numbers, but not symbols defined in the exclusion list. Start by creating a new routine called `GetAlphaNumericKey` with the following declarations:

```
Public Function GetAlphaNumericKey(bLength As Byte) As String
    Dim i As Byte
    Dim s As String
    Dim ch As Integer
    For i = 1 To bLength
```

Next, we need to get a random character that is not in the list. To do this, we'll get a character as shown in the previous example, but check it against the `EXCLUSIONS` constant using the `InStr` function:

```
' get a random character that is not in the exclusion list
' make sure it is not an excluded character
Do
    ch = Int((ASCII_z - ASCII_0 + 1) * Rnd + ASCII_0)
Loop Until (InStr(EXCLUSIONS, Chr(ch)) = 0)

' append it to the string
s = s & Chr(ch)
Next

GetAlphaNumericKey = s
End Function
```

Testing the Key Generation

To test these functions, add the following routine to the module. Replace the `Debug.Print` statement shown here to add the random keys to a file or table to use them later:

```
Public Sub TestRandomKeys()
    Dim i As Integer

    For i = 1 To 3
        Debug.Print GetNumericKey(10)
    Next

    For i = 1 To 3
        Debug.Print GetAlphaNumericKey(25)
    Next
End Sub
```

When you run this code, you should see output similar to the following in the Immediate window:

```
3814078245
0690709966
1196811203
sAdn6ml75RvFBG70wgPRRpNeL
MAA5kmHOzdTnIsVkJxKFjs8sr
hHXmkPX54ny1aw9LTsm3tTb41
```

By modifying the list of exclusions, these functions can also be used to create randomly generated strong passwords.

Part IV: Finalizing the Application

Creating a Registration Web Service

Once you've generated some registration keys, you'll need some mechanism for users to validate the key. In this example, we'll create a Web service that communicates with a SQL Server database on the server that contains registration keys.

A detailed discussion of creating Web services including service-oriented architectures (SOA) or SQL Server security is beyond the scope of this book.

Create the Registration Database

The registration database will be stored on SQL Server. For the purposes of the example, we are running SQL Server 2005 Express Edition on Windows 2003 Server with Internet Information Services (IIS). In a production environment, you might run SQL Server and IIS on separate computers.

Begin by creating a new database on SQL Server called AppReg. Create a new table in this database called PID with the following columns:

Field Name	Data Type	Properties
id	int	Identity (1,1)
pidkey	nvarchar (255)	
used	bit	

The id column is an IDENTITY column similar to the AutoNumber data type in Access. The pidkey column is used to store the actual registration key. The used column is a bit field that will be set to True when a successful registration occurs. This is also used to prevent existing registration keys from being reused.

Add some data to this table. The registration keys that you add will be used later.

The Web service will connect to the database using SQL authentication. Make sure SQL authentication is enabled on the SQL Server. Add a new user to the SQL Server called wsuser with SELECT and UPDATE permissions on the PID table.

Create the Web Service

We will write the Web service using C# in Visual Studio 2005. Follow these steps to set up the Web service and prepare it for code.

- 1.** Launch Visual Studio 2005.
- 2.** Click File, then click New, and choose Web Site.
- 3.** Choose ASP.NET Web Service in the New Web Site dialog box.
- 4.** Change the location to HTTP and enter the address of a Web server to which you have access as follows:

`http://servername/regws`

5. Rename the Service.asmx file in the Web site to Registration.asmx.
6. In the Registration.asmx file, change the CodeBehind attribute of the Web service to point to Registration.cs, and the Class attribute from Service to Registration.
7. Rename the Service.cs file in the App_Code folder in the Web site to Registration.cs.
8. In the Registration.cs source file, rename the class from Service to Registration.
9. Remove the HelloWorld method that appears in the Registration.cs source file.
10. Change the Namespace property of the WebService attribute of the class from `http://tempuri.org` to `http://www.wrox.com/ExpertAccess2007Book`.
11. Select Add New Item from the Website menu in Visual Studio and choose Web Configuration File, and then click OK.
12. Edit the web.config file to add the following code beneath the system.web element in the configuration file.

```
<webServices>
    <protocols>
        <add name="HttpGet" />
    </protocols>
</webServices>
```

With the core steps out of the way, we can add some code. Begin by adding the following using statements to the top of the Registration.cs source file:

```
using System.Data;
using System.Data.SqlClient;
```

Add the following private data to the class:

```
SqlConnection m_connection = null;
```

Add the following WebMethod to the class. This is the actual method that will be executed using the Web service.

```
[WebMethod]
public bool IsValidPIDKey(string pidKey)
{
    try
    {
        if (GetPIDCount(pidKey) > 0)
        {
            UpdatePIDRegistration(pidKey);
            return true;
        }
        return false;
    }
    catch (Exception)
    {
        throw;
    }
}
```

Part IV: Finalizing the Application

```
        finally
        {
            m_connection.Close();
        }
    }
```

This method calls two helper methods. The first method, `GetPIDCount`, returns the number of unused registration keys that match the specified key in the PID table.

```
private int GetPIDCount(string pidKey)
{
    string connString = "Data Source=<SQLServerName>";
    connString += ";Initial Catalog=AppReg";
    connString += ";User ID=wsuser";
    connString += ";Password=<YourPassword>";
    // this is the actual action we want to take
    m_connection = new SqlConnection(connString);
    m_connection.Open();
    // build the command text
    string commandText = string.Format("SELECT COUNT(*) FROM " +
        "pid WHERE used = 0 AND pidkey = {0}", pidKey);
    SqlCommand command = new SqlCommand(commandText);
    command.Connection = m_connection;
    command.CommandType = CommandType.Text;
    // return
    return (int)command.ExecuteScalar();
}
```

The next method, `UpdatePIDRegistration`, updates the PID table when a registration key is used:

```
private void UpdatePIDRegistration(string pidKey)
{
    if (m_connection.State == ConnectionState.Closed)
    {
        m_connection.Open();
    }
    string commandText = string.Format("UPDATE pid SET used = 1 WHERE " +
        "pidkey = {0}", pidKey);
    SqlCommand command = new SqlCommand(commandText);
    command.Connection = m_connection;
    command.CommandType = CommandType.Text;
    // execute
    command.ExecuteNonQuery();
}
```

Save the source file and build the Web site.

A version of the Web service written in Visual Basic .NET 2005 is available with the sample code for download with this book.

Validate the Registration Using the Web Service

To validate a registration key using the service, create a new database with a standard module. Add the following VBA code to the module. We'll use the `XMLHTTP` object to send a GET request to the URL for the Web service.

We're using MSXML5, which is included with Office 2007.

```
Public Function IsValidRegistration(stRegKey As String) As Boolean
    Dim xmlhttp As Object
    Dim szUrl As String
    Dim stResponse As String
    Dim xmldom As Object
```

Create the `XMLHTTP` and `DOMDocument` objects as follows:

```
Set xmlhttp = CreateObject("MSXML2.XMLHTTP.5.0")
Set xmldom = CreateObject("MSXML2.DOMDocument.5.0")
```

Specify the URL, which includes the call to the Web service. Notice that we're performing a GET request by specifying the parameters in the URL itself.

```
' set the URL
szUrl = "http://<ServerName>/regws/registration.asmx/IsValidPIDKey?pidKey=" &
stRegKey
```

Tell the `XMLHTTP` object to issue the GET request and send it to the server:

```
' get the XML from the web service
xmlhttp.Open "GET", szUrl, False
xmlhttp.send

' wait for a response
While (xmlhttp.ReadyState <> 4)
    DoEvents
Wend
stResponse = xmlhttp.responseText
```

Because we are using an ASP.NET Web service, the response will come back as XML. Load this XML into a `DOMDocument` object using the `loadXML` method. We'll then parse this XML and look for the result of the `IsValidPIDKey` method defined in the service:

```
' parse the response
If (Not (xmldom.loadXML(stResponse))) Then
    ' could not parse - return False
    IsValidRegistration = False
Else
    IsValidRegistration = _
        (xmldom.documentElement.childNodes(0).nodeValue = "true")
End If

End Function
```

Part IV: Finalizing the Application

To test the service, call the `IsValidRegistration` function in the Immediate window. Pass in a valid registration key twice. The function should return `True` the first time and `False` the second time when the PID table is updated.

If a Web service is not the appropriate mechanism for server validation in your architecture, there are many other methods you can use to validate a registration key. Other choices might include e-mail, telephone, key exchange using a text or XML file, or checking a file location for internal applications on a given network.

Miscellaneous Deployment Scenarios

Once the application has been deployed and users begin to use it in the real world, there still may be work to be done. For example, what happens if you have a split database and the back-end database moves? Or, if you want to publish an update to your application, how do you verify that users always have the latest version? We'll look at ways you can accomplish these tasks in your applications to help prevent failures from occurring and keep applications up-to-date.

Updating Your Applications

Keeping an application up-to-date can also be a challenge. If you're writing applications within an organization, you may have to coordinate with everyone to close the database. If you're writing an off-the-shelf application, your customers may be spread over a wide distance making it even more difficult. Here are a few things you should think about to keep an application up-to-date.

Checking for Latest Version

Earlier we defined a version number that we could stamp in our applications. This version number can then be used to determine when a newer version of the application is available. In order to detect an update, you'll need some other piece of information available, typically in a central location. The solution will contain two databases. The first is called `VersioningServer.accdb` and will contain the application history including the version numbers, as well as a class module that is responsible for doing most of the work. The second database is the actual client application, which for the purposes of demonstration, we've simply named `ClientApp.accdb`. Both of these files are available for download with the sample files included with this book.

Figure 14-6 shows the architecture and flow of such a solution.

Updates may be best served when the data in the database is linked. Before you rename the previous database, be sure there is a backup!

Create the Versioning Server

Begin by creating `VersioningServer.accdb`. In this database, create a new table with the fields shown in the following table.

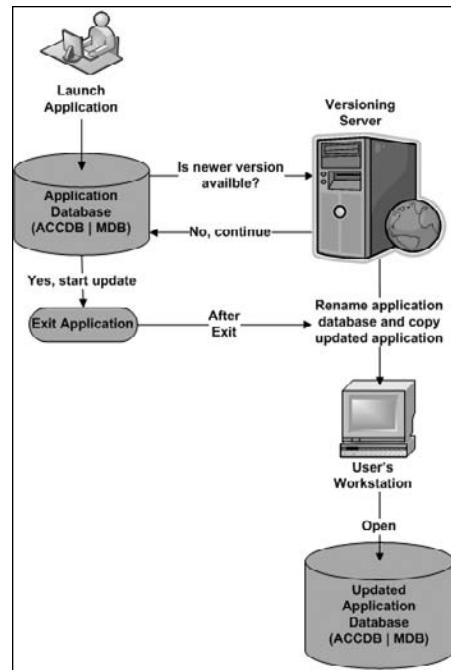


Figure 14-6

Field Name	Data Type (Size)
ID	AutoNumber (Primary Key)
VersionNumber	Text (25)
VersionDescription	Text (255)
FilePath	Text (255)

Save this table with the name `USysAppHistory`. Enter one record as shown in the table that follows.

ID	VersionNumber	VersionDescription	FilePath
1	1.0	Initial release	

Be sure to change the path to the ClientApp.accdb on your computer. This path should represent a copy of the ClientApp.accdb file that we'll create later.

Part IV: Finalizing the Application

Next, create a new form in the versioning server called `frmTimer`. Add the following code to the Timer event for the form. The Timer event is used to give the original client application enough time to close. This event will start the actual updating of the application.

```
Private Sub Form_Timer()
    ' turn off the timer
    Me.TimerInterval = 0

    ' kick off the actual update
    m_updater.Update
End Sub
```

Next, create a new class module called `Updater`. This class will be responsible for doing the bulk of the work. Add the following declaration code to the class. Notice that for simplicity we're using a `Public` variable called `ApplicationDatabase` in place of a Property procedure.

```
Private Const TIMER_INTERVAL As Long = 3000
Private Const IDX_MAJOR As Integer = 0
Private Const IDX_MINOR As Integer = 1
Private Const IDX_BUILD As Integer = 2
Private Const IDX_REVISION As Integer = 3

Public ApplicationDatabase As String
```

Next, add a new method to the class called `StartUpdate`. This method will open the `frmTimer` form and set the `TimerInterval` property of the form.

```
Public Sub StartUpdate()
    ' open the timer form
    DoCmd.OpenForm "frmTimer"

    ' sets the timer that will start the update
    Forms("frmTimer").TimerInterval = TIMER_INTERVAL
End Sub
```

To determine whether a client is running the latest version of the application, we'll create a method in the server application called `IsLatestVersion`. Declare the method as follows:

```
Public Function IsLatestVersion() As Boolean
    Dim v1 As String
    Dim v2 As String
    Dim stSQL As String
    Dim db As DAO.Database
    Dim rs As DAO.Recordset2
```

Make sure that the `ApplicationDatabase` property has been set and is valid. If it has not been set or does not exist, we'll assume that the client is running the latest version and return `True`. This is to prevent any update later on.

```
' validate the ApplicationDatabase
If (Len(ApplicationDatabase) = 0 Or Dir(ApplicationDatabase) = "") Then
    MsgBox "ApplicationDatabase property not set", vbExclamation
```

```

    ' assume the database is current so we don't try to update it later
    IsLatestVersion = True
    Exit Function
End If

```

It's time to start retrieving the versions of the applications. We'll start with the client application. The version information has been stored in a custom database property called AppVersion. This is similar to the solution used earlier in this chapter. To retrieve this property, we'll open the database using DAO, and then call a helper function called GetVersionOfDatabase, which will be defined shortly.

```

' get the version of the application database
Set db = DBEngine.OpenDatabase(ApplicationDatabase, False, True, "")
v1 = GetVersionOfDatabase(db)
db.Close
Set db = Nothing

```

Next, we need to retrieve the latest version number from the USysAppHistory table. To do this, we'll open a recordset against this table that retrieves the version number for the record with the highest ID number:

```

' get the latest version of the database defined in USysAppHistory
stSQL = "SELECT VersionNumber FROM USysAppHistory " & _
        "WHERE ID=(SELECT Max([ID]) FROM USysAppHistory)"
Set rs = CurrentDb().OpenRecordset(stSQL)
v2 = rs("VersionNumber")
rs.Close
Set rs = Nothing

```

Last, we'll call a helper function called CompareVersions that compares the two version strings to determine which one is greater.

```

    ' return
    IsLatestVersion = (CompareVersions(v1, v2) <= 0)
End Function

```

Before we write the code that does the update, let's take a look at the two helper functions we declared, beginning with GetVersionOfDatabase:

```

Private Function GetVersionOfDatabase(db As DAO.Database) As String
    GetVersionOfDatabase =
        db.Containers("Databases").Documents("UserDefined").Properties("AppVersion").Value
End Function

```

The other helper function we declared is CompareVersions. The design of this function is modeled after the Compare method of the String object in the .NET Framework. The function returns -1 if the first version is greater than the second version, 1 if the second version is greater than the first version, and 0 if they are equal.

```

Private Function CompareVersions(ByVal stVersion1 As String, _
                                ByVal stVersion2 As String) As Integer
    ' parse the version strings
    Dim astVersion1() As String
    Dim astVersion2() As String
    Dim iCnt As Integer

```

Part IV: Finalizing the Application

To ensure that version numbers are always in a consistent format, we'll call the helper function `NormalizeVersionString`. This function formats a version number in the format: major.minor.build.revision. If the build and revision parameters are not supplied, the function fills in zeros:

```
' normalize the strings into
' major.minor.build.revision
stVersion1 = NormalizeVersionString(stVersion1)
stVersion2 = NormalizeVersionString(stVersion2)
```

Split the strings using a period as the delimiter. The `Split` function returns an array for each item of a string given a delimiter.

```
' split the strings
astVersion1 = Split(stVersion1, ".")
astVersion2 = Split(stVersion2, ".")
```

Next, loop through each element of the arrays and compare the array for the first version to the array of the second version:

```
For iCnt = IDX_MAJOR To IDX_REVISION
    ' validate - make sure there is an index
    If (UBound(astVersion1) >= iCnt And UBound(astVersion2) >= iCnt) Then
        If (CLng(astVersion1(iCnt)) > CLng(astVersion2(iCnt))) Then
            ' stVersion1 > stVersion2
            CompareVersions = -1
            Exit Function
        ElseIf (CLng(astVersion1(iCnt)) < CLng(astVersion2(iCnt))) Then
            ' stVersion2 > stVersion1
            CompareVersions = 1
            Exit Function
        Else
            ' equal: do not exit because we need to make it all the way through
            CompareVersions = 0
        End If
    End If
Next
End Function
```

Add the following code for the `NormalizeVersionString` function:

```
Private Function NormalizeVersionString(stVersion As String) As String
    ' parse the version strings
    Dim a() As String
    a = Split(stVersion, ".")  
  
    ' normalize the strings:
    ' major.minor.build.revision
    If (UBound(a) = IDX_MINOR) Then
        stVersion = stVersion & ".0.0"
    ElseIf (UBound(a) = IDX_BUILD) Then
        stVersion = stVersion & ".0"
    End If
```

```
    NormalizeVersionString = stVersion
End Function
```

With those functions out of the way, we can begin to see how the updates are actually done. Begin by declaring a new method called `Update`.

```
Public Sub Update()
    Dim obj      As Access.Application
    Dim rs       As DAO.Recordset2
    Dim stBackup As String
    Dim stSQL    As String
```

Earlier, we defined a method called `StartUpdate`, which opens the `frmTimer` form. If this form was not opened, we know that the `StartUpdate` method was not called. Make sure the form is open and display an error otherwise.

```
' make sure that the Timer form is open
' used to make sure that StartUpdate has been called
If (Not CurrentProject.AllForms("frmTimer").IsLoaded) Then
    MsgBox "Cannot call Update method without StartUpdate method", _
           vbExclamation
    Exit Sub
End If
```

It is the responsibility of the client application to close itself if it is not running the latest version. To make sure that the file has been closed, we'll loop until the `LDB` file for the client application has been deleted and no longer exists:

```
' wait for the file to close
Do
    DoEvents
Loop Until (Dir(LDBOfDB(ApplicationDatabase)) = "")
```

Next, we'll make a rudimentary backup of the client application by appending a `.old` file extension to it. This deletes an existing file with this name before making a backup:

```
' delete the existing database that was backed up as .old
stBackup = ApplicationDatabase & ".old"
If (Dir(stBackup) <> "") Then
    Kill stBackup
End If

' rename the existing database
Name ApplicationDatabase As stBackup
```

The path to the updated version of the file is stored in the `USysAppHistory` table. Add the following code to copy this file to the path defined by the `ApplicationDatabase` property.

```
' copy the new file
stSQL = "SELECT VersionNumber, FilePath FROM USysAppHistory " & _
        "WHERE ID=(SELECT Max([ID]) FROM USysAppHistory)"
Set rs = CurrentDb().OpenRecordset(stSQL)
```

Part IV: Finalizing the Application

```
FileCopy rs("FilePath"), ApplicationDatabase
rs.Close
Set rs = Nothing
```

Almost done. We need to start the client application. To do this, we'll automate a new instance of Access and set the `UserControl` property to `True` so the user can interact with it:

```
' start the new instance
If (Dir(ApplicationDatabase) <> "") Then
    Set obj = New Access.Application
    obj.AutomationSecurity = 1 ' Low
    obj.UserControl = True
    obj.OpenCurrentDatabase ApplicationDatabase
End If
```

Last, after the new instance has been launched, close the versioning server database:

```
' cleanup
Application.Quit acQuitSaveNone
End Sub
```

We defined a helper function in this method called `LDBOfDB`, which returns the name of an `.ldb` or `.laccdb` file for a given database.

```
' Gets the LDB file name for a given database
Private Function LDBOfDB(stDb As String) As String
    Dim stExt As String
    stExt = Mid(stDb, InStrRev(stDb, ".") + 1)

    If (stExt Like "md*") Then
        LDBOfDB = Left(stDb, InStrRev(stDb, ".") - 1) & ".ldb"
    ElseIf (stExt Like "accd*") Then
        LDBOfDB = Left(stDb, InStrRev(stDb, ".") - 1) & ".laccdb"
    End If
End Function
```

To return an instance of this class to the client application, we need to create a routine that returns an instance of it. Create a new standard module called `basUpdater` with the following code.

```
Public m_updater As Updater
Public Function GetUpdaterInstance() As Updater
    If (m_updater Is Nothing) Then
        Set m_updater = New Updater
    End If

    Set GetUpdaterInstance = m_updater
End Function
```

Because class modules are Private by default, this code will fail when you call it from the client application. This is because the client application won't be able to create an instance of the Updater class. To solve this problem, set the Instancing property of the Updater class to PublicNotCreatable. The GetUpdaterInstance function in the versioning server is responsible for creating an instance of the class but it needs to be visible to the client application.

Create the Client Application

The hard part is done — we just need to hook it up into a client application. Start by creating a new database called `ClientApp.accdb`. Create the `AppVersion` property in this database as follows:

1. Click the Office menu and choose Manage; then click Database Properties.
2. Choose the Custom tab in the Properties dialog box.
3. Type **AppVersion** in the Name text box.
4. Type **1.0** in the Value text box, and then click Add. Click OK to close the dialog box.

Next, create a new form called `frmCheckVersion` and add the following code to the `Load` event of the form.

```
Private Sub Form_Load()
    Dim obj As Access.Application
    Dim upd As Object
```

We're going to automate the versioning server to get an instance of the `Updater` class. To do this, we need to run code in this database. Launch a new instance of Access and set the `AutomationSecurity` property of the new instance to allow code to run:

```
Set obj = New Access.Application
obj.AutomationSecurity = 1
```

Next, open the versioning server and hide it. Change the path to the versioning server on your computer.

```
obj.OpenCurrentDatabase "<PathTo>\VersioningServer.accdb"
obj.Visible = False
```

Because the client application doesn't have a class module called `Updater`, we're using late binding to the `Updater` class in the versioning server. Remember that the server application has a function called `GetUpdaterInstance` to give us an instance of this class. Add the following code to get a late-bound instance of the `Updater` class and set the `ApplicationDatabase` property:

```
' get the object
Set upd = obj.Run("GetUpdaterInstance")
upd.ApplicationDatabase = CurrentDb.Name
```

Last, use the `IsLatestVersion` function in the class to determine whether the client is running the latest version. If it is not, call the `StartUpdate` method of the `Updater` class in the server application. If the client is running the latest version, we'll display the version number from the `AppVersion` property and close the `frmCheckVersion` form.

```
If (Not upd.IsLatestVersion) Then
    ' start the update - sets the timer interval of frmTimer
    upd.StartUpdate

    ' close this database
    Application.Quit acQuitSaveNone
Else
    MsgBox "You are running version: " & _
    CurrentDb.Containers("Databases").Documents("UserDefined").Properties("AppVersion")
```

Part IV: Finalizing the Application

```
DoCmd.Close acForm, Me.Name  
End If  
End Sub
```

Testing the Versioning Server

Follow these steps to test this solution:

1. Make a copy of ClientApp.accdb and place it in another location.
2. Open the original ClientApp.accdb and verify that you receive the message that says you are running version 1.0. Close ClientApp.accdb.
3. Open VersioningServer.accdb.
4. Open the USysAppHistory table in VersioningServer.accdb and add one record to the table. Make sure that the FilePath field points to the correct location of the copy of ClientApp.accdb.

ID	VersionNumber	VersionDescription	FilePath
2	1.1	Minor update	<PathTo>\UpdatedClientApp.accdb

5. Close VersioningServer.accdb.
6. Open the copy of ClientApp.accdb and change the AppVersion database property to 1.1 to match the entry in the USysAppHistory table. Close the copy of ClientApp.accdb.
7. Open the original ClientApp.accdb.

The frmCheckVersion form should open for a moment and then close. The updated version of the client application should open in a few seconds.

Re-Linking Tables Automatically

Many Access applications split the database — that is, the tables are located in one database while the application components (forms, reports, and code) are in another. The database where the tables reside is called the *back-end* database, and as you'd imagine, the other database is called the *front-end*. Tables in the front-end are linked to the back-end. To split the database in Access 2007, choose the Database Tools tab in the Ribbon and then click the Access Database button in the Move Data group. This opens the Database Splitter, as shown in Figure 14-7.

By default, the Database Splitter creates a new database with the text _be added to the name of your database.

When deploying a solution where the database is split, you may need code that dynamically refreshes the linked tables in the front-end database. This is to make sure that the database always has the latest connection information. To do this, create a new module and add the following routine:

```
Public Function EnsureTablesConnected() As Boolean  
    Dim db As DAO.Database  
    Dim td As DAO.TableDef  
    Dim stBackendDb As String
```

```

Dim stNewBackend As String
Dim stNotUpdated As String
Dim iUpdated As Integer      ' number of tables updated

' get the current database
Set db = CurrentDb()

```



Figure 14-7

The first thing we want to do is iterate through all the tables in the database so we'll begin a loop. We're only concerned with linked tables so we'll check the length of the `Connect` property of the `TableDef` object in DAO to determine if the table is linked.

```

' iterate through the tables and refresh links
For Each td In db.TableDefs
    If (Len(td.Connect) > 0) Then

```

The `Connect` property of the `TableDef` stores the connection string. For a table linked to another Access database, this is in the form: `;DATABASE=<PathToDatabase>`. Therefore, we'll want to remove the prefix from the connection string to get the path to the database itself.

```

' get the backend database for the table
stBackendDb = Mid(td.Connect, Len(";DATABASE=") + 1)

```

If this file doesn't exist, prompt for a new one. We'll implement the `GetFileName` called here in a moment.

```

' if the file doesn't exist, then prompt
If (stNewBackend = "" And Dir(stBackendDb) = "") Then
    stNewBackend = GetFileName()
End If

```

If the user selected a new back-end database (as specified in `stNewBackend`), then update the `Connect` property of the table:

```

If (Len(stNewBackend) > 0) Then
    ' refresh
    td.Connect = ";DATABASE=" & stNewBackend

```

Part IV: Finalizing the Application

Next, we need to refresh the linked table using the `RefreshLink` method. If this fails for any reason, we're appending the name of the offending table to a variable called `stNotUpdated`. We'll use this to display an error to the user later on.

```
' do some inline error handling in case something fails
On Error Resume Next

td.RefreshLink

If (Err <> 0) Then
    stNotUpdated = stNotUpdated & td.Name & vbCrLf
    Err.Clear
End If
On Error GoTo 0
```

We declared a counter variable to indicate how many tables were updated. Increment the counter and close the `If` blocks and `For` loop:

```
' update the counter
iUpdated = iUpdated + 1
End If
End If
Next
```

If any tables were not updated, we want to display them to the user; otherwise, display a message that the update succeeded. The `iUpdated` counter variable is used to display the succeeded message only when tables were actually updated.

```
' check for errors
If (Len(stNotUpdated) > 0) Then
    MsgBox "Tables not updated: " & vbCrLf & _
        Trim(stNotUpdated), vbExclamation, "Relink Tables Complete"
Else
    If (iUpdated > 0) Then
        MsgBox "Tables relinked successfully!", vbInformation, "Relink Tables
Complete"
    End If
End If
```

Last, do some cleanup and exit the function:

```
' cleanup
db.Close
Set td = Nothing
Set db = Nothing
End Function
```

The `GetFileName` function is used here to display the standard Open file dialog box to the user. This function is implemented as follows. You'll need a reference to the Microsoft Office 12.0 Object Library for this code to work.

```
Public Function GetFileName() As String
    With Application.FileDialog(msoFileDialogFilePicker)
```

```

    .Title = "Select backend database"
    .Show
    If (.SelectedItems.Count > 0) Then
        GetFileName = .SelectedItems(1)
    End If
End With
End Function

```

To test this, you need a database with some linked tables. To get the database to verify that the tables are linked when the application loads, call this routine using the `RunCode` action in a macro named `autoexec`.

The RunCode action will fail if the database is opened in disabled mode.

Programmatically Creating DSNs

If the data in your database comes from an ODBC data source such as SQL Server or Oracle, it might be useful to create Data Source Names (DSN) programmatically. There are a few different kinds of DSNs that you can use, depending on your needs:

- File DSN — As the name suggests, the connection information stored in the DSN is in a file.
- User DSN — Connection information is stored in the Registry for a specific user under an `HKEY_CURRENT_USER` hive.
- System DSN — Connection information is stored in the Registry and available for all users under an `HKEY_LOCAL_MACHINE` hive.

Creating a DSN-Less Connection

Because we tend to use user and system DSNs, we'll take a look at those. Before we do, however, let's actually look at how you can create a linked table without a DSN. Linked tables without a DSN are said to have a *DSN-less connection*. These types of connections are preferred from a deployment perspective because they don't require anything extra on the machine — only the ODBC driver.

Use the following code to create a DSN-less connection to SQL Server. Change the connection string to a valid ODBC connection string for other data sources.

```

Public Sub CreateDSNLessConnection()
    Dim stConnection As String

    ' DSN-less connection string for SQL Server
    stConnection = "ODBC;Driver={SQL Server};Server=<YourServerName>;" & _
                  "Database=<YourDatabase>;Trusted_Connection=Yes"

    ' connect
    Dim db As DAO.Database
    Dim td As DAO.TableDef

    Set db = CurrentDb()
    Set td = db.CreateTableDef("LinkedTable1", _
        dbAttachSavePWD, _
        "<YourTable>", _
        stConnection)

```

Part IV: Finalizing the Application

```
db.TableDefs.Append td

' cleanup
db.Close
Set td = Nothing
Set db = Nothing
End Sub
```

Creating a User DSN

The RegisterDatabase method in DAO is used to create a user DSN on the computer. Use the following code to create a user DSN:

```
Sub CreateUserDSN()
    Dim stAttributes As String
    ' Build the attributes string
    stAttributes = "Database=RibbonXSQL" & vbCrLf & _
        "Description=Access 2007 Expert Book DSN Test" & vbCrLf & _
        "OemToAnsi=No" & vbCrLf & _
        "Server=<YourSQLServer>" & vbCrLf & _
        "Trusted_Connection=Yes"
    ' Create the DSN
    DBEngine.RegisterDatabase "CreateUserDSN", "SQL Server", True, stAttributes
End Sub
```

Creating a System DSN

System DSNs are useful when you have multiple users who log on to a single computer and they all need access to a particular ODBC data source. There is no way to directly create a system DSN in Access so we'll fall back on an API function called `SQLConfigDataSource`. This function can actually be used to create both system and user DSNs.

Because DSNs contain several pieces of information such as the server name, database name, and credentials (just to name a few), we'll create a class module to simplify the creation process. Start by creating a new class module called DSN. The first thing we'll do then is to add an enumeration to the class called `DSNRequestType`. This `Enum` will be used to tell the API which action to take — whether to create a user or system DSN, remove a DSN, or configure an existing DSN.

```
Public Enum DSNRequestType
    ODBC_ADD_DSN = 1
    ODBC_CONFIG_DSN = 2
    ODBC_REMOVE_DSN = 3
    ODBC_ADD_SYS_DSN = 4
    ODBC_CONFIG_SYS_DSN = 5
    ODBC_REMOVE_SYS_DSN = 6
End Enum
```

Next, add the `Declare` statement for the API:

```
Private Declare Function SQLConfigDataSource Lib "odbccp32.dll" _
    (ByVal hwndParent As Long, _
    ByVal lRequest As DSNRequestType, _
    ByVal lpszDriver As String, _
    ByVal lpszAttributes As String) As Long
```

Public variables in a class module become properties of the class. For simplicity, we'll use public variables instead of Property routines. These variables will store the data for the DSN, but will not contain any validation.

```
Public DSN          As String
Public Driver       As String
Public Server       As String
Public Database     As String
Public Description  As String
Public User         As String
Public Password     As String
Public TrustedConnection As Boolean
Public Request      As DSNRequestType
Public ShowDialog   As Boolean
```

The class will contain one method, simply called `Create`. Start with the declaration of the method.

```
Public Function Create() As Boolean
    Dim stAttributes As String
    Dim rc As Long
    Dim hWin As Long
```

In much the same way that we built the attribute string in the previous section, we need to do the same thing here. Because we're passing this string to an API function, and we know from Chapter 2 that API functions written in C or C++ tend to use null-terminated strings, we need to append the null-termination character to our strings. In VBA, you do this using the `vbNullChar` constant.

```
' create the attribute string
stAttributes = "DSN=" & Me.DSN & vbNullChar & _
               "Server=" & Me.Server & vbNullChar & _
               "Database=" & Me.Database & vbNullChar & _
               "Description=" & Me.Description & vbNullChar
```

If the `TrustedConnection` property of the class is set, we want to add the `Trusted_Connection` attribute to the connection information; otherwise, we'll add the user name and password.

```
' credentials
If (Me.TrustedConnection) Then
    stAttributes = stAttributes & _
                  "Trusted_Connection=Yes" & vbNullChar
Else
    stAttributes = stAttributes & _
                  "UID=" & Me.User & vbNullChar & _
                  "PWD=" & Me.Password
End If
```

If no request is specified, we'll assume you're creating a DSN:

```
' default the request type
If (Me.Request = 0) Then Me.Request = ODBC_ADD_DSN
```

Part IV: Finalizing the Application

The SQLConfigDataSource function includes an argument that is a window handle. This argument is used as the parent window for the Create DSN dialog box from the ODBC Administrator tool included with Windows. In our class, we've created a property called ShowDialog to set the window handle. When the property is True, we'll use the Access window handle as the parent window for the dialog box.

```
' set the window handle
If (ShowDialog) Then
    hWin = hWndAccessApp()
Else
    hWin = 0
End If
```

Last, we need to call the API function and return:

```
' call the API
Create = SQLConfigDataSource(hWin, Me.Request, Me.Driver, stAttributes)
End Function
```

To test this function, create a new standard module and add the following test code. Update your information as needed.

```
Sub CreateSystemDSNTest()
    Dim objDSN As New DSN

    ' set properties
    objDSN.Driver = "SQL Server"
    objDSN.DSN = "CreateSystemDSNTest"
    objDSN.Description = "Expert Access 2007 Programming"
    objDSN.Server = "rob-vista\sqlexpress"
    objDSN.TrustedConnection = True
    objDSN.Database = "RibbonXSQL"
    objDSN.Request = ODBC_ADD_DSN
    objDSN.ShowDialog = True

    Debug.Print objDSN.Create()
    Set objDSN = Nothing
End Sub
```

Creating a system DSN requires administrator permissions on the computer because the DSN is stored in the HKEY_LOCAL_MACHINE hive of the Registry. On Windows Vista, you'll need to run the Access process as administrator in order to create a system DSN.

Ensuring an Application Runs Locally

For certain applications, you may want to specify a requirement that the application be run from the local machine. Such applications might include a split database where a copy of the front-end is available somewhere, but is also installed on the local machine. To prevent users from stumbling across the copy on the server, you might include some code in a startup form that detects where the database is being launched from.

Begin by creating a new form called frmCheckLocal (see Figure 14-8). Set the form properties according to your preferences. The form is used to check the path to the current database and display a message to the user if there is a problem. We've also added a simulated progress bar as described in Chapter 8, but this is optional.

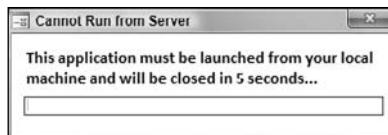


Figure 14-8

There are two checks we need to do to determine whether the file is opened locally. The first is a check to see whether the file is opened from a UNC path. A UNC path will begin with two backslash characters: \\. The other test we'll do is to determine whether the drive letter for the path is a local drive. To do this, we'll use the GetDriveType API function.

Add the following enumeration and declaration to the code behind the form.

```
Private Enum DriveType
    DRIVE_UNKNOWN = 0
    DRIVE_NO_ROOT_DIR = 1
    DRIVE_REMOVABLE = 2
    DRIVE_FIXED = 3
    DRIVE_REMOTE = 4
    DRIVE_CDROM = 5
    DRIVE_RAMDISK = 6
End Enum
Private Declare Function GetDriveType Lib "kernel32.dll" Alias "GetDriveTypeA" _
    (ByVal lpRootPathName As String) As DriveType
```

Next, add the following helper routine to determine whether the file is opened locally.

```
Private Function IsOpenedLocally() As Boolean
    Dim stDrive As String

    stDrive = Left(CurrentProject.FullName, 2)
    If (stDrive = "\\") Then
        IsOpenedLocally = False      ' Opened from UNC path
    ElseIf (GetDriveType(stDrive & "\\") <> DRIVE_FIXED) Then
        IsOpenedLocally = False
    Else
        IsOpenedLocally = True
    End If
End Function
```

Using the Load event of the form, call the IsOpenedLocally helper function. If this function returns False, we need to close the application. Because our form contains a label that displays a message to the

Part IV: Finalizing the Application

user, we're setting the TimerInterval property of the form to give the form a chance to display onscreen. If the database is opened locally then we'll open a form called frmStartup.

```
Private Sub Form_Load()
    If (Not IsOpenedLocally()) Then
        Me.TimerInterval = 5000
    Else
        ' // close the form
        DoCmd.Close acForm, Me.Name

        ' // Open the startup form
        DoCmd.OpenForm "frmStartup"
    End If
End Sub
```

Last, we need to define the Timer event that will be called by the form.

```
Private Sub Form_Timer()
    Application.Quit
End Sub
```

The code shown here exits the application. The code included with the samples for this book includes the custom progress bar code to increment the width of the rectangle using the Timer event. The name of the sample database that contains this code is OpenLocalOnly.accdb.

Summary

Minimizing deployment issues and automating processes where you can will help you in the long run. Investing in development work items such as a build process ensures that you are consistent and can eventually save you time and troubleshooting. In this chapter, you saw that:

- ❑ By using a consistent build process you can release applications more quickly and free up some time to focus on business problems.
- ❑ Attachment fields in Access 2007 make it possible to embed your dependencies in an application to ensure that references do not break.
- ❑ Functionality can be limited based on a number of factors to give users a trial experience of your application.
- ❑ By adding code to automatically keep an application up-to-date, you can deploy new functionality or fixes automatically.

The next chapter shows you how you can integrate help as a fit-and-finish feature for your applications.

15

Help and Documentation

Thorough user documentation and user assistance are two features required of any commercial application, and even small workgroup applications can benefit from help documentation. If you don't have time to create complete documentation for your database application, a simple help feature can provide lots of benefits to your user, and reduce the amount of effort you need to provide in support of your users.

In this chapter, we discuss several strategies for providing assistance to end users. We show you how to:

- Use intrinsic properties to control the status bar and control tips (tooltips) on forms and reports
- Create and edit custom properties using VBA code
- Create a context-sensitive help system using Access forms
- Create a help system using HTML pages
- Create HTML pages from data in an Access database
- Create a help system using compiled help files

We will begin by looking at some of the built-in features for communicating with users through the Access user interface.

Documentation via Built-in Properties

Access objects have several properties that help you communicate the intention of the data fields in your application, as well as provide suggestions for how to enter or interpret data in form and report controls.

Part IV: Finalizing the Application

In addition to being beneficial to your users, consistent use of these properties can help you, the developer, when creating your application. Consider how many times you have returned to a project that you were away from for a long period of time, and think about how much effort it was to re-familiarize yourself with the objects and logic in your application. As we have suggested throughout this book, commenting your code is invaluable and the first step in documenting your application for the developer. The properties we describe in this section provide the first level of documentation for the user.

Using the Description Property

All objects in Access have a `Description` property, which allows you to describe the purpose of the object. We recommend that you always supply a value for this property in order to make your database more self-documenting. Most documentation tools, including the Database Documenter built in to Access, rely on this field to describe the object.

You can supply a description for tables, queries, forms, reports, and macros. In addition, you can independently specify a `Description` for each field in a table or query. Adding a description for each field clarifies the intent of the field and makes your tables and queries clearer and self-documenting. It also has another side benefit: The text in the `Description` property is automatically copied to the `StatusBarText` property of controls on forms and reports. This provides an automatic hint to the user in the status bar whenever a control receives focus on a form or report.

It is possible to set up to 255 characters in the `Description` property of a table or query. You can set the `Description` property via code just as you can with any other user-defined property:

```
Set db = DAO.Database
Set td = db.TableDefs("Order Details")
td.Properties("Description") =
    "Product line item details for each order"
td.Fields("UnitPrice").Properties("Description") =
    "Price per unit, in US Dollars"
```

It may not be obvious why we can't just reference the `Description` property on the `TableDef` or `Field` object. If you look at these objects in Object Browser, you will not see a `Description` property listed. `Description` is an Access-defined property (as opposed to a DAO property). Only DAO properties can be accessed directly using dot notation. For example, `Name` is a DAO property, and so you can access this property using dot notation:

```
Dim td as DAO.TableDef
Set td = CurrentDb().TableDefs("Order Details")
Debug.Print td.Name                      ' Successful
Debug.Print td.Properties("Name")        ' Successful
```

`Description` is not a DAO property, and it can be referenced only through the `Properties` collection of the object. If you attempt to use dot notation with a non-DAO property, you will receive a compile error (or a runtime error 438 reporting "Object does not support this property or method").

```
Debug.Print td.Description           ' COMPILE ERROR or RUN-TIME ERROR 438
Debug.Print td.Properties("Description") ' Successful
```

You can set the `Description` property in the Access table designer by entering text in the `Description` column in the upper portion of the table designer.

Figure 15-1 shows where to add a Field description to a table field.

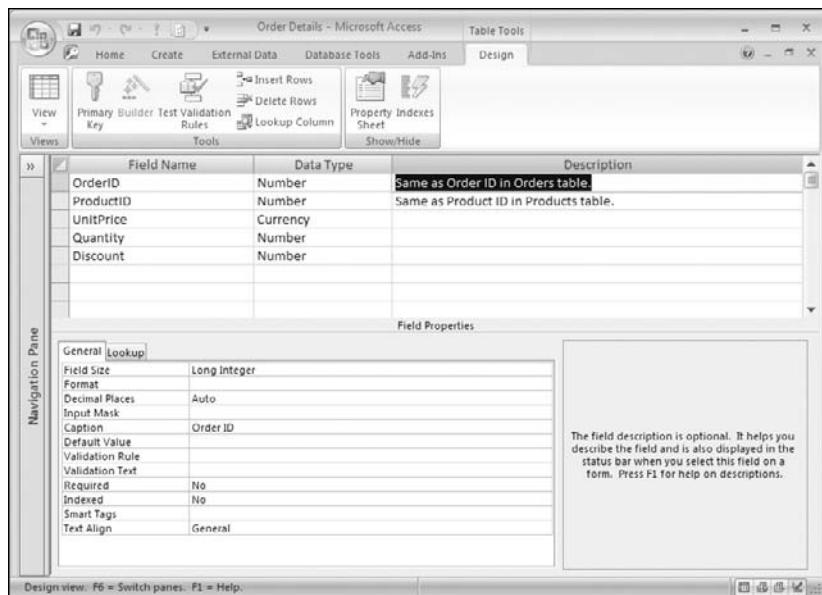


Figure 15-1

You can add a description to a query field by displaying the property sheet, and then setting focus into the field that you wish to describe. Figure 15-2 shows a Field description added to a query field.

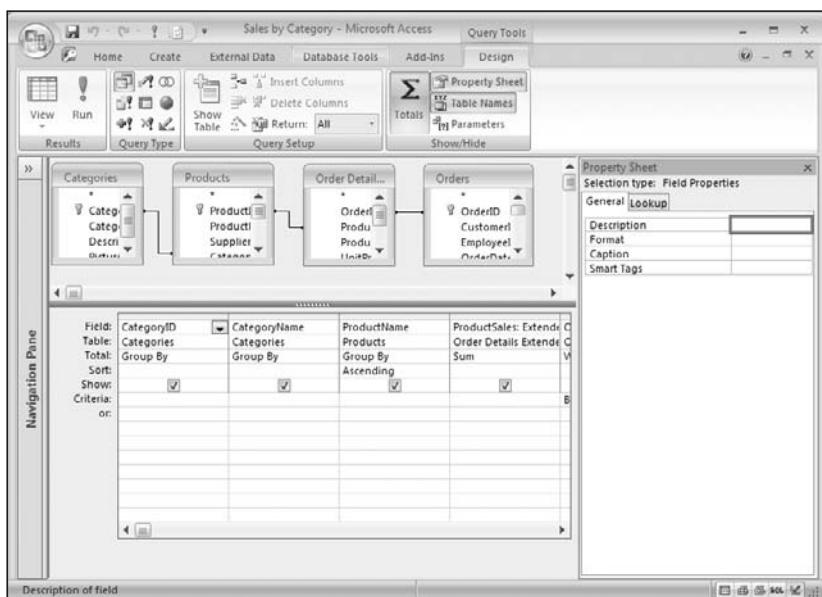


Figure 15-2

Part IV: Finalizing the Application

When adding a description to an object programmatically, you must create the `Description` property and append it to the `Properties` collection if it does not already exist. If you are working with a new database, the `Description` property will not already exist unless you used the Access UI to set the property values.

We use helper methods to set user-defined property values. The helpers attempt to set the property value normally, and trap for error 3270 (Property Not Found). In the error handler for error 3270, we create a new property object and append it to the `Properties` collection. Let's take a look at our helper methods for setting user-defined properties:

```
Private Sub SetObjectProperty(object As Variant, _
    prpName As String, _
    prpValue As Variant, _
    Optional prpType As DataTypeEnum = DataTypeEnum.dbText)
On Error GoTo Err_Handler
object.Properties(prpName) = prpValue
```

Our `SetObjectProperty` helper method accepts a variant argument. This argument can be any DAO object that supports a `Properties` collection (we will test this function with a `TableDef` and a `Field` object shortly). It also accepts a string argument that names the property we want to set, a variant argument that accepts the property value, and finally, an optional argument that determines the data type for the property. If we omit the data type argument, the property will be created as a string property. Notice that we have set an error handler, which we look at next:

```
Terminate:
On Error GoTo 0
Exit Sub
Err_Handler:
Select Case Err.Number
    Case 3270 ' Property not found
        ' Create the property in the collection
        On Error Resume Next
        object.Properties.Append _
            object.CreateProperty(prpName, prpType, prpValue)
    If Err.Number <> 0 Then
        MsgBox Err.Description, vbCritical, "Error creating property:" & _
            Err.Number
    End If
```

We include a standard terminate block that disables error handling and then terminates. We do not need a cleanup block because we do not allocate any objects within this function.

Our error handler specifically handles two error conditions. Error 3270 will be thrown if we attempt to set the value of a property that does not exist in the `Properties` collection. For example, when you create a new `TableDef` object in a database, there is no `Description` property in the `Properties` collection (because as noted earlier, `Description` is not a DAO property, and Access does not create a `Description` property until you set the `Description` value using the Access UI). If we attempt to set a value into the `Description` property, we will receive a 3270 error.

In our error handler for error number 3270, we simply create a new property, using the name, value, and data type passed into the function. We trap for an error creating the property by setting `On Error`

Resume Next and checking the value of the Err object immediately after we attempt to create the property.

```
Case 424, 438 ' Object Required, doesn't support this property or method
    MsgBox "the object passed does not support Property collections.", _
        vbCritical, "SetObjectProperty: InvalidObject"
```

We also include an error handler for error numbers 424 and 438. We will receive one of these errors if we attempt to set a property value on an object that does not have a Properties collection. For example, if we passed a string in the object argument, we will receive error 424 (because a string is not an Object in the VBA world). Likewise, if we pass a reference to the Access.Application object (which is an Object but does not have a Properties collection), we will receive error 438. In both these cases, this error handler will display a message reporting the error.

```
Case Else
    MsgBox Err.Description, vbCritical, Err.Number
End Select
Err.Clear
Resume Terminate

End Sub
```

Finally, we have a generic error handler to catch any unexpected problems (reporting the error in a message box). Following our normal pattern, we clear the error after the Select Case statement, and resume execution at the Terminate label.

Although this function provides flexibility by using Variant arguments, we prefer to use type-safe coding practices, and so we also create a type-safe wrapper method for each of the types we intend to set properties on. The following code shows the type-safe wrapper we use for TableDef objects:

```
Public Sub SetTableDefProperty(td As DAO.TableDef, _
    prpName As String, _
    prpValue As Variant, _
    Optional prpType As DataTypeEnum = DataTypeEnum.dbText)
    SetObjectProperty td, prpName, prpValue, prpType
End Sub
```

Note that this sub is declared `Public`, whereas the helper `SetObjectProperty` procedure was declared `Private`. We do this to discourage the use of the `SetObjectProperty` procedure. The type-safe public `SetTableDefProperty` procedure exposes a `TableDef` object argument instead of a Variant. Inside the procedure, we merely pass the arguments to the private `SetObjectProperty` procedure. Note that the signature of the type-safe wrapper procedure is identical to the signature of the helper procedure except for the variant argument. We use the same technique to create a type-safe wrapper for use with Field objects.

```
Public Sub SetFieldProperty(field As DAO.field, _
    prpName As String, _
    prpValue As Variant, _
    Optional prpType As DataTypeEnum = DataTypeEnum.dbText)

    SetObjectProperty field, prpName, prpValue, prpType

End Sub
```

Part IV: Finalizing the Application

When testing these methods, we want to be able to see what properties exist in the `Properties` collection and what their values are. The following simple method dumps the contents of the `Properties` collection to the Immediate window:

```
Public Sub DumpProperties(object As Variant)
    Dim prp As DAO.Property
    On Error Resume Next
    For Each prp In object.Properties
        Debug.Print prp.Name, prp.Type; prp.Value
        ' Certain property values cannot be referenced in
        ' design mode; in this case, report the name and
        ' property type
        If Err.Number <> 0 Then
            Debug.Print object.Name, prp.Name, _
                prp.Type, "<" & Err.Number & ":" & Err.Description & ">"
            Err.Clear
        End If
    Next prp
    Set prp = Nothing
End Sub
```

This function uses a `For...Each` loop to enumerate each `Property` in the `Properties` collection, writing the parent object name, and the property's Name, data Type, and Value.

Some property values cannot be accessed when an object is in design view (as it is when working with a `TableDef` object or a `Field` in the `Fields` collection of a `TableDef` or `QueryDef`), and one of several different errors will be thrown in this case. To ensure we can see each property in the collection, we set `On Error Resume Next`, and check for an error after each attempt to write the property data to the Immediate window. If an error has occurred, we write all the data again except this time we don't attempt to access the `Value` property of the `Property` object.

The following code demonstrates how to use our helper methods to set properties and dump properties to the Immediate window:

```
Public Sub TestSettingProperties()
    Dim db As DAO.Database
    Dim td As DAO.TableDef

    Set db = CurrentDb
    Set td = db.TableDefs("Order Details")

    ' Create a property on the table using the default data type (string)
    SetTableDefProperty td, "CustomProperty", "Custom Property Value"
    ' Create a long-integer property on the table
    SetTableDefProperty td, "CustomLongIntProperty", 1, dbLong
    ' Dump the properties of the TableDef's Properties Collection
    DumpProperties td
    ' Create a property on a field using the default data type
    SetFieldProperty td.Fields(0), "Field0Property", "String Value"
    ' Create a date-time property on a field
    SetFieldProperty td.Fields(0), "PropCreateDate", Date(), dbDate
    ' Dump the properties of the Field's Properties Collection
    DumpProperties td.Fields(0)
```

```
' Demonstrate error conditions in SetObjectProperty
' even though varItem is a Variant, it is set to a string, not
' not an Object, and so we will receive error 424
Dim varItem As Variant
varItem = "String ABC"
SetObjectProperty varItem, "AAA", "BBB" ' 424 error
' here we pass an Object to the function that does not
' support a Properties collection, and so we receive
' the 438 Object doesn't support this property error
SetObjectProperty Application, "FFF", "GGG"
```

End SubSetting the Status Bar Text

Access provides a couple of different ways to specify text to be displayed in the Access status bar.

All form and report controls contain a `StatusbarText` property. When focus is placed in a control, the contents of the `StatusbarText` property are displayed in the status bar. When focus is removed from the control, the status bar text is removed and replaced by Access's default status bar text. You can place a descriptive message to your user in the status bar, providing a brief description of what data you expect to be entered in the control, or an explanation of codes that are used in the field.

If a control is bound to a table or query field that has a value in the `Description` property, the `Description` value is added to the `StatusbarText` field by default. The default value can be replaced with a message of your choice, or the status bar message may be removed altogether by clearing the property.

Many developers use the `Description` property of a field to provide technical information (for example, describing foreign key relationships). While this is good information to include in developer documentation, you may not want to expose this information to your users in the status bar.

Any string up to 255 characters in length may be set into this property. If you prefer not to display any text in the status bar, you can remove the contents of the property. However, providing status bar hints to the user is a best practice that we strongly encourage you to implement in your applications.

Setting Status Bar Text Programmatically

The value of the status bar may be set using various `Application.SysCmd` constants. This can be very helpful when running lengthy operations in code. Set the Status Bar text to display the progress of your operation.

When programmatically controlling the status bar, set a message using the `acSysCmdsetStatus` constant. The following code will initialize the status bar and display a message:

```
SysCmd acSysCmdsetStatus, "Processing row 100"
```

When you wish to remove your message and let the status bar resume its default behavior, clear the status bar with the `acSysCmdClearStatus` constant:

```
SysCmd acSysCmdClearStatus
```

Part IV: Finalizing the Application

The SysCmd method cannot be used directly from the RunCode macro action. To set and clear status bar text from a macro, create two functions to wrap the SysCmd calls:

```
Public Function SetStatusBarText(msg As String)
    SysCmd acSysCmdsetStatus, msg
End Function
Public Function ClearStatusBarText()
    SysCmd acSysCmdClearStatus
End Function
```

Then, call the functions using the RunCode macro action. Set the FunctionName argument of the macro action to the function you want to call, passing any arguments in parentheses.

```
SetStatusBarText("text to display in status bar")
```

To call the ClearStatusBarText function, pass an empty argument list to the ClearStatusBarText function.

```
ClearStatusBarText()
```

Remember that when setting an expression in the FunctionName argument of the RunCode macro action, you do not begin the expression with an = (equal sign) character.

Using the Tag Property

Every object in Access contains a Tag property. The Tag property is a catch-all string object of up to 2,048 characters. This property is provided for your convenience: You may store any information in this field, and use it in any fashion you desire. Access never relies on this property, and values in the Tag property do not affect the operation of Access or the Access user interface. Think of it as a big shoebox attached to a control that can be used to store any information you want to store. You just drop the information into the box. It's up to you to know what information is in the box; Access neither knows nor cares what is in there. The item can be retrieved at any time. You can even store multiple items in this property if you come up with a convention (for example, a semicolon-delimited list of name-value pairs; Chapter 5 describes techniques for parsing individual values out of a delimited list).

When creating a custom help system, the Tag property can be used to store information to be displayed for the object. Later in this chapter, we will look at implementing a custom help solution, and we will use the Tag property in one scenario.

Setting the Control Tip

Form and report controls contain a ControlTipText property that contains the content that is displayed when the mouse is hovered over a control. (Control tips are also commonly referred to as ToolTips.)

You can set any string (up to 255 characters) into the ControlTip property. When this property contains a value, a ControlTip is rendered when you hover the mouse anywhere over the control. The tip appears a few moments after the mouse is placed over the control, and remains visible for a short time. A control tip is shown in Figure 15-3.

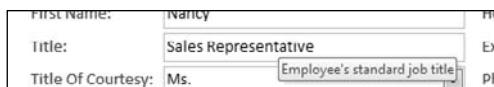


Figure 15-3

The Control Tip text for a field is completely independent of the Status Bar text. Frequently, the Status Bar will display a description of the contents of the field, while the ControlTip will display instructions to the user. There are no real standards for what kind of information goes into each field. The most important consideration is that you use consistency within your own application.

Windows Vista provides enhanced control tips that you can create programmatically, which allow headlines and collapsing sections; we discussed how to create these enhanced tips in Chapter 2. Access does not provide any intrinsic method to create these enhanced control tips.

Database Documenter

Access provides a built-in Database Documenter tool that generates a detailed report on the objects inside your database. The report includes the values of these self-documenting properties if they are present. The Documenter is available from the Analyze chunk of the Database Tools ribbon.

Although the Database documenter provides a thorough report on your database, it is not very flexible. The report cannot be saved as a report, and so you cannot customize the layout or design in any way. A limited set of options is available (click the Options button in the Document dialog box).

Providing Help to Users

In this section, we look at one way to provide a simple help system for users of your application. Although you can create a compiled help file and use the intrinsic HelpFile and HelpContextID properties to display the help file, authoring a compiled help file can be difficult and requires the installation of help compiler tools. A simple help system can be created with nothing more than a table to store your help information and a simple form to display the help content.

Storing Help Text in a Table

The first step in creating this simple help system is to create a table to store your help topics. This table can be as simple as the schema in the following table.

Field Name	Data Type	Description
idHelpTopic	AutoNumber (long integer)	
HelpText	Memo (rich-text)	Help text to display

Part IV: Finalizing the Application

You may wish to add additional elements depending upon the needs of your help system. For example, you may wish to assign topics based on control names in specific forms and/or reports, or provide a tooltip message in addition to the full help topic. The `HelpTopics` table used in the sample database for this chapter includes a `Title` field that describes the help topic, and a `ToolTip` field that stores up to 255 characters. The following data-definition query will quickly create a similar table:

```
CREATE TABLE HelpTopics
(
    idHelpTopic AUTOINCREMENT PRIMARY KEY,
    Title      Text(255),
    ToolTip    Text(255),
    HelpText   MEMO
)
```

If you set the `HelpText` field to a rich-text memo field, you can include text formatting in your topics, including bold and italic text, colors, highlights, customized fonts, and hyperlinks to online content. To support rich-text, open the table in design view, and set the `TextFormat` property to Rich Text. Use the formatting controls on the Home ribbon to apply formatting to the text in this field. To enter a hyperlink, simply type the URL of the link (for example, `http://mylink`) into the field.

Creating a Help Form

Once you have created the `HelpTopics` table, create a form to display the help content. You have complete control over the design of this form. You can make it as large or as small as you wish. Bind the form to the `HelpTopics` table (by setting the `RecordSource` property to point to the `HelpTopics` table or a query based on the `HelpTopics` table). Add a text box control to the form, bound to the `HelpText` field (or to the field containing the data you wish to display to your users). Consider some of the following features for the form:

- Set the `PopUp` property to true to have the form float above any other forms.
- Disable scroll bars.
- Disable the `Navigation Buttons` and `Record Selectors` properties.
- Set the border style to thin or dialog.
- Prefix the form name with `USys` to hide the help form in the `NavPane`.
- Add code to prevent the user from opening the form directly.

Add a public method to the form to allow code to choose which help topic to display. When the method is called, the form recordset is filtered to display only the requested help topic. If the help topic does not exist, the form will appear empty.

```
Public Sub SetHelpTopic(HelpTopicID As Integer)
    ' filter the help form to display the
    ' requested topic
    Me.Filter = "idHelpTopic = " & HelpTopicID
    Me.FilterOn = True
End Sub
```

The sample database for this chapter uses a form named `USysHelpForm` to display help topics. `USys` is a special prefix that indicates an object is a User System object. User System objects are not displayed in

the Navigation Pane unless the Show System Objects option is enabled. Because we used the `USys` prefix on the form name, the form is considered a User System form, and by default, it does not appear in the Navigation Pane. We could also hide the `HelpTopics` table in the Navigation Pane by renaming the table to `USysHelpTopics`. To show User System objects, check the Show System Objects option in the Navigation Options dialog box (right-click on the Navigation Pane header to display this dialog box).

Trapping the F1 Key

The `F1` key has evolved into the universally recognized Help key in Microsoft Windows applications. Users are conditioned to press this key to find assistance. Most applications recognize which control or element has focus and provide information specific to that control or element — context-specific help. By default, when you press `F1` anywhere in Access, the help file that ships with Access will be displayed, with a help topic related to the control or element that has focus. When you create your own help system, you will want to detect when the user has pressed the `F1` key and display your custom help topic.

When trapping specific keystrokes, you will want to set the `KeyPreview` property of the form or report to `Yes` (or `True` if you set the property programmatically). This allows the forms key events (`KeyPress`, `KeyDown`, `KeyUp`) to check the value of any key actions before they are passed to the individual form controls and to the Access application. If you do not set the form's `KeyPreview` property, you will need to add code to trap the `F1` key to the `KeyDown` property on each control you will supply help for. Setting the `KeyPreview` property is by far the easiest way to handle trapping the `F1` key. You can set this property in the property sheet, or via code (generally in the `Load` event handler).

Access displays the help file in response to the `KeyDown` event when the `F1` key is pressed. You can add the following code to a form's `KeyDown` event to suppress the display of the built-in help file.

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    ' Trap the F1
    If KeyCode = KeyCodeConstants.vbKeyF1 And Shift = 0 Then
        ' Do not pass the F1 key to the Application
        KeyCode = 0
    End If
End Sub
```

For a custom help system, rather than just suppressing display of the default help file, you will want to test whether or not you have a custom help topic available, and if so, display the custom help topic instead of the default help file.

In the sample database for this chapter, we have instrumented the `Issue List` form to display customized help. Custom help topics have been defined for each field on the form except for the `OpenedDate` and `OpenedBy` fields. The help topic ID is stored in the `HelpContextID` property for each field.

The following code exists in the `Form_KeyDown` event handler to check for the presence of a custom help topic for the field with focus. The code checks the `HelpContextID` property for the `Screen.ActiveControl` (the control with focus). If a custom help topic exists, `KeyCode` is set to zero to prevent the default Access help file from being displayed. Then an instance of the custom help form is created, `SetHelpTopic` is called on that form instance, and finally the help form is displayed.

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    ' Trap the F1 key
    If KeyCode = vbKeyF1 And Shift = 0 Then
```

Part IV: Finalizing the Application

```
' Check to see if a HelpContextID is defined for this field
Dim idHelpTopic As Long
idHelpTopic = Screen.ActiveControl.HelpContextId
If idHelpTopic > 0 Then
    ' A help topic exists, so suppress the default help file
    KeyCode = 0

    ' Create an instance of the help form
    Set hlpForm = New Form_USysHelpForm

    ' Set the topic for the help form
    hlpForm.SetHelpTopic CInt(idHelpTopic)
    hlpForm.Visible = True
End If
End If
End Sub
```

If you want to suppress the default help file regardless of whether you have defined a custom help topic, simply move the `KeyCode = 0` line outside of the `IF` statement. A custom help display appears in Figure 15-4.

Alternatives to a Floating Help Window

Because you are creating a help system from scratch, you have infinite flexibility for designing how you present help information to your users.

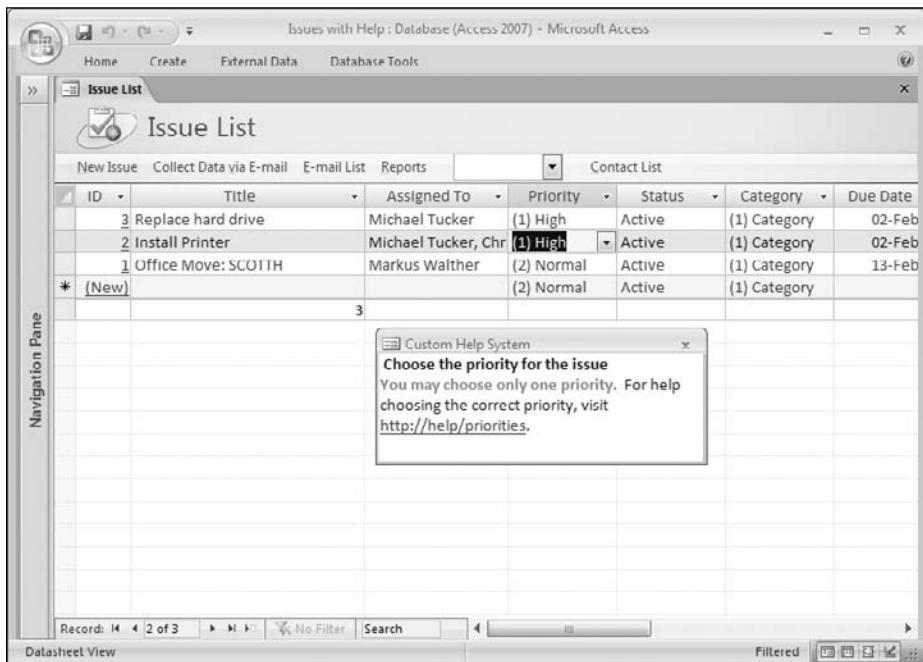


Figure 15-4

Instead of the traditional floating help form, you may prefer to have a help control built into your data entry form. Add a `SetHelpTopic` method to the data entry form to update the value of the text box with the help text, and modify the Form's `KeyDown` handler to call `Me.SetHelpTopic`.

Other alternatives include authoring a help pane or a custom help ribbon.

External Approaches

In this section, we will look at two approaches for creating a help system for your application that does not rely on an Access form to display the help content. The first approach uses Web pages created using HTML (or Hypertext Markup Language) and displayed in a Web browser. This technique builds on the F1 key trapping behavior we discussed in the previous section.

We will also look at one way to generate a compiled help file using the HTML Help Workshop, which is a free download from the Microsoft Web site.

Writing HTML Content for Help

The HTML-based help system displays your help content in Web pages created in HTML. Throughout this section, I assume you have some familiarity with HTML and know how to create pages. If you are new to this technology, there are many excellent references that can get you up and running quickly.

To get us up and running quickly to demonstrate the HTML help system, we are going to write a function to create a web page for each help topic in the `HelpTopics` table.

We will use a template to generate the web page. The template is merely a string, stored in a memo field, that has HTML tags and replacement tokens in the `{0}` format that the `StringFormat` function we discussed in chapter 5 expects. We will pass the template to `StringFormat`, along with data from the `HelpTopics` table.

We will need a table to store the templates. Run the following DDL SQL query to create the `HtmlTemplates` table:

```
CREATE TABLE HtmlTemplates
(
    ID AUTOINCREMENT PRIMARY KEY,
    HtmlTemplate memo
)
```

After you create the table, add the following HTML code to the `HtmlTemplate` field in a new row:

```
<html>
<head>
<title>Custom HTML Help - {0}</title>
</head>
<body>
<h1>{0}</h1>
<div><b>{1}</b></div>
<div>{2}</div>
</body>
</html>
```

Part IV: Finalizing the Application

Now that we have a template, we can write a small function named `CreateHtmlHelpFiles` that creates a Web page for each help topic in the `HelpTopics` table from the last section. This function creates one HTML Help page for each topic in the table.

```
Public Function CreateHtmlHelpFiles(HtmlTemplateID As Long)
    ' This function writes out one HTML page for each entry
    ' in the Help Topics table
    '
    ' The files are written to the same folder as the database
    ' using the following filename syntax
    ' help.<title>.html
    Dim template As String
    Dim fileName As String
    Dim hFile As Long

    Dim db As DAO.Database
    Dim rs As DAO.Recordset

    On Error GoTo Err_Handler
```

The function first retrieves an HTML template from the `HtmlTemplates` table, which is based on the `HtmlTemplateID` passed in as an argument to the function.

```
' Retrieve the template
template = DLookup("HtmlTemplate", "[HtmlTemplates]", "ID=" & HtmlTemplateID)
If vbNullString = Nz(template, vbNullString) Then
    MsgBox "No template found for HtmlTemplateID " & HtmlTemplateID
    Goto Terminate
End If
```

Next, we build a recordset of help topics. Here, we're selecting every topic in the table. Depending upon the structure of your `HelpTopics` table, you might want to filter the selection based on categories or other fields in your table. We enumerate the recordset, and generate a unique filename based on the location of the Access database (retrieved from the `CurrentProject.Path` property) and the title of the help topic and use several VBA file handling functions to create a text file.

```
' Create recordset with files to create
Set db = CurrentDb
Set rs = db.OpenRecordset("SELECT * FROM [HelpTopics]")

Do While Not rs.EOF
    ' For each row in the table, create a filename
    ' and use the StringFormat function to replace
    ' the template variables
    fileName = CurrentProject.Path & "\help." & rs("title") & ".html"

    ' Create the output file, write the content, and close
    hFile = FreeFile
    Open fileName For Output Access Write Shared As #hFile
```

To generate the HTML content, pass the template to the `StringFormat` function, along with fields from the recordset. Token `{0}` maps to the `Title` field, `{1}` maps to the `ToolTip` field that we have been using as the primary help comment, and finally `{2}` maps to the extended help text contained in the `HelpText`

field. The output from the `StringFormat` function is written to the text file, and then the file is closed and we move on to the next row in the recordset.

```
Print #hFile, StringFormat( _
    template, _
    rs("Title"), _
    rs("ToolTip"), _
    rs("HelpText"))
' Don't forget to iterate the recordset!
rs.MoveNext
Loop
```

Of course, our function contains the obligatory cleanup and error handling code:

```
Cleanup:
If Not rs Is Nothing Then rs.Close
Set rs = Nothing
Set db = Nothing

Terminate:
Exit Function

Err_Handler:
' Report error and terminate
MsgBox Err.Description, vbCritical, "Error encountered: " & Err.Number
GoTo Cleanup

End Function
```

You can extend this function in many ways, depending upon the content and layout of your templates and help pages, but this function provides a simple set of help pages that we can use to test our HTML page help system. The error handling in this function is minimal because it is designed to be used from the Immediate window or developer code, and will never be run by the customer. Obviously, if you design a system such that the customer might generate help content dynamically, you will want considerably more robust error handling, including among other things more granular reporting on file creation errors (access denied, invalid path, and so on).

Mapping HTML Files to Objects

There are a couple of different approaches you can take to map the HTML file to controls on your forms and reports.

The simplest method is to store the page name in the `Tag` property of the associated control. Because we are storing a page name string instead of a numeric index into the `HelpTopics` table, we cannot use the `HelpContextID` property (which accepts numeric values only). The `Tag` property provides a convenient generic property that you can use to store this value. Your code can query the `Tag` property for the page name, and then launch the browser. We look at how to launch the browser in the next section.

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    ' Trap the F1 key
    If KeyCode = vbKeyF1 And Shift = 0 Then
```

Part IV: Finalizing the Application

```
Dim helpPage As String
' Check to see if a page name is defined in this field's tag property
helpPage = Screen.ActiveControl.Tag
If vbNullString <> Nz(helpPage, vbNullString) Then
    ' A help topic exists, so suppress the default Access help file
    KeyCode = 0
    ' open the help topic in the browser
    LaunchHtmlPage helpPage
End If
End If
End Sub
```

Opening the Browser

All of the functionality for opening the browser is isolated in the `LaunchHtmlPage` function. Call this function by passing the name of the page that you wish to display in the browser. The function calculates the fully qualified location of the HTML page (in our case, by using the `BuildPath` helper function to combine the folder where the database is located and the page name we passed in).

To prepare for the `LaunchHtmlPage` procedure, we need to declare the `ShellExecute` API function, as follows:

```
Public Declare Function ShellExecute Lib "shell32.dll" Alias "ShellExecuteA" ( _
    ByVal hwnd As Long, _
    ByVal lpOperation As String, _
    ByVal lpFile As String, _
    ByVal lpParameters As String, _
    ByVal lpDirectory As String, _
    ByVal nShowCmd As Long) As Long
Private Const SW_SHOWDEFAULT As Long = 1
Private Const SE_ERR_NOASSOC As Long = 31
Private Const ERR_FILE_NOT_FOUND As Long = 2
Public Enum PathStyle
    Unc
    Uri
End Enum
```

The `PathStyle` enumeration is used by the `BuildPath` function to indicate which style of path we are constructing: a UNC path that requires backslash delimiters (\), or a URL path that requires forward slash delimiters (/).

The `BuildPath` helper function accepts three arguments: two strings that will be combined into a single path string, and an optional style enumeration. By default, this function builds UNC style paths.

```
Public Function BuildPath(ByVal item1 As String, ByVal item2 As String, Optional
    style As PathStyle = PathStyle.Unc)
    Dim conjunction As String

    ' determine which conjunction we want
    Select Case style
        Case PathStyle.Unc
            conjunction = "\"
        Case PathStyle.Uri
            conjunction = "/"
    End Select
    item1 = item1 & conjunction & item2
    ' remove any extra backslashes
    item1 = Replace(item1, "\\", "\\")
    ' remove any extra forward slashes
    item1 = Replace(item1, "\", "/")
    ' remove any leading or trailing slashes
    item1 = Trim(item1)
    ' remove any double slashes
    item1 = Replace(item1, "\\\"", "\\")
End Function
```

```
Case PathStyle.Uri
    conjunction = "/"
End Select
```

Based on the `PathStyle` argument, we store the path conjunction string:

```
' cleanup item1
' trim leading and trailing white space
' ensure item1 contains a trailing conjunction
item1 = Trim$(item1)
If conjunction <> Right$(item1, Len(conjunction)) Then
    item1 = item1 & conjunction
End If
```

We clean up the first path string by trimming off white space, and adding a conjunction to the end of the string if one does not already exist:

```
' cleanup item2
' trim leading and trailing white space
' ensure item2 does not contain a leading conjunction,
' since this has been applied to item1
item2 = Trim$(item2)
If conjunction = Left$(item2, Len(conjunction)) Then
    item2 = Mid$(item2, Len(conjunction) + 1)
End If
```

The second path string is also trimmed. If this string begins with a path conjunction, we remove it because one already exists at the end of the first path string.

```
BuildPath = item1 & item2

End Function
```

Finally, we concatenate the two path strings and return.

The `LaunchHtmlPage` procedure accepts a single argument: a fully qualified reference to a Web page to display. This argument may be either a UNC filename (i.e., `c:\myapp\page.html` or `\server\share\page.html`) or a URL reference to a page on an intranet or the Internet (i.e., `http://intranetserver/page.html` or `http://www.tempuri.org/page.html`).

```
Public Sub LaunchHtmlPage(pageName As String)
    Const MB_TITLE As String = "Error opening Help page"

    Dim fquri As String ' fully-qualified URL
    Dim lngReturnValue As Long
    Dim sError As String
```

After setting up some variables, we build a fully qualified path name to the help page using the `BuildPath` function, and then we call the `ShellExecute` API function. This function resolves the document passed in the third argument and attempts to open the document using the application registered

Part IV: Finalizing the Application

for that document type. In the case of our HTML files, `ShellExecute` will launch the help page in the currently registered browser.

```
' we are using ShellExecute to launch the browser;
' this requires that our pageName have an extension
' or protocol that is associated with the registered browser
fquri = BuildPath(CurrentProject.Path, pageName, PathStyle.Unc)
lngReturnValue = ShellExecute(0, "open", fquri, vbNullString, vbNullString, 0)
```

Finally, we include some error handling code. When `ShellExecute` successfully launches a document in an application, it returns a value between 0 and 32. So here, we trap for a return value of > 32. We provide specific error messages for the most common error cases: a bad path to the document, and a document that cannot be resolved to an application.

```
' ShellExecute returns a value > 32 on success.
If 33 > lngReturnValue Then
    Select Case lngReturnValue
        Case SE_ERR_NOASSOC
            sError = "The extension of the help " & _
                      "file is not associated with " & _
                      "any application: " & fquri
        Case ERR_FILE_NOT_FOUND
            sError = "The file was not found: " & fquri
        Case Else
            sError = "Unexpected error returned " & _
                      "by ShellExecute: " & lngReturnValue
    End Select

    ' Display the error message
    MsgBox sError, vbCritical, MB_TITLE
End If
End Sub
```

This sample HTML system uses help pages created in the same folder as the database. However, there is no reason you couldn't place the pages in another folder on the local machine, or even on a Web server. If you are hosting your Web pages on a Web server (either on the Internet or on a private Web server within your organization), you can update this function to build a path using `PathStyle.Uri` and the root location of your Web server instead of the database folder.

```
Const WebServerRoot As String = _
"http://myserver/myapplication"

Shell BuildPath(WebServerRoot, pageName, PathStyle.Uri)
```

As long as you use the `.html` extension for your pages (or specify either `http://` or `https://` protocol when attempting to launch help pages based on a Web server), you should have no problems resolving your help pages to the registered Web browser.

The `LaunchHtmlPage` function uses the browser registered with the system as the default browser. If you have particular browser requirements, you can modify this function to call a specific browser. You will need to provide error handling in the case that the specific browser you require is not installed on the system.

Creating Compiled HTML Help Files

Now that we've looked at two ways of creating a home-brew help system, let's look at how you can create a help system that uses the standard Compiled HTML Help files that Windows uses.

You must use an external tool to create compiled HTML Help files. Many applications are on the market that allow you to create compiled HTMLHelp files. We will be looking at Microsoft's HTML Help Workshop, which is a free download that provides a toolset to take HTML files and compile them into a CHM file. Other products that are available allow you to create Help topics in Microsoft Word or the editor of your choice.

To download the HTML Help Workshop, visit <http://download.microsoft.com> and search on "HTML Help Workshop." Visit this book's Web site at www.wrox.com/174029 for links to several commercial products that create HTML content. You can also download the sample files used in this section to create a compiled help file.

Creating Content for HTML Help Workshop

The HTML Help Workshop uses two types of files to provide help content: HTML pages and a topics file. HTML pages are used for help topics that you can browse to — for example when you open the Help File and use the Contents tab to browse to a specific help topic. We can reuse the HTML pages we created in the previous section for these browsable pages.

We will add a new Title Page, named `help.TitlePage.html`, which displays information that users may want to see when they open the help file manually (in this case, the name of this book and the authors). This page is the first page that is displayed if the user doesn't request context-sensitive help. Figure 15-5 shows the browsing of a compiled help file.



Figure 15-5

Part IV: Finalizing the Application

A topics file and a header file are used for context-sensitive help. Context-sensitive help topics (as defined by the HTML Help compiler) are displayed in a small pop-up window instead of in the help file display window. The header file defines a set of constants (known as IDH constants) that we will use within the Access application to identify the correct popup. This file is named `HelpSystem.h` in the sample files accompanying this chapter.

```
#define IDH_ISSUES_TITLE      1000
#define IDH_ISSUES_ASSIGNEDTO 1001
#define IDH_ISSUES_CATEGORY   1002
#define IDH_ISSUES_DUEDATE    1003
#define IDH_ISSUES_PRIORITY   1004
#define IDH_ISSUES_STATUS     1005
```

The topics file is a text file with a unique style that defines the text that will appear in the popup. Each topic in the file is defined by a header, which begins with a period, followed by the keyword `topic`, followed by the IDH identifier from the header file. The help text appears immediately below the header row. In this case, I copied and pasted the content from the HTML files into this topics file. This file is named `HelpSystem.Popup.txt` in the sample files accompanying this chapter.

```
.topic IDH_ISSUES_TITLE
Describe the issue. Enter a descriptive title for the Issue.
.topic IDH_ISSUES_ASSIGNEDTO
Choose contact that this issue should be assigned to. You can choose as many
contacts as you wish.
.topic IDH_ISSUES_CATEGORY
Choose the category for the issue. You may choose only one category. &nbsp;For
help choosing a category, visit <a
href="http://help/category">http://help/category</a>.
.topic IDH_ISSUES_DUEDATE
Enter the due date for the issue. This is an optional field. &nbsp;If there is no
required due date, leave this field empty.
.topic IDH_ISSUES_PRIORITY
Choose the priority for the issue. You may choose only one priority. For help
choosing the correct priority, visit <a
href="http://help/priorities">http://help/priorities</a>.
.topic IDH_ISSUES_STATUS
Choose the status for the issue. You may choose only one status. For help choosing
a status, visit <a href="http://help/status">http://help/status</a>.
The first step in creating a compiled HTML Help file is to generate your browsable
help topics. We can reuse the HTML files we generated in the previous section for
this content.
```

Once you have these files created, you can launch the HTML Help Workshop environment and create a new project (you can also open the `HelpSystem.hhp` file from the accompanying files folder for Chapter 15).

Add Files to Your Project and Set Project Properties

The New Project wizard allows you to quickly add your content files to the project. The wizard will ask you to name your project, choose whether or not you want to include a table of contents, index, and existing HTML files. Leave the table of contents and index buttons unselected for now (you will be able to create a table of contents from within the application).

The HTML Help Workshop is shown in Figure 15-6.

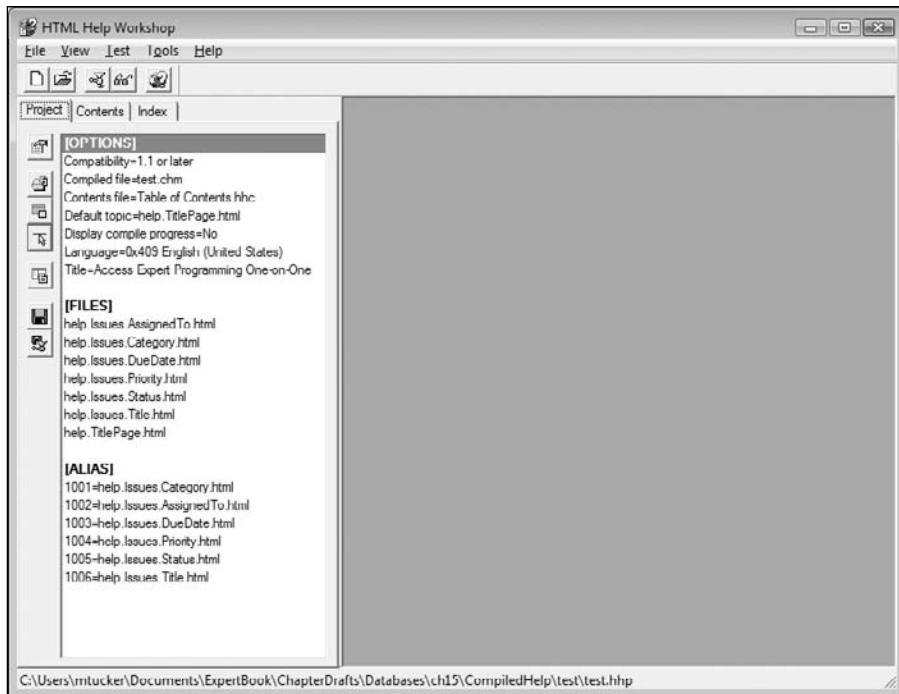


Figure 15-6

Choose the Project tab, and then click the Add/Remove Topic Files button to add your HTML files. This application doesn't have the best user interface. You may have to hover over each button on the project tab to identify the Add/Remove Topic Files button, and there is no menu item to perform this task. (Consider how difficult this application is to use when you design the user interface for your applications!)

Include all of the HTML files for your browsable topics. You should also include an HTML file with data to display on the "Title Page" of your help file. This will be the first page that is displayed when you open the help file manually.

While still on the Project tab, click the Change Project Options button. Set the title of the help file (in our sample, "Access Expert Programming One-on-One"). In the Default File drop-down, choose the Title Page file (help.TitlePage.html). This sets the page that is first displayed to the user.

Add Context Mapping

On the Project tab, click the HtmlHelp API button. This will display the HtmlHelp API information dialog. Choose the Alias tab, and then click the Add button to define an Alias for each topic that you want to be displayed when you press F1 on your form. Figure 15-7 shows the dialog box for adding context ID aliases.

Part IV: Finalizing the Application

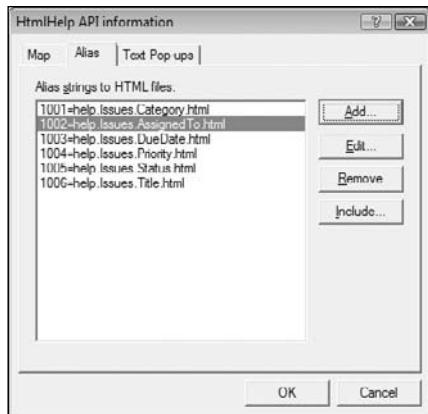


Figure 15-7

Click the Add button to display the Alias dialog box. Here, you will enter the HelpContextID value that you will assign to each control. On our sample form, I assigned the values listed in the table that follows.

Control Name	HelpContextID
Category	1001
AssignedTo	1002
DueDate	1003
Priority	1004
Status	1005
Title	1006

For each control that you want to assign context-sensitive help, add an alias. In the first text box on the Alias dialog box, enter the HelpContextID value. In the HTML file drop-down, choose the file that contains the topic you wish to display when the user presses F1 on your form. The comment is optional. Figure 15-8 shows the editing of alias details.

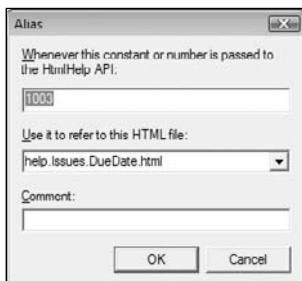


Figure 15-8

Repeat this until you have created an alias for each control.

Build the Table of Contents

Once you have added your HTML content, choose the Contents tab, and build the table of contents. Click on the Insert a Heading button (again, you'll have to hover, or you can right-click in the contents pane and choose Insert Heading) to display the Table of Contents Entry dialog box. Specify the text you want to appear in heading in the Entry Title text box. In our sample database, this is the name of the book, "Access 2007 Expert Programming."

Now for the tedious part! For each and every topic, click on the Insert a Page button. Enter the text you want to appear in the table of contents in the Entry Text text box (for example, "AssignedTo"). Then, click the Add button. You will be able to select from any HTML content page you added in the previous step, or you can browse to any file and add by clicking the browse button. Choose the correct file ("Custom HTML Help — Issues.AssignedTo") and click OK. Repeat this until all of your topics have been added to the help file.

Compile the Help File

Now that all your information has been entered, it's time to compile the help file. From the File menu (imagine that!), choose Compile. If all goes well, in a few seconds, you will see a log file that reports no errors or warnings. If you see this, you now have a compiled help file! Check the folder for a .chm file with the same name as your project.

Adding HelpContextID Values to Your Controls

Now that you have a compiled help file, you need to add `HelpContextID` values to your form controls. Simply open up your form in design view, display the properties sheet, and enter the `HelpContextID` value you chose for the control (in the case of our sample, enter the appropriate value from the table in the preceding section).

Finally, you need to define which help file will be called. Set the form or report's `HelpFile` property to point to the .chm you generated in the HTML Help Workshop.

When you define the `HelpFile` property on a form or report, you do not need to trap for the F1 key. Access will automatically forward the `HelpContextID` from the currently active focused control to the help file you defined in the property. If you are modifying a form or report that has a `KeyDown` handler to trap the F1 key, you need to remove the handler (or at least avoid setting `KeyCode` to zero in order to pass through the F1).

Summary

In this chapter, you looked at a variety of techniques that can be used to provide documentation and assistance to users of an application. You learned how to:

- Set intrinsic properties to control the status bar
- Set control tips (tooltips)
- Create and update custom properties using VBA code

Part IV: Finalizing the Application

- Create a context-sensitive help system using Access forms
- Create a help system using HTML pages
- Create HTML pages from data in an Access database
- Create a help system using compiled help files

Providing a context-sensitive help system is a highly visible sign of a well-designed and well-documented application. Well written and easily accessible Help topics not only improve the usability of an application, they reduce support costs by allowing users to help themselves. We hope you can use some of these techniques to increase user satisfaction with your applications.

A

Programming Tips and Tricks

This appendix contains some useful tips and tricks that you can use in your applications. Many of these code snippets are ones we have used over the years to help standardize our code or represent specific problems that we were trying to solve. Several of the techniques were born out of *refactoring* efforts to help make code more elegant. Refactoring is the process of making code more readable or simplified without changing the end result.

This appendix also covers such topics as running code dynamically and working with objects and dates. The appendix ends with some cool tools that you can add to your applications to make the process of writing code easier.

Dynamically Running Code

In many cases, you may want to run code dynamically — in other words, where the name of the routine that you want to execute is not known until runtime. For example, in Chapter 8 we used the `Eval` function to pass the name of a routine that was stored in a table for custom navigation in the application. VBA and the Access object model provides three mechanisms to do this: `CallByName`, `Eval`, and `Application.Run`.

CallByName

The `CallByName` function is defined by VBA and is used to invoke a property or method on an object instance. It can also be used to set a property value for an object instance. The parameters for the `CallByName` function are listed in the table that follows.

Programming Tips and Tricks

Parameter Name	Parameter Type	Description
Object	Object	An instance of an object whose property or method you will invoke.
ProcName	String	The name of the property or method you will invoke.
CallType	VbCallType	Specifies the type of property or method.
Args()	Variant	ParamArray of arguments that are passed to the property or method.

Keep in mind a couple of things when working with this function. First, the name of the property or method you call must exist for the object instance passed to the function. Second, the number of arguments passed to a property or method must match the arguments expected by the code being called. Because code executed using the CallByName function is invoked using late-binding, you won't see a compile error for either of these issues. Instead, you'll receive a runtime error, which may be unexpected by your users.

With that in mind, let's look at some examples of using the CallByName function to work with properties and methods.

Retrieve a Property Value

Say that you have a class in your application named CommonFolders that encapsulates Windows API functions and the Environ function in VBA to provide an easy way to retrieve system folders. This class has a property named Favorites that returns the path to the Favorites folder for the current user, and a property named System that returns the path to the system directory on the computer. The code in the class is as follows:

```
Public Property Get Favorites() As String
    Favorites = Environ$("USERPROFILE") & "\Favorites"
End Property
Public Property Get System() As String
    System = Environ$("SYSTEMROOT") & "\system32"
End Property
```

Using CallByName, you can retrieve the values from these properties as follows. We use an array to store the names of the properties to retrieve and then loop through the array to invoke the properties, as shown in the following code.

```
Sub CallPropertyGetByName()
    Dim strFolder      As String
    Dim strFolders(1)  As String
    Dim intCounter    As Integer
    Dim objFolders    As CommonFolders

    ' Create a new instance of the class
    Set objFolders = New CommonFolders
    ' simulate external data using the array
    strFolders(0) = "Favorites"
```

```
strFolders(1) = "System"

' retrieve a property value from the instance
' of CommonFolders using VbGet
For intCounter = 0 To UBound(strFolders)
    strFolder = CallByName(objFolders, strFolders(intCounter), VbGet)
    Debug.Print strFolder
Next

' cleanup
Set objFolders = Nothing
Erase strFolders
End Sub
```

Notice that we passed `objFolders` to `CallByName` as the object instance. This is our instance of the `CommonFolders` class. We can also create a new instance of an object in the `CallByName` function directly as you see in a few moments. `VbGet` instructs the `CallByName` method to invoke a `Property Get` procedure. In this case, we assign the return value of the specific `Property Get` procedure to a variable named `strFolder`.

Set a Property Value

Because `CallByName` accepts any instance of an object, it can also be used to change properties of Access objects, such as a form or control. The following example illustrates how you can change multiple properties of several open forms using `CallByName`.

For this example, start with two simple forms: `frmCallByName1` and `frmCallByName2`. One of the properties that we update is the `Moveable` property of the form. Because objects appear in tabbed windows in Access 2007 by default, set the `Popup` property of both forms to Yes.

Next, start with the following declarations in a new module. For test purposes, we store the properties we want to change and their values in an array. You could also use a table to store this data, which would serve as a configuration table, as described in Chapter 12.

```
Sub CallPropertySetByName()
    Dim frm As Form
    Dim avarProps(2) As Variant
    Dim intCounter As Integer
```

Next, fill the array of properties. By setting these properties to `False` later on, we create a standard user interface to the two forms created earlier.

```
' fill the array of properties to set:
avarProps(0) = "Moveable"
avarProps(1) = "RecordSelectors"
avarProps(2) = "NavigationButtons"
```

So far so good. Now, open the two forms we created earlier for testing. This is to ensure that the forms are open when you call the `CallByName` function.

```
' open some forms for testing
DoCmd.OpenForm "frmCallByName1"
DoCmd.OpenForm "frmCallByName2"
```

Programming Tips and Tricks

Now for the good part — dynamically calling the properties. The first task is to loop through all of the open forms in the `Forms` collection. We'll also loop through the array of properties (`avarProps`).

```
' loop through forms
For Each frm In Forms
    ' set form properties to False using the array
    For intCounter = LBound(avarProps) To UBound(avarProps)
```

The next step is to call the `CallByName` function on the Form instance (`frm`). In this case, we specify `VbLet` to indicate that we want to invoke the `Property Let` routine for the form instance. Specify `False` as the last argument to the `CallByName` function as the value for the property. Because the last argument to `CallByName` is a `ParamArray`, you can pass a variable number of arguments to the function depending on the arguments defined by the property or method that is being invoked. We also close out the loops and do some cleanup.

```
    CallByName frm, avarProps(intCounter), VbLet, False
    Next
Next

    ' cleanup
    Erase avarProps
End Sub
```

When you run this code, the two forms should open and their properties should have been set to the values specified in the array.

Call a Method

Last, we use `CallByName` to invoke a method. As you might imagine, we specify the `VbMethod` argument to `CallByName` to indicate that we want to call a method. In the `CommonFolders` class, we define a method called `Browse` that opens a folder using the `Shell` function. The method accepts one argument to indicate the folder to open. The `Browse` method is defined as follows:

```
Public Sub Browse(strFolder As String)
    ' browses the folder by shelling to explorer.exe
    Dim strCmd As String

    ' create the command-line and run it using Shell
    strCmd = "explorer.exe " & Chr(34) & strFolder & Chr(34)
    Shell strCmd, vbNormalFocus
End Sub
```

In this case, because we invoke a single method, we create a new instance of the `CommonFolders` class in the `CallByName` function using the `New` keyword, as shown:

```
Sub CallMethodByNameWithArgument()
    ' invoke the CommonFolders.Browse method
    CallByName New CommonFolders, "Browse", VbMethod, CurrentProject.Path
End Sub
```

After you run this code, the directory where the database is located should be open.

Application.Run

Unlike `CallByName`, the `Run` method on the Access `Application` object is used to run a routine in a standard module. The `Run` method accepts up to 30 optional arguments that are passed to the procedure in question. Use this method when you want to execute a routine, which may or may not accept arguments, or to retrieve a value from a procedure, which may or may not accept arguments. We frequently use this method to call a routine where the name of the routine is stored in a table.

If you're working with managed code, you should be aware that Visual C# does not support the concept of optional arguments. As such you must pass `Type.Missing` (by reference) for any arguments you do not want to provide when using C# to automate an instance of Access, as shown in the following code:

```
// store Type.Missing
object missing = Type.Missing;
// run some code in Access
string result = app.Run("RunMethodTest", ref missing, ref missing,
                        ref missing, ref missing, ref missing, ref missing);
```

Eval

The `Access Application` object defines the `Eval` function, which is used to evaluate a string as an expression. These expressions are passed to the Access database engine expression service. Because the argument passed to `Eval` results in an expression being evaluated, you can pass in something as simple as:

```
Eval("1 + 1")
```

to something more complex, such as the name of a routine to execute. In that regard, the `Eval` function is similar to the `Application.Run` method but does not accept arguments. We also use this method frequently when the name of a routine to execute is stored in a table.

Object Helper Functions

How many places in code do you create new instances of objects or destroy objects? Probably quite a few. You can abstract this code into helper functions that handle this work for you. Why, you ask. Creating or destroying objects is easy enough — why add the overhead of a function call to the equation?

The answer to these questions comes back to the issue of refactoring. By maintaining functions to handle this work, code can be simplified and even reduced depending on the complexity of the helper function.

Code quality is also an issue. With centralized functions you may be more likely to remember to clean up after yourself when working with objects. In addition, a centralized object creation function can be used throughout your applications to ensure that objects are always in the state you expect them to be. These helper functions are also useful for global initialization or termination routines that execute when your application launches.

Programming Tips and Tricks

Object Initialization

In order to work with an object it must be instantiated using the `New` keyword. Typical usage might resemble the following:

```
Dim objMyObject As MyType  
Set objMyObject = New MyType
```

You can abstract this into a separate routine as follows:

```
Public Sub InitObject(objToSet As Object, newObject As Object)  
    If (objToSet Is Nothing) Then  
        Set objToSet = newObject  
    End If  
End Sub
```

Using the `Settings` class that was created in Chapter 12, you may have a global object variable to store an instance of the class. In this case, the usage for `InitObject` becomes the following:

```
InitObject gobjSettings, New Settings
```

An object initialization routine is particularly useful for `Recordset` objects. Again, think about how many times you have written something such as the following:

```
Dim rs As DAO.Recordset  
Set rs = CurrentDb().OpenRecordset("tblMyTable")
```

You can simplify this greatly using a helper routine, defined as follows:

```
Public Sub InitDAORecordset(objRS As DAO.Recordset2, strName As String, _  
    Optional ExistingRecordset As DAO.Recordset2)  
  
    ' set the object  
    If (objRS Is Nothing) Then  
        ' create a copy of the recordset or a new recordset  
        If (Not (ExistingRecordset Is Nothing)) Then  
            Set objRS = ExistingRecordset.Clone()  
        Else  
            Set objRS = CurrentDb.OpenRecordset(strName)  
        End If  
    End If  
End Sub
```

This routine accepts the following three arguments:

- ❑ `objRS` — The DAO recordset object to initialize.
- ❑ `strName` — The recordset to open. This is the name of the table, query, or SQL statement passed to the `OpenRecordset` method.
- ❑ `ExistingRecordset` — An `Optional` argument that specifies an existing recordset to copy using the `Clone` method.

To call the `InitDAORecordset` routine, use something like this:

```
Dim rst As DAO.Recordset2
InitDAORecordset rst, "SELECT * FROM tblMyTable"
```

This is also useful for ADO recordset objects. You need a reference to the ActiveX Data Objects library for the following code to work:

```
Public Sub InitADORecordset(objRS As ADODB.Recordset, strSource As String, _
    Optional ExistingRecordset As ADODB.Recordset)
    ' set the object
    If (objRS Is Nothing) Then
        If (Not (ExistingRecordset Is Nothing)) Then
            Set objRS = ExistingRecordset.Clone()
        Else
            Set objRS = New ADODB.Recordset
        End If
    End If
    If (objRS.State <> adStateOpen) Then
        ' open the recordset
        objRS.Open strSource, CurrentProject.Connection, _
            adOpenDynamic, adLockOptimistic
    End If
End Sub
```

Object Termination

Initialization routines tend to be more complex than termination routines, but that doesn't mean that a termination routine doesn't have value. By using a termination routine, you can add additional checks or code to ensure that objects haven't already been destroyed. For a simple object variable, you might simply set it to `Nothing` as follows:

```
Public Sub DestroyObject(obj As Object)
    Set obj = Nothing
End Sub
```

For recordset objects, you can close the recordset before you destroy it:

```
Public Sub DestroyDAORecordset(rs As DAO.Recordset2)
    If (Not (rs Is Nothing)) Then
        rs.Close
        DestroyObject rs
    End If
End Sub
```

and its ADO equivalent:

```
Public Sub DestroyADORecordset(rs As ADODB.Recordset)
    If (Not (rs Is Nothing)) Then
        If (rs.State <> adStateClosed) Then
            rs.Close
        End If
    End Sub
```

Programming Tips and Tricks

```
End If  
DestroyObject rs  
End Sub
```

Global Object Properties

Earlier we mentioned that you might have a global object variable named `gobjSettings` to store an instance of the `Settings` class that was created in Chapter 12. However, if `gobjSettings` goes out of scope or is somehow destroyed you receive runtime error 91-Object variable or With block variable not set. This subsequently requires error handling to trap for this error in any place this object is used. Wouldn't it be nice to avoid this problem altogether? You can by using a property routine.

Instead of a public `gobjSettings` variable, make it private and wrap it in a `Property Get` procedure as follows:

```
Private mobjSettings As Settings  
Public Property Get SettingsObject() As Settings  
    ' make sure the instance is valid  
    InitObject mobjSettings, New Settings  
    ' return the object instance  
    Set SettingsObject = mobjSettings  
End Property
```

Now, instead of using `gobjSettings` to call the `GetOptionValue` or `SetOptionValue` methods, use the property as follows:

```
Public Sub InitSettings()  
    SettingsObject.SetOptionValue "Setting1", "Value1"  
    SettingsObject.SetOptionValue "Setting2", "Value2"  
    SettingsObject.SetOptionValue "Setting3", "Value3"  
End Sub
```

This has the added benefit of ensuring that the object instance is never out of scope.

You could make this a Function routine but we prefer the syntax and feeling of a Property routine.

Date Data Type Tricks

Internally, VBA stores variables of the `Date` data type as a `Double`. To retrieve the underlying value for a date, use the `CDb1` conversion function in VBA:

```
CDb1 (#2/15/2007#)
```

The number to the left of the decimal point represents the number of days since December 30, 1899. The number to the right of the decimal point represents the time as the percentage of a 24-hour day. If you have a date value such as 2/15/2007 12:00 PM, its underlying value is 39128.5, where 12:00 noon is one-half of a 24-hour day or 0.5.

The following examples take advantage of the fact that dates are just numbers.

Looping Through Dates

Because the underlying data type for a Date is a Double you can loop through them like other numbers. Let's say that you are designing a form that resembles a daily planner. The form should display the hours of the day down the side, as shown in Figure A-1.

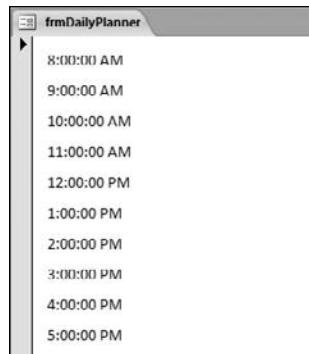


Figure A-1

Rather than hard-code the label captions, we fill them dynamically by looping through the hours of the day. This provides the flexibility for the user to define her working hours in the application. For our purposes, we assume the working hours are from 8:00 a.m. to 5:00 p.m. and we hard code them.

Start by creating a new form called frmDailyPlanner. On this form, add ten labels. To make it easier to set the captions for the label, name them sequentially as follows: lblHour1, lblHour2, lblHour3, and so on. Next, add the following code to the Load event of the form:

```

Private Sub Form_Load()
    ' locals
    Dim dteStart As Date
    Dim dteEnd As Date
    Dim dteLoop As Date
    Dim dblHour As Double
    Dim intCounter As Integer

    ' how long is one hour: 1:00 AM - 12:00 AM
    dblHour = CDbl(#1:00:00 AM#)

    ' get the start and end time
    dteStart = #8:00:00 AM#
    dteEnd = #5:00:00 PM#

    ' loop from the start time to the end time
    ' with an increment of one hour
    For dteLoop = dteStart To dteEnd Step dblHour
        intCounter = intCounter + 1
        Me.Controls("lblHour" & intCounter).Caption = dteLoop
    Next
End Sub

```

Programming Tips and Tricks

This code starts by calculating the time value of one hour. Because we know that the time portion of a date is a `Double`, the value of one hour is represented by 1:00 a.m. We convert this to a `Double`. Next, we hard-code the start time as 8:00 a.m. and the end time as 5:00 p.m. Last, we need to loop from the start time to the end time with an increment of one hour using the `Step` keyword. Inside this loop we increment a counter that determines which label caption to set on the form. The `Caption` property of the label is then set to our loop variable: `dteLoop`. When you run the code, you should have a form that looks similar to the one in Figure A-1.

Arrays of Dates

Using a similar technique, filling an array of dates for later use is pretty easy. If you store a month's worth of dates in an array, it's even easier — at that point the day of the month can be used as the index in the array. The following code shows how you can fill an array of dates for a given month. It uses the `DateDiff` function to determine the number of days in the month. The `Day` function is used to specify the index of the array.

```
Sub FillArrayOfDates()
    Dim aDates() As Date
    Dim dteStart As Date
    Dim dteEnd As Date
    Dim dteLoop As Date
    Dim i As Integer

    ' define start and end date
    dteStart = #3/1/2007#
    dteEnd = #3/31/2007#

    ' resize the array from 1 to the number of days in the month
    ReDim aDates(1 To DateDiff("d", dteStart, dteEnd) + 1)

    ' fill the array
    For dteLoop = dteStart To dteEnd
        aDates(Day(dteLoop)) = dteLoop
    Next

    ' print the values in the array
    For i = LBound(aDates) To UBound(aDates)
        Debug.Print aDates(i)
    Next
End Sub
```

Determining Dates Dynamically

We worked on an application one time where we had to determine the *N*th day of the month for tracking occasions and holidays. For example, the second Tuesday of the month, or the fourth Thursday of the month for a given year. The solution for this was the following function:

```
Public Function NthDayOfMonth(N As Integer, iDayOfWeek As VbDayOfWeek, iMonth As
Integer, Optional iYear) As Date
    Dim iDay As Integer
    Dim iDay1 As Integer
```

```

Dim iOffset As Integer

' verify data
Debug.Assert (N > 0 And N < 7)
Debug.Assert (iDayOfWeek >= vbSunday And iDayOfWeek <= vbSaturday)
Debug.Assert (iMonth >= 1 And iMonth <= 12)

' default to the current year
If IsMissing(iYear) Then iYear = Year(Date)

' calculate the offset (number of days to add for N weeks)
iOffset = (N - 1) * 7

' calculate the first day of the month for the specified day of the week
' such as the 1st Saturday of the month
iDay1 = 8 - Weekday(DateSerial(iYear, iMonth, 1), (iDayOfWeek + 1) Mod 8)

' the actual day is the 1st day + the offset
iDay = iDay1 + iOffset

' return
NthDayOfMonth = DateSerial(iYear, iMonth, iDay)
End Function

```

Miscellaneous Tips

Here are a few tips and tricks that we use frequently when working with code. The first tip is a matter of style, whereas the others are more specific implementations that we find useful.

Categorizing Constant Values

You've probably noticed that enumerated values in the Access object model tend to begin with the same prefix. For example, take the enumeration of `RunCommand` options:

- acCmdCloseAll
- acCmdCloseDatabase
- acCmdCloseWindow

These examples illustrate grouping by using the same prefix: `acCmd` where "ac" stands for Access itself, and "Cmd" stands for Command. You can use the same technique in your applications to help organize constants and find their values easier. This also works with non-enumerated constants, such as programmatic IDs.

Consider the following code to categorize a set of programmatic IDs for automation:

```

Public Const PROGID_DL As String = "Outlook.Application"
Public Const PROGID_XL As String = "Excel.Application"
Public Const PROGID_WD As String = "Word.Application"

```

Comparing Class Instances

The `Object` class in the .NET Framework provides two Shared (static in C#) methods that are used to test for equality: `Equals` and `ReferenceEquals`. The `Equals` method accepts two object instances and determines whether they are equal based on their data type. The `ReferenceEquals` method determines whether two instances of an object are the same instance. While VBA does not support Shared or static members like Visual Basic.NET or C#, you can implement these methods for your classes.

Equals

The `Equals` method is used to determine equality between two instances of the same class. Without support for Shared members, we have to define this method in each class where it is used. This method is used to compare members of the class to determine whether they are equal.

Let's suppose, for example, you have a class named `Person`. This class has a property called `SSN`, defined as follows:

```
Private m_ssn As String
Public Property Get SSN() As String
    SSN = m_ssn
End Property
Public Property Let SSN(SSN As String)
    m_ssn = SSN
End Property
```

In this class, if the social security number (SSN) of two instances of the class is the same, then the objects are equal. The `Equals` method for the class to compare the `SSN` property appears as follows:

```
Public Function Equals(obj As Person) As Boolean
    Equals = (Me.SSN = obj.SSN)
End Function
```

As you can see, you can use data within the class instance to determine equality. To use this method, call it from an instance of the `Person` class:

```
Sub EqualsTest()
    Dim p1 As New Person
    Dim p2 As New Person

    ' Assign the SSN properties
    p1.SSN = "000-00-0001"
    p2.SSN = "000-00-0002"

    Debug.Print p1.Equals(p2)
End Sub
```

You can use this method to compare multiple properties of a class for more complex objects. By using the `Equals` method you can avoid complex comparison statements throughout your code.

ReferenceEquals

Similar to the `Equals` method, `ReferenceEquals` is used to determine equality. However, this method compares two objects to determine whether they are the same instance — in other words, it does not

perform a comparison of members of an object. To determine whether two instances of an object are equal, use the hidden `ObjPtr` function defined in VBA as follows:

```
Public Function ReferenceEquals(obj As Person) As Boolean
    ReferenceEquals = (ObjPtr(Me) = ObjPtr(obj))
End Function
```

The `ObjPtr` function returns the address of an object variable in VBA. If two instances of a class point to the same address in memory those instances are the same.

Take a look at the following test:

```
Sub ReferenceEqualsTest()
    Dim p1 As New Person
    Dim p2 As New Person
    Dim p3 As Person

    ' copy p1 to p3
    Set p3 = p1

    ' test
    Debug.Print p1.ReferenceEquals(p2)
    Debug.Print p1.ReferenceEquals(p3)
End Sub
```

In this test routine, we've created three instances of the `Person` class. The instances defined as `p1` and `p2` were declared `As New`, whereas the instance defined as `p3` was created from a copy of `p1`. If you run this code, you see that `p1` is not the same instance as `p2`, but is the same instance as `p3`.

Add Number of Retries to a Code Block

When there is a chance that a piece of code may fail, error handling is usually available to catch any errors that may be displayed. You can take this a step further by wrapping such code in what we like to refer to as a *retry loop*. A retry block is a loop that wraps around the code in question to run the block a set number of times or retries. The counter for the number of retries can be incremented either in the loop itself or in an error handler that looks for specific errors.

In this example, we try to connect to a SQL Server using an `ADODB.Connection` object but the server does not exist. Because error cases such as this are known to us ahead of time, we increment the retry counter in an error handler. Because we're using ADO in this example, you need to add a reference to one of the ActiveX Data Object libraries for this code to work.

```
Sub ConnectionRetries(iRetries As Integer)
    ' attempt to connect to a server a specified number of times
    Dim cn      As New ADODB.Connection
    Dim iRetry As Integer

    On Error GoTo ConnectionErrors

    ' start the retry loop
    While (iRetry < iRetries)
        Debug.Print "Retry: " & iRetry
        ' code to connect to the server
        ' ...
        iRetry = iRetry + 1
    End While
    ' code to handle connection failure
    ' ...
    ' resume execution
    Resume
ConnectionErrors:
    ' code to handle connection failure
    ' ...
    ' resume execution
    Resume
End Sub
```

Programming Tips and Tricks

```
' connect to the server
cn.Open "Provider=SQLOLEDB.1;Data Source=InvalidServer;" & _
        "Initial Catalog=MyDatabaseName;Integrated Security=SSPI"
Wend

' exit now
If (cn.State = adStateClosed) Then
    Set cn = Nothing
    MsgBox "Failed to connect to server"
    Exit Sub
End If

' do the rest of the work
' ...

ConnectionErrors:
If (Err = &H80004005) Then
    ' increment the retry counter
    iRetry = iRetry + 1
    Resume Next
Else
    Stop
End If
End Sub
```

For the sake of example, the number of retries was passed to the routine. This could be a global variable defined in the application or a configuration setting for the functionality of connecting to a server.

Add a Timeout to a Code Block

Using the number of retries you can execute code for a specific number of attempts, but what if you want to wait for a certain amount of time? For this you can add a timeout loop to a code block using the `Timer` function in VBA.

Consider the connection example we used a moment ago, which has been modified to use timeout after 60 seconds. In this case there is no counter to increment, but rather we need to check the `Timer` function each time in our loop. This may result in code that runs slightly over 60 seconds if the code running inside the loop waits for some time. Use the following code to implement a timer based timeout.

```
Sub ConnectionTimeout()
    Const TIMEOUT As Long = 60

    ' attempt to connect to a server for a set amount of time
    Dim cn      As New ADODB.Connection
    Dim sngTimer As Single

    On Error GoTo ConnectionErrors

    ' start the timer
    sngTimer = Timer()
```

```

' start the retry loop
Do
    ' connect to the server
    cn.Open "Provider=SQLOLEDB.1;Data Source=InvalidServer;Initial
Catalog=MyDatabaseName;Integrated Security=SSPI"
    DoEvents
Loop While ((Timer() - sngTimer) < TIMEOUT)

' exit now
If (cn.State = adStateClosed) Then
    Set cn = Nothing
    MsgBox "Failed to connect to server"
    Exit Sub
End If

' do the rest of the work
' ...

ConnectionErrors:
If (Err = &H80004005) Then
    ' server not found
    Resume Next
Else
    Stop
End If
End Sub

```

Tools for the Immediate Window

Before we discovered VBA (in Excel 5.0 using the macro recorder), we wrote ad hoc programs and reports in languages such as SAS, QuikJob, and FOCUS on an IBM OS/390. So we admit it, we're keyboard guys. We tend to stay away from the mouse whenever possible. (We even used shortcut keys to apply formatting to the content in this book.) With that in mind, we sometimes use these routines during development to help the process along and help perpetuate what simply appears to be a fear of the mouse.

Some of these were also inspired by working in Visual Studio 2005. If you type a greater than (>) symbol in the Immediate window in Visual Studio 2005, you are presented with a list of commands that are available in the IDE, as shown in Figure A-2.

This is a pretty cool feature that we were trying to replicate in the Visual Basic Editor (VBE).

These routines are called from the Immediate window in the VBE and saved in a standard module named with a single character: c (for commands). The single character makes it easy to call while still providing IntelliSense. The standard module is included in the database during development, and then removed before deployment as a part of our build process.

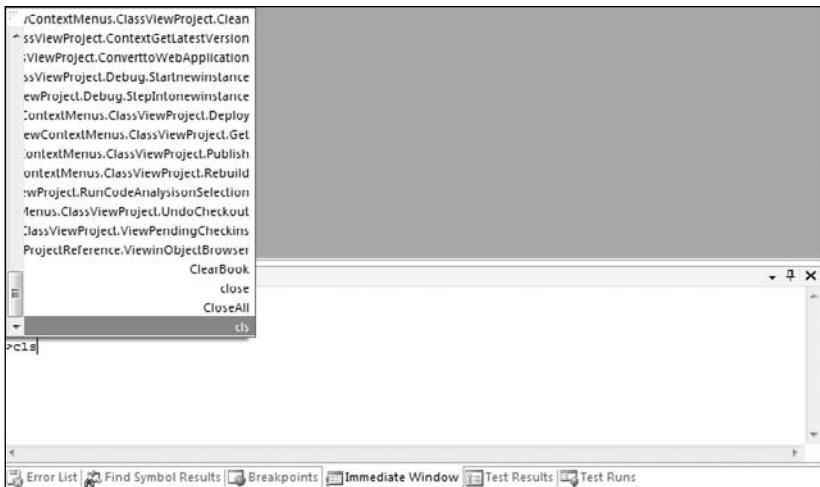


Figure A-2

Clearing the Immediate Window

Often times we want to clear the Immediate window but we're too lazy to press **Ctrl+G**, then **Shift+Ctrl+Home**, followed by **Delete**. So, we wrote this routine to do it for us:

```
Sub cls()
    SendKeys "({HOME})"
    SendKeys "{DEL}"
End Sub
```

This routine uses the `SendKeys` function in VBA, which has some side-effects such as toggling the NumLocks key, but this works for us for the time being.

Displaying the Watch Window

This is a simple wrapper around the `Windows` collection in the `VBE` object. This collection is defined in the Visual Basic for Applications Extensibility 5.3 object model: `VBE6EXT.OLB`.

```
Sub w()
    VBE.Windows("Watches").Visible = Not VBE.Windows("Watches").Visible
End Sub
```

Displaying the Locals Window

Same functionality as the Watch window but this time for the Locals window:

```
Sub l()
    VBE.Windows("Locals").Visible = Not VBE.Windows("Locals").Visible
End Sub
```

Making a Backup of Objects

Some time ago, we wrote an add-in that wrapped the `SaveAsText` method. The `SaveAsText` method is a hidden method in the `Access Application` object that persists the definition of the specified object to a text file. This add-in called the method for each object in a collection such as `AllForms` or `AllReports`. This can also be useful during development, so here is a basic implementation of this code for the utility module.

Start by declaring the `Sub` procedure. This one accepts an optional argument called `lType` that defaults to `AcObjectType.acDefault`. More about that in a moment. The name of this routine is `b` (for backup).

```
Sub b(Optional lType As AcObjectType = AcObjectType.acDefault)
    Dim ao      As AccessObject
    Dim col     As AllObjects
    Dim sPath   As String
    Dim sObjType As String
```

In this example, the objects are saved to a subdirectory of the folder where the database resides called `backup`. Add the following code to create the directory if it does not exist:

```
' create the directory
If (Dir(CurrentProject.Path & "\backup", vbDirectory) = "") Then
    MkDir CurrentProject.Path & "\backup"
End If
```

Next, we need to get the appropriate `AllObjects` collection based on the value of `lType`. The `AllObjects` collection is the base class for the `AllForms`, `AllReports`, and the other related collections. Remember from Chapter 3 that a base class defines the properties and methods for a derived class.

```
' get the appropriate AllObjects collection
Select Case lType
    Case AcObjectType.acForm
        Set col = CurrentProject.AllForms
    Case AcObjectType.acMacro
        Set col = CurrentProject.AllMacros
    Case AcObjectType.acModule
        Set col = CurrentProject.AllModules
    Case AcObjectType.acQuery
        Set col = CurrentData.AllQueries
    Case AcObjectType.acReport
        Set col = CurrentProject.AllReports
    Case AcObjectType.acDefault
        ' call this routine for each supported type
        c.b acForm
        c.b acMacro
        c.b acModule
        c.b acQuery
        c.b acReport
End Select
```

Before we save each object, notice that when `lType` is `acDefault` we handle things a little differently. In this case, we recursively call the backup routine for one of the specialized types. This enables you to save a backup of each object type by calling this routine without any parameter.

Programming Tips and Tricks

Okay, now it's time to save each object. Each object in an `AllObjects` collection is of the type `AccessObject`. The following code enumerates through the collection object (`col`) and saves the object:

```
' save all objects
If (Not (col Is Nothing)) Then
    For Each ao In col
        ' build the path
        sPath = CurrentProject.Path & "\backup\" & ao.Name & ".txt"

        ' save the object
        SaveAsText ao.Type, ao.Name, sPath
    Next

    ' get the type of the object for display
    sObjType = Choose(lType + 1, "Tables", "Queries", _
                      "Forms", "Reports", "Macros", "Modules")
    Debug.Print "Save complete for type: " & sObjType
End If
End Sub
```

You've probably also noticed that we cheated a little and used the `Choose` function instead of another `Select...Case` statement.

After you run this routine, you should have a group of text files in a subdirectory called `backup` of the current directory. These text files represent the object definition for an object in the database.

Closing All Code Windows

By now you realize that when we ask a leading question we're going to turn around and answer it. So, wouldn't it be nice if you could close all open code windows with one action? Using the Visual Basic for Applications Extensibility library mentioned earlier, you can do just that. The following routine is named `ca` (for close all) and enumerates the `Windows` collection that was used earlier:

```
Sub ca()
    ' close all code windows
    Dim w As VBIDE.Window

    ' close all code windows
    For Each w In VBE.Windows
        If (w.Type = vbext_wt_CodeWindow) Then
            w.Close
        Else
            Debug.Print w.Type, w.Caption
        End If
    Next
End Sub
```

Figure A-3 shows the IntelliSense when you type a c in the Immediate window of the VBE.



Figure A-3



Query Performance

Many different factors determine how efficient and performant your queries will be. Simply reordering elements of your query can make a world of difference. Other times, changes to your database schema will provide a more efficient design that the database engine can use to decrease the time it takes to run your queries. Even simple periodic maintenance can speed things up.

In this appendix, we discuss several issues and techniques you can use to design tables and queries that will make it easy to create efficient queries.

The Query Optimizer

Both the Access database engine and Jet database engine contain a Query Optimizer that analyzes a set of statistics and determines the optimal query execution strategy. Every time you save a query, the SQL is automatically run through the Optimizer, which determines the optimal way to run a query.

The Query Optimizer is a cost-based analyzer. The Optimizer examines the SQL and identifies a list of actions that can be performed when the query is executed. An estimated cost is generated for each action. The Optimizer identifies the least expensive set of actions that will generate the required results. This set of actions is known as the *query execution strategy* or *query plan*. The database engine stores the execution strategy along with the query so that when you run the query, the execution plan is already determined and you save the time of computing the optimal execution plan. Queries that have a stored execution plan are said to be *compiled queries*.

Among other numbers, the Optimizer looks at how many records are in a table, how many data pages are used by each table, and where the data pages are located within the database file. Indexes are an important input to the Optimizer, which will choose a different strategy depending on whether an index exists, and on what type of index is available.

Query Performance

You cannot directly control the optimization process. However, there are certain things you can do when creating your tables and queries to ensure that the Optimizer can identify an optimal execution strategy.

Designing Tables for Performance

Even though this appendix describes how to increase query performance, you can't achieve maximum query performance without properly designed and properly tuned tables.

Use the Smallest Appropriate Data Type

When creating fields in your tables, always choose the smallest appropriate data type that will contain your data. Don't use a long integer if your field allows you to choose only numbers from 1 to 10 (use a byte instead). When creating text fields, limit the number of characters when possible. Sometimes, you'll need the maximum 255 characters, but usually data will fit in much smaller fields, and smaller fields lead to smaller indexes. Smaller fields often mean faster data entry, too!

Indexes

Many factors contribute to a performant query, but none more important than indexes. Just as an index to a book enables you to zoom immediately to the pages that have the content you want to read, indexes in a database enable the database engine to quickly locate the data you need.

Add a Primary Key Index to the primary key in every table. Tables in properly normalized databases always have a set of fields that perform the function of a primary key. Adding a primary key index ensures that the Optimizer can determine relationships and build the most efficient joins possible.

Use unique indexes whenever you index a field that contains non-repeating values. In many cases, a unique index is not appropriate and in these cases, it is necessary to create a non-unique index; however, unique indexes provide the maximum performance benefit. Note that primary key indexes are unique indexes by definition.

Indexes take space, and it takes time to update the indexes, so you probably do not want to index every field in every table. If you have an index on every field, each index must be updated every time you add or remove a row from the table. The queries in your database are the best guide to which fields need to be indexed. The following section describes situations in which you should create indexes.

Use Numeric Values for Primary Key Fields

Numeric values are generally faster than string values in indexes. Even if you have string data that uniquely identifies a record (say a company name field or first, middle, and last name fields), for performance reasons, it makes sense to add a numeric identifier as a primary key field. This reduces the amount of storage required in foreign tables, which reduces the size (and thus increases the speed) of indexes.

In some cases, a text primary key that contains more human readable values may make sense for your application, but you should restrict these uses to smaller tables or tables that are not joined frequently to other tables.

Favor numeric data types for primary key fields.

Index Criteria, Sort, and Join Fields

When you filter data in a query using a `WHERE` clause, performance will suffer dramatically if you add a criteria to a field that is not indexed. If you don't have an index, the database engine must look at each and every record to determine whether it should be included in your results. With the index, the engine is able to directly select the correct record.

When you are joining two tables, the same issue exists. Without an index on the join fields in each table, the engine must search through every record in both tables in the join to find the matching records. If you have an index on only one of the fields, the engine will still need to search through every record in the other table. Having an index on both fields in the join will dramatically increase performance, especially when querying against large datasets. When creating one-to-many joins, ensure the join field(s) on the one side are in either a primary key or a unique index.

Fields that exist in your `ORDER BY` clause should have an index. Indexes by their very nature exist in a sorted order, so if there is an index on each field in your `ORDER BY` clause, the database engine already knows what order the rows need to be in. If you haven't created an index, the database engine essentially creates one for each field in your `ORDER BY` clause each and every time you run the query.

Ensure every field in your WHERE and ORDER BY clauses is indexed. Ensure both fields in a join specification are indexed, and that the index on the one-side is a unique or primary key index.

Single Field vs. Multi-Field Indexes

Databases that use the Access database engine are able to efficiently use multiple single-field indexes. Compound indexes can be very efficient, and save a little space compared to multiple single-field indexes on the same fields. However, compound indexes are less flexible. If you need to sort or select on the first field in a compound index, the index will be used efficiently. But if you want to sort or select on subsequent fields in the index, the index cannot be used, and so the database engine will fall back to the default table scan behavior where it looks at every row in the record source.

If you have multiple single-field indexes, the engine can combine these indexes effectively regardless of which field you are trying to search or sort on.

Consider multiple single-field indexes instead of compound indexes in your application.

Designing Queries for Performance

When you're creating queries for your applications, here are some steps you can take to help improve their performance.

Reduce the Amount of Data

The less data there is to process, the faster it can be processed. That's a simple and intuitive rule of thumb that leads to the next rule for improving performance: Reduce the amount of data that needs to be processed by restricting data wherever possible and as early as possible.

Query Performance

Reduce the number of columns that are returned by your query by selecting only those columns you actually need in your query. Avoid using the * (asterisk) wildcard to select every field in the table if you don't actually need to return each field. Prune the field list to the smallest number of items that will allow you to solve your problem.

Often you will want to filter your query based on a field that you don't actually need to use in your results. When you add a field to a query to use in a WHERE clause, determine whether or not you need to access this field in your results list. If not, clear the Show box in the query designer, or make sure you do not add the field to your field list.

Reduce the number of rows that are returned by your query by adding criteria to the WHERE clause to remove rows that do not apply to your problem.

When working with nested queries, reduce the number of rows as much as possible in the inner-most queries that are run first; by reducing the number of rows returned by the inner-most queries, there is less data to be retrieved by the outer layers.

When working with grouped data, filter as much as possible in the WHERE clause. Only restrictions that require aggregated data should appear in the GROUP BY clause; all other restrictions should be moved to the WHERE clause. WHERE clause restrictions are applied before data is grouped, reducing the quantity of data that needs to be grouped.

Use Between . . . And, IN Operators

When you are comparing data, using the BETWEEN . . . AND operator is more efficient than using both the < (less than) and > (greater than) operators together. BETWEEN . . . AND is able to perform one comparison, whereas the use of less than and greater than requires two comparisons. The IN clause is more efficient still, and should be used when you want to compare to a list of values.

The following query uses the less efficient greater than and less than to identify all orders in a range of values:

```
SELECT [OrderID]
FROM [Orders]
WHERE [OrderID] >= 10290 AND [OrderID] < 10301
```

This query can be optimized using the BETWEEN . . . AND operator:

```
SELECT [OrderID]
FROM [Orders]
WHERE [OrderID] BETWEEN 10290 AND 10300
```

This query can be further optimized using the IN operator; however, IN can become unwieldy (although still performant) when lots of values are included in the list:

```
SELECT [OrderID]
FROM [Orders]
WHERE [OrderID] IN
(10290, 10291, 10292, 10293, 10294, 10295, 10296, 10297, 10298, 10299, 10300)
```

You can use the NOT operator to reverse the logic with no impact on performance:

```
SELECT [OrderID]
FROM [Orders]
WHERE [OrderID] NOT BETWEEN 10290 AND 10300
```

Avoid Calculated Fields

Queries with certain expressions cannot be optimized. Examples include the IIf VBA function that is commonly used to create data fields based on data state. Any query that contains an IIf function will not be optimized and must be compiled each time the query is run, which may result in a sub-optimal query execution strategy.

You may be able to move the calculation from the query into a form or report control. For example, if your query contains an IIf expression, such as

```
=[FirstName] & IIf(Not IsNull([Initial]), " " & [Initial]) & " " & [LastName]
```

you can change the query to return FirstName, Initial, and LastName without the expression. Now, the query can be optimized and stored in a compiled state. In your form or report, you can create a control that builds the full name using the expression.

Many times, the need for an IIf statement will outweigh the potential performance penalty.

When working with nested queries, you can minimize the performance impacts by ensuring that any non-optimizable expressions exist only in the outermost query. If you have expressions that cannot be optimized in the nested queries, the entire query cannot be optimized. However, if your nested queries can be independently optimized, their execution plans can be stored and they will run faster.

Avoid using domain lookup functions in your queries. Domain lookup functions create new subqueries under the hood, which are not compiled.

Finally, avoid using calculated fields in WHERE clauses whenever possible.

Use Fixed Column Headings in Your CROSSTAB Queries

Fixed column headings not only ensure a consistent experience, they improve performance. Fixing your column headings ensures that the set of columns in the output remains constant, regardless of the data in your underlying tables. This allows the query to be optimized and stored in a compiled state.

Use Count(*) for Totals

When you need to count the number of rows in a recordset, it is more efficient to use the * (asterisk) wildcard character than to count a specific column.

The following query reports the number of orders for each employee:

```
SELECT [EmployeeID], Count([OrderID]) AS [Order Count]
FROM ORDERS
GROUP BY [EmployeeID]
```

Query Performance

However, replacing `Count ([OrderId])` with `Count (*)` is more efficient and results in better performance:

```
SELECT [EmployeeID], Count(*) AS [Order Count]
FROM ORDERS
GROUP BY [EmployeeID]
```

Use Stored Queries Instead of Recordset Properties

When you save SQL as a stored query, the query is optimized and saved in a compiled state. If a SQL string is specified in a recordset property (like the `RecordSource` property of a form or report), the SQL must be optimized and compiled each and every time you open the form or report. For maximum performance, save your SQL string as a saved query and use the query name in the recordset property.

Other Performance Considerations

In addition to the steps you can take when authoring a query, there are some other things to be aware of that may affect query performance.

Bulk Edits

When you have indexes on a table, the indexes need to be updated each time a row in the table is added, changed, or removed. Access is very efficient when updating indexes, but there is still substantial overhead involved. The overhead is negligible when working with a small number of rows (as when updating data through a form or grid).

The overhead of updating an index or multiple indexes can be enormous when performing bulk updates comprising many thousands of rows. In these cases, it often makes sense to remove the indexes entirely, then update the data, and finally, re-create the indexes.

Queries in Access are optimized by the Query Optimizer based on statistics about the underlying data at the time the query was last compiled. When you perform a bulk edit of data, it is possible that the query optimizer will decide on a different strategy for executing the query that is more performant given the current arrangement of data in your database. After performing a bulk edit, open your queries, make a change (add a space or remove the semicolon) so that the query is marked as dirty, and then save the query. This will cause the query to be run through the Query Optimizer and recompiled.

Remove indexes before performing bulk updates. Recompile your queries after performing bulk edits to ensure the query execution strategy is properly optimized.

Minimize Network Traffic

Network considerations are not an issue for a single-user database located on your workstation. But when you start dealing with multiuser databases located on a file server, or ADP projects with data stored on a centralized SQL Server, they become one of the biggest performance issues.

The network is a bottleneck that introduces enormous variability in your query performance, depending on network load from other users. Design your queries to reduce network traffic wherever possible.

When querying remote databases, run queries on the server. Use SQL Pass-Through queries to execute queries or stored procedures on the server.

When working with remote data using a DSN, try to make all of the tables in the query share the same DSN. If necessary, split your queries so that you restrict the remote data in a query where all tables share the same DSN, and then take the results of that query and join to local data or data in another remote data store. If you join on data from multiple DSNs, all of the data must be brought back to your local machine for processing.

Database Maintenance

Compact your databases regularly. Be sure to compact after all bulk updates and whenever you delete large amounts of data. When you delete data from a database, the data is marked for deletion, but it actually remains in your database until you perform a compact operation. Compacting the database will free deleted pages for reuse, and will perform a minimal defragmentation of the database. In addition to reclaiming deleted space and shrinking the size of the database file, the defragmentation of the pages will optimize the process of locating and retrieving data.

You can instruct Access to compact a database every time it is closed. From the Access Options dialog box, choose Compact on Close in the Current Database page. Choosing this option can result in a delay when closing a database depending on the size of the database being compacted.

Hard Drive Maintenance

Keeping your hard drive defragmented is equally important. It doesn't matter if the database organization is optimized if the database file itself is fragmented and disorganized on the hard drive.

Set up a regularly scheduled drive defragmentation. Windows includes a basic drive defragmentation tool that can be scheduled using the AT command. The defragmentation tool included with Windows Vista allows you to create and modify a schedule from within the tool. Most third-party defragmentation tools provide a robust set of scheduling options.

Microsoft Access Performance Analyzer

Access includes a Performance Analyzer that will suggest options for increasing performance in your database. If you choose to analyze your queries, the Analyzer will suggest changes to your database that may improve performance. The tool is a good way to identify missing indexes on your tables. The Analyzer will suggest indexes that can be used by the Query Optimizer to improve the performance of your queries.

While the suggestions made are generally valid, there are many performance improvements that the Analyzer does not suggest, so just because the Analyzer doesn't offer a suggestion is not a certification that your database is performant.

To launch the Performance Analyzer, choose the Database Tools ribbon and select the Analyze Performance button in the Analyze chunk.



Pattern Reference

Throughout this book, we have presented code that you can re-use in many applications. The decision about whether to create a library for some of this code may be personal in some instances and practical in others, but the fact is that by re-using code you can save yourself quite a bit of time. Code that you re-use has the benefits of being well-tested and proven, and likely very efficient.

In this appendix, we discuss patterns and consolidate some of them into a single location that you can use to quickly find information. We'll do this by:

- Introducing patterns
- Calling out the patterns that we've used in this book
- Providing pointers about where you can find more patterns

Introducing Patterns

Patterns as a means for describing a particular problem space were first introduced in architecture by an architect named Christopher Alexander. Although initially described to address ways to solve problems in physical architecture, Alexander's work has been tailored into other fields such as software development. Patterns provide a formalized mechanism for describing the problem, including the intent of the pattern, and how it solves the problem in question.

What Is a Pattern

As with architecture, a pattern in software development is an approach to solving a particular problem. The pattern isn't necessarily prescriptive — in many cases it provides only a guideline for solving a problem as there is often more than one way to solve it. What the pattern provides, however, is a *name* and description of a problem. By giving them names, patterns create a language by which software developers can communicate.

Pattern Reference

Consider a scenario in which you need to read records from a database. In our case, the database might be a relational database such as Access, but even the term “database” might have more than one meaning. To some, a database might refer to a flat file, to others a relational database, and to others an object-oriented database. The means for accessing data in the database is an implementation detail, but when you talk with other developers about retrieving records, most people know what it means to retrieve records.

Why Use Patterns?

In addition to providing a language with which we can discuss development problems with other developers, patterns provide a framework for what works. As software developers, we tend to strive for code re-use where we can and constantly revisit code that we’ve written previously with an eye toward using it in other projects. In many cases, we can standardize on the code and focus on solving the underlying problem that a database is designed to solve.

Patterns Used in This Book

In this book, we’ve included VBA implementations of some well-known design patterns in software development, as well as discussed several of those that we use in our own development work. These patterns are discussed in the remainder of this appendix.

Gang of Four Patterns

In 1994, the book *Design Patterns: Elements of Reusable Object-Oriented Software* was released by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Collectively, these authors came to be known as the Gang of Four. This book was instrumental in its description of solving problems in object-oriented programming and is considered a classic in software engineering.

We have included three of the patterns described in the *Design Patterns* book in this book with implementations in VBA. Let’s take a look.

Prototype

The Prototype pattern is a means for creating a copy of an object, or a prototype. The Gang of Four implementation of this pattern includes a method called `Clone` to create the copy. In Chapter 3, we implemented this method as a means for creating a copy of a class module. The steps for implementing the `Clone` method are:

1. Create a new instance of the object being copied.
2. Copy the property values from the current instance to the new instance.
3. Return the new instance.

Here is our implementation:

```
Private Function ISearch_Clone() As ISearch
    ' create the new instance and return
    Set ISearch_Clone = New ContactSearch
    ISearch_Clone.Name = ISearch_Name
End Function
```

Factory

The Factory pattern is a technique for object instantiation. In most cases, the object being created may not be known until runtime but the specific object type may derive from a common interface. The implementation of this pattern involves a method that performs the instantiation. To address this issue, we've implemented a Factory method called `CreateLog` in Chapter 4. Here is our implementation:

```
Public Function CreateLog(LogType As LogTypes) As ILog
    Select Case LogType
        Case LogTypes.LogDatabase
            Set CreateLog = New DatabaseLog
        Case LogTypes.LogTextFile
            Set CreateLog = New TextLog
    End Select
End Function
```

Iterator

The Iterator pattern as described by the Gang of Four is a means for providing iteration through a collection while hiding the implementation inside the collection. In other words, a collection object knows how to enumerate through its members while hiding the underlying implementation from callers. In Chapter 3, we added a `ForEach` method of the `Searches` collection class that meets this criterion. Our method was used to run an action for each item in the collection.

Here is our implementation:

```
Private Sub ICollectionEx_Foreach(strFunction As String)
    ' runs the specified function name for each item
    Dim i As Long

    For i = 1 To m_col.Count()
        Application.Run strFunction, m_col(i)
    Next
End Sub
```

VBA Object Patterns

As Access developers, you likely use certain pieces of code over and over again. You've tested it on many occasions and have released it more than once. In other words, it just works. In Appendix A, we've added some routines that we use to manipulate object data types in VBA. These are patterns in the sense that they solve the problem of object creation or destruction, and do so in a manner that is designed to provide consistency throughout an application.

Let's give some names to these patterns. We'll start by categorizing them into *creational* patterns or *destruction* patterns.

Creational

The creational patterns are used to create instances of objects. The type of the object being created doesn't necessarily matter, and will vary in many cases. The Factory pattern mentioned earlier is an example of a creational pattern.

Pattern Reference

Object Instantiation

We typically use this pattern when creating new instances of objects, and include a check to ensure that the object being set is not `Nothing`. This is to prevent overwriting the existing object.

This code was mentioned in Appendix A:

```
Public Sub InitObject(objToSet As Object, newObject As Object)
    If (objToSet Is Nothing) Then
        Set objToSet = newObject
    End If
End Sub
```

Recordset Instantiation

There are many places throughout an application where we create recordset objects such as when we retrieve a single value, update rows, or enumerate records to write to a file. For these cases, we invariably call the `OpenRecordset` method if we are using DAO, which may require a Database object. We've abstracted this call into a simple method that can be called throughout an application.

This code was also mentioned in Appendix A:

```
Public Sub InitDAORecordset(objRS As DAO.Recordset2, strName As String, _
    Optional ExistingRecordset As DAO.Recordset2)

    ' set the object
    If (objRS Is Nothing) Then
        ' create a copy of the recordset or a new recordset
        If (Not (ExistingRecordset Is Nothing)) Then
            Set objRS = ExistingRecordset.Clone()
        Else
            Set objRS = CurrentDb.OpenRecordset(strName)
        End If
    End If
End Sub
```

The preceding code makes a copy of an existing `Recordset` object using the `Clone` method if possible.

Destruction

A destruction pattern is a pattern used to free resources for a given object in VBA. Again, the type of the object doesn't necessarily matter and in many cases will vary based on the specific type of object being destroyed. Because VBA does not have built-in garbage collection as mentioned in Chapter 1, destruction of objects is very important.

Object Destruction

Cleaning up an object is as simple as setting it to `Nothing`. We abstract it into a separate method as a reminder to clean up objects, and to provide a consistent place to free global objects. We tend to wrap this in a cleanup method that runs when the user exits the application.

```
Public Sub DestroyObject(obj As Object)
    Set obj = Nothing
End Sub
```

Recordset Destruction

When you clean up a recordset, it's also a good practice to close the recordset first. Because there are extra steps to take before destroying the object, we've created a method for cleaning up Recordset objects:

```
Public Sub DestroyDAORecordset(rs As DAO.Recordset2)
    If (Not (rs Is Nothing)) Then
        rs.Close
        DestroyObject rs
    End If
End Sub
```

Other Patterns

Besides the Gang of Four and VBA patterns listed here, we've thrown in some other patterns that are used in other frameworks.

Equals Method Pattern

In the .NET Framework, the `Equals` method is used to determine whether two objects are equal. This method is defined on a given object, and compares values in one object to those of another instance of the object to determine equality. The `Equals` method solves the problem of how to compare two instances of a class.

This method was defined in Appendix A:

```
Public Function Equals(obj As Person) As Boolean
    Equals = (Me.SSN = obj.SSN)
End Function
```

Notice that we are comparing the current instance (as specified using the `Me` keyword) to the instance specified in the argument.

Seek Method

There are many tasks that we do in DAO on a routine basis such as opening recordsets, searching for data, or updating records. The code for many of these tasks often looks the same. The `Seek` method on the `Recordset` object in DAO is used to search for records in a table by using an index on the table. In Chapter 7, the following code demonstrates common use of the `Seek` method:

```
Dim rs As DAO.Recordset2
Dim vbm As Variant

Set rs = db.OpenRecordset("tableName", dbOpenTable)
rs.Index = "Index Name"
' Cache the current record so we can restore if no
' match is found and the current record is not set
vbm = rs.Bookmark

' Perform the seek
rs.Seek "=", "FieldValue"
' if no match is found, restore the bookmark so
```

Pattern Reference

```
' that we have a current record
If rs.NoMatch Then
    rs.Bookmark = vbm
Else
    ' We found the desired record, do something useful
    Debug.Print rs(0)
End If
```

Attachment Fields

As described in Chapter 7, the Attachment data type is a new data type in Access 2007. This data type is used to store files in a database in a more efficient manner than the OLE Object data type. An Attachment field is a special type of *multi-valued field*. Remember that multi-valued fields can store multiple records in a single field. These records are stored in an embedded recordset in the field. Because two recordsets are involved, we use code similar to the following when working with attachment fields in code:

```
Dim db as DAO.Database
Dim rsOuter as DAO.Recordset2
Dim rsComplex as DAO.Recordset2
Set db = CurrentDb()
Set rsOuter = db.OpenRecordset("TableName")
' Enumerate each record in the table
Do until rsOuter.EOF
    ' Get the complex data recordset
    Set rsComplex = rsOuter("MultiValuedFieldName").Value
    ' Enumerate the items in the multi-valued field
    Do Until rsComplex.EOF
        '// Do something with the complex item
        Debug.Print rsComplex(0)
        rsComplex.MoveNext
    Loop
    rsComplex.Close
    Set rsComplex = Nothing
    rsOuter.MoveNext
Loop
rsOuter.Close
Set rsOuter = Nothing
db.Close
Set db = Nothing
```

Working with Strings

Many applications work with strings in one form or another. You might need to parse a complex piece of information or extract a string from another string, or replace a portion of a string with another string. To help with this, here is a recap of some of the functions we often use.

Token Replacement

The `String` class in the .NET Framework provides a `Format` method that is used to produce a formatted string. The method accepts a string which includes format identifiers that are indicated by index,

and an array of arguments that are used to replace the format identifiers. For example, in C#, you might use this method as follows to create a string separated by tabs:

```
string myString = string.Format("{0}\t{1}\t{2}", "item1", "item2", "item3");
Console.WriteLine(myString);
```

The output of this method would be:

```
item1    item2    item3
```

We can use the `ParamArray` keyword in VBA with the `Replace` function to achieve the same thing. The following code was included in Chapter 5:

```
Public Function StringFormat(ByVal item As String, _
    ParamArray args() As Variant) As String
    Dim i As Long
    For i = LBound(args) To UBound(args)
        item = Replace(item, "{" & i & "}", args(i))
    Next i
    StringFormat = item
End Function
```

Formatting Numbers as Hexadecimal

When we debug or use the `ObjPtr` function, we tend to like to view numbers in their hexadecimal form. Hexadecimal is a number system in base-16 where numbers range from 0 to 9 and A to F, where F = 15. The `Hex` function in VBA is used to format a number as a hexadecimal string, but it uses the least amount of space possible. If you wanted to format the number to a fixed number of digits, you'll need to write your own. This is done by combining a few different functions in VBA: `Hex`, `Format`, and `Replace` as shown:

```
Public Function FormatHex(lValue As Long, _
    Optional nLength As Long = 8, _
    Optional sPrefix As String = "0x") As String
    ' format the number
    FormatHex = Replace(Format(Hex(lValue), String(nLength, "@")), " ", "0")
    FormatHex = sPrefix & FormatHex
End Function
```

Where and How to Find Patterns

You can find patterns throughout your code. Consider a piece of code that you write frequently. This code was written to solve a problem, and if you can name it and describe it, and it is designed to solve a particular problem in development, then it could be a pattern. In addition to the patterns that you use every day, there are other places to find patterns. Here are a few of the resources we have found for patterns:

- ❑ **Design Patterns on Wikipedia:** http://en.wikipedia.org/wiki/Design_Patterns
- ❑ **GoF design patterns in C# or VB.NET:** www.dofactory.com/Patterns/Patterns.aspx
- ❑ **Portland Pattern Repository:** <http://c2.com/ppr/index.html>

Index

SYMBOLS

(Hash Mark), with LIKE operator, 207–208
% (percent sign) ANSI wildcard, 261
* (asterisk)
adding fields with, 197–199
to count recordset rows, 661–662
with LIKE operator, 207–208
[] (square brackets), 261
with LIKE operator, 207–208
in SQL code, 200–201
< (less than) operator
vs. BETWEEN . . . AND operator, 660–661
searching indexes with, 267
and WHERE conditions, 206
< > (not equal to) operator, 207
<= (less than or equal to) operator, 206, 267
= (equal to) operator, 207, 267
> (greater than) operator
vs. BETWEEN . . . AND operator, 660–661
in Immediate window, 651–652
searching indexes with, 267
and WHERE conditions, 207
>= (greater than or equal to) operator, 207, 267
? (Question Mark) with LIKE operator, 207–208
“ (quotation marks) for delimiting strings, 207
() parentheses to group conditions, 206

A

About dialog box, 351
ACCDB file format, 377, 527–528, 531
ACCDE file format, 531, 557
ACCDR files, 557

Access 2007
Access 2007 VBA Programmer’s Reference, 6, 272, 567
Append Only memo fields, 285–289
Attachment fields, 285
Developer Extensions, 139
exporting HTML from, 182–187
navigation pane. See navigation pane
Performance Analyzer, 663
Report view, 357
security overview. See security
SQL usage overview, 194
writing code for, 3–5
writing managed code libraries for, 6–7
access control entries (ACE), 527
access control lists (ACL), 527
Access object model
basics, 4
referencing from managed code, 7
accidental parameters, 230
ACE (Microsoft Access database engine), 193, 261
action queries, 238–239
Active Directory, 526
ActiveX controls, 319, 321
ActiveX Data Objects (ADO) object model. See ADO (ActiveX Data Objects) object model
Add method (VBA Collection object), 86, 89–90
address element granularity, 180–181
AddressOf operator (VBA), 54, 550
AddWindowsPrinterConnection method, 406
administrator, user as (Shell function), 27
ADO (ActiveX Data Objects) object model
changing database passwords with, 555–556
defined, 4
AfterDelConfirm event, 301
AfterInsert event, 300
AfterUpdate event, 301
aggregate functions
SQL, 234–236
VBA domain, 238
Alexander, Christopher, 665
aliases
field names, 202–203
keyword, 13
tables names, 204–205
ALIKE (ANSI-Like) operator, 261
ALL predicate, 231
AllowBypassKey property, 556–557
alpha-numeric keys
(registration), 591
ALTER DATABASE PASSWORD statement, 259
ALTER TABLE statement, 257–258
And Not operators, 66–67
ANSI (American National Standards Institute)
ANSI-89 and ANSI-92 syntax, 261
file formats, 160–164
Strings, 16
API (application programming interface). See also Windows API
defined, 5
resources, 12
API functions
determining locale with, 492–493
error messages from, 18–19
finding, 14
GetDriveType API, 611
LoadString API, 504–506
locking computer, 548
reasons for, 12
ShellExecute API, 629–630
SQLConfigDataSource API, 608–609
appending
Append Only memo fields, 285–289
Append queries, 240–243
records, 241–243

application dependencies (example)

application dependencies (example)

installing files from attachment fields, 579–581
late binding, 584–585
testing references, 583–584
updating references, 581–582

application options
Color scheme option (Office 2007), 506–507
displaying in Ribbon, 516–518
managing from tables, 509–510
managing in registry, 509–513
option constants, creating, 509
option forms, creating, 513–516
options class, creating, 509
storing in registry, 508–509
storing in tables, 508
uses for, 508

application programming interface (API). See API (application programming interface)

Application.ColumnHistory function, 285–289

Application.Run method, 641

applications

adding forms to. See forms build process for Access-based. See BuildAccApp example licensing. See licensing applications localization of. See localization of applications Office. See Office applications (Microsoft) running from local computer, 610–612 updating. See updating applications

appointments, automating in Outlook, 429–432

Array VBA function, 282

arrays of dates, 646

AS reserved word, 202

ASC (ascending sort order) keyword, 258

assertions

Assert method, writing, 117–120 conditional compilation, 116–117 Debug.Assert method, 115–116 DebugEx.Assert method, 119–120

asterisk (*)

adding fields with, 197–199 to count recordset rows, 661–662 with LIKE operator, 207–208

Attachment fields

displaying images with, 377–378 getting list of attachments in, 289–292 installing files from, 579–581 loading images from, 468–469 new in Access 2007, 285 patterns, 670 searching for records with, 292–294 searching for specific, 294–296

attachments

AttachmentCount property, 323 displaying multiple onscreen, 322–323

attributes

description (menus), 459 imageMso, 463 itemSize (menus), 459 loadImage callback, 463–465 onAction, 446, 450

Ribbon controls customization, 446

authentication, 526–527

authorization, 526–527

automated builds, creating. See BuildAccApp example

automation

basics, 387–391 of Internet Explorer 7, 432–440 of Office applications, 409–432 server, defined, 389 of Windows environment, 391–409

AutoNumber fields, 255

B

back-end database, 604

backups

of objects, 653–654 and restoration of data, 276–278 with Timer event, 306–307

bars, navigation, 352–353

BeforeDelConfirm event, 301

BeforeInsert event, 300

BeforeUpdate event, 302–303

BETWEEN . . . AND operator, 660–661

BETWEEN lowValue AND highValue operator, 207

BETWEEN...AND operator, 209–210

binary fields, sorting data on, 218

binding, late vs. early, 389–390

BOF property, 268–271

borderless forms, moving, 307–308

boxOutside/boxInside rectangle controls, 319–321

Break on Unhandled Errors (VBE), 118

Break When Value Changes watch type, 108

Break When Value Is True watch type, 107–108

BrokenReference property, 581

browsers

Object Browser (VB), 389–390 opening default, 60 opening HTML help pages in, 628–630

BuildAccApp example

adding references, 560–561 build button, 569 build properties, 563–564 build properties, setting on source database, 576–577 build properties, setting on target database, 574–576 calculating version number, 577–578 creating target database, 573–574 creating with Visual Studio, 560 deleting data from tables, 577 form design, 563–569 helper functions, 561–562 private data, 561–562 retrieving information from source database, 569–571 writing BuildApplication method, 571–572

bulk printing (reports), 384–385

bulk updates, 662

buttons

adding (BuildAccApp example), 565–566 build (BuildAccApp example), 569 exit (BuildAccApp example), 569 radio in task dialogs, 50–52 in Ribbon customizations, 455 setting text in task dialogs, 44–46 split buttons, 462, 482–484 toggle buttons in Ribbons, 455–456

ByRef keyword, 20, 168

ByRef vs. ByVal, 15

Byte Order Mark (BOM), 160

C**C languages**

constructors in C#, 73
conversions to VBA data types, 14–18

managed code in C#, 5–6

calculated fields, avoiding, 661**calendar reports**, 370–376**Call Stacks**

building with code, 111
class, creating, 111–112
generating with error handling, 113–115
instrumentation of code, 112–113
viewing, 110

callbacks

callback functions, 52–54
callback routines with Ribbons, 449–452
defined, 446, 449
to display options in Ribbon, 517–518
getImage and getItemImage, 465–468

CallByName VBA function, 637–640**cancelling events**, 98**Cartesian products (Cartesian joins)**, 225–227**Case Else block**, 151**categorization**

of constant values, 647
of data, 272–276
of errors, 124
Navigation pane categories, 533–536
of reports, 382

cb parameter prefixes, 27**cch parameter prefixes**, 27**CDbl conversion VBA function**, 644**certificates**

Certificate Services servers, 531
self-signed, 532

characters

in API data types, 14
line-continuation (code), 138

charts, automated Excel, 421–429**check boxes**

controls (BuildAccApp example), 565

in Ribbons, 456–457

checksum functions

library of, 278–279
MD5 algorithm, 279
Mod-10 checksum, 279–285

Choose VBA function, 273**class module members**

collection classes, 86
enumerations, 65–67
events, 72–73
methods, 70–71
properties, 68–70
setting default, 91

class modules, VBA

class instances, copying, 100–101
code reuse in VBA, 73–76
Collection object. See Collection object (VBA)
designing, 99–103
events, 96–99
exposing interface of derived types, 101
interfaces, 80–85
raising errors, 102–103
subclassed, 76–80
uses for, 64–65

classes

Callstack, creating, 111–112
comparing instances of, 648–649
creating instance of ILog, 134–135
creating options, 509
defined, 4–5
error handler, 125–129, 135
TaskDialog, 64, 68
VBA class modules, 99–103
writing collection class, 86–88

Clear method (Collection object), 93**client application (updating applications)**, 603–604**client-server databases**, 556**Clone method**, 101, 666**Close statement (VBA)**, 149–151**CloseHandle function**, 29**CLSID (classification identifier)**, 389**CoCreateGuid API function**, 58–59**code**

adding functionality to forms. See form events; forms
adding functionality to reports. See reports
adding retries to code blocks, 649–650
adding timeouts to code blocks, 650–651
building Call Stacks with, 111
closing code windows, 654–655
creating HTML with, 187
dynamically running, 637–641
instrumentation of, 112–113

pausing execution of, 28
refactoring, 139–141, 637
reusable. See patterns
reuse in VBA, 64, 73–76
writing for Access, 3–5

coding practices

code reviews, 142
comments, 136–138
formatting, 138
naming conventions, 138–139
refactoring, 139–141
version control, 139

Collection object (VBA), 85–96

Add method implementation, 89–90

Clear method, 93

Contains method, 93

For Each enumeration, 91–92

Find method, 93

ForEach method, 94

ICollectionEx interface, 92–93, 95–96

setting default members, 91
writing collection class, 86–88

Color scheme option (Office 2007), 506–507, 518**ColumnHistory function**, 285–289**columns**

fixed headings in crosstab queries, 661

ordering headings, 248–249

specifying in field lists, 197

COM (Component Object Model)

add-ins, 441–442
as basis for automation, 387
and GUIDs, 58
references, 7

combo boxes (Ribbons), 457–458**comctl32.dll**, 35**command line**

retrieval function, 21–23
specifying debug builds with, 121

command links, 46–48**commands**

CommandBarControl object, 449
overriding, 485–486
running with ShellExecute method, 392–394

comments, code, 136–138**compacting databases**, 663**comparison operators**, 206–207, 267**Compiled HTML Help files**, 631–635

computer Internet connection Shell function, 27
computer name return function, 21
conditional compilations
creating debug builds with, 122
defining constants, 116–117
#Const directive, 116
constants
option, creating, 509
values, categorizing, 647
constructors in C# and C++, 73, 562
Contains method (Collection object), 93
context mapping (HTML Help files), 633–634
context-sensitive help topics, 632
contextual tab controls, 453–454
control panel
ControlPanelItem method, 394
launching items in, 392–394
controls
callback routines for, 449–452
disabling, 310
finding in Office 2007, 443–444
list boxes. See list boxes
refreshing in customized Ribbons, 452
on reports, sorting with, 358–359
repurposing, 485–486
reusing subform controls, 310–311
validating control data, 309–310
controls, Ribbon
buttons, 455
check boxes, 456–457
combo boxes, 457–458
dialog box launchers, 460–461
drop-down controls, 457–458
dynamic menus, 462
galleries, 461
images. See images
labels and edit boxes, 458
menus, 458–460
split buttons, 462
toggle buttons, 455–456
ControlTipText property
customizing, 308–309
in form and report controls, 620–621
Count (*) function, 661–662
Count property (VBA Collection object), 87
country codes, 181–182
CREATE INDEX statement, 258

CREATE TABLE statement, 255
CreateObject vs. GetObject, 389
CreateProcess API function, 32
CreateReportControl method, 371
CreateShortcut method, 402–403
creational patterns, 667–668
cross-products. See Cartesian products (Cartesian joins)
crosstab queries, 246–250, 661
CSIDL values, 24
CurrentProject object, 529
custom events
creating, 96–98
listening for, 98–99
custom form navigation, 352–355
custom reports, user, 385–386
customUI node, 445

D

DAO (Data Access Objects) object model
adding database passwords with, 554–555
changing database passwords with, 553–554
creating encrypted databases with, 553
defined, 4
properties, 614
dashboards
component table, creating, 334
components, adding, 341–348
dashboard components table, creating, 334–335
dashboard table, creating, 333–334
defining layouts, 324–332
filling, 348–349
form for choosing, 336–340
opening selected, 340–341
table relationships, creating, 335
data
adding from different database formats, 230–231
adding from multiple tables. See table joins
aggregating, 234–238
data bars, creating reports with, 366–370
Data Definition queries, 254–259
deleting from tables (BuildAccApp example), 577

determining added (form events), 300–301
determining changed (form events), 301
determining deleted (form events), 301–302
filtering. See HAVING clauses; WHERE clauses
prompting with parameters, 227–230
protecting with client-server databases, 556
reduction for performance, 659–660
validating control, 309–310
validating form, 302–304
Data Access Objects (DAO). See DAO (Data Access Objects) object model
data manipulation
Attachment fields. See Attachment fields
backup and restore, 276–278
categorization and sorting, 272–276
checksum functions. See checksum functions
ColumnHistory function, 285–289
Find methods, 264–266
Move methods, 268–271
searching for records with attachments, 292–294
searching for specific attachments, 294–296
Seek method, 266–268
data types
C conversions to VBA, 14–18
choosing smallest for table fields, 658
conversion functions (VBA), 171–174
Date. See dates
Memo data type, 285
parsing text into, 171–174
SQL, used with DDL, 255–256
databases
adding from different formats, 230–231
adding passwords with DAO, 554–555
ALTER DATABASE PASSWORD statement, 259
automating Outlook appointments from, 422

- back-end database, 604
 changing passwords with ADO, 555–556
 changing passwords with DAO, 553–554
 compacting, 663
 creating registration, 592
 creating target (BuildAccApp example), 573–574
 Database Documenter, 621
 database paths (BuildAccApp example), 563
 Database Splitter, 604
 encryption of, 527–528
 encryption with DAO, 553
 front-end database, 604
 logging errors to tables, 130–132
 properties, creating debug builds with, 122–123
 protecting data with client-server, 556
 setting build properties on source, 576–577
 setting build properties on target, 574–576
 spaces, usage of, 201
- dates**
 application expiration, 586–588
 arrays of, 646
 CDate data type conversions, 173
 DateDiff function, 646
 determining dynamically, 646–647
 Find methods and, 272
 looping through, 645–646
 matching on, 214
 in reports, 375
- DCount function, 316**
- DDL (Data Definition Language)**
 basics, 195
 SQL data types used with, 254–256
- debugging VBA code, 106–123**
 assertions. See assertions
 Call Stacks. See Call Stacks
 debug builds, creating, 120–123
 Debug.Assert method, 115–116
 Debug.Print, 109
 Immediate window, 109–110
 Locals window, 110
 Watches window, 106–109
- Declare statements, writing, 13–18**
- declaring parameters, 228–229**
- DefaultControl property (Report objects), 371**
- defragmenting hard drives, 663**
- deleting**
 data from tables (BuildAccApp example), 577
 Delete event, 301
 DELETE FROM statement, 246
 DELETE queries, 245–246
 determining deleted data (form events), 301–302
 indexes from tables, 259
 items in list boxes, 317–318
 registry keys/values, 398
 tables, 258
- dependencies, application.**
 See **application dependencies (example)**
- depends.exe graphical tool, 12**
- deploying resource DLLs, 504**
- DESC (descending sort order) keyword, 258**
- description attribute (menus), 459**
- Description property**
 Access objects, 614–619
 WshShortcut object, 404
- designing**
Design Patterns: Elements of Reusable Object-Oriented Software, 666
 Design Patterns on Wikipedia, 671
 VBA class modules, 99–103
- desktop shortcuts, creating, 402–404**
- destruction patterns, 668**
- dialog box launchers, 460–461**
- dialog box, property (Shell function), 25–27**
- digital signatures, 531**
- Disable Design View shortcuts, 539**
- disabled mode feature (security), 528–529**
- displaying**
 Display As Hyperlink property, 364
 Display method (error handler class), 128
 Locals window, 652
 options in Ribbon, 516–518
 Watches window, 652
- DISTINCT predicate, 233–234**
- DLLs, resource. See resource DLLs**
- DML (Data Manipulation Language)**
 basics, 195
 vs. DDL, 254
- documentation**
 ControlTipText property, 620–621
 Database Documenter, 621
 Description property, 614–619
 StatusBarItem property, 619–620
 Tag property, 620
- documents, automated MS Word, 412–421**
- DOM (Document Object Model), 435**
- domain aggregate functions, 238**
- domain lookup functions, 661**
- Double data type, 644–646**
- drill-down (reports), 363–365**
- DROP COLUMN statements, 257**
- DROP INDEX statement, 259**
- DROP TABLE statement, 258**
- drop-down controls in Ribbons, 457–458**
- drop-down lists, 354–355**
- DSNs (Data Source Names)**
 creating programmatically, 607
 DSN-less connections, creating, 607–608
 system DSN, creating, 608–610
 user DSN, creating, 608
- dumpbin.exe command-line tool, 12**
- dynamic determination of dates, 646–647**
- dynamic menus**
 managing filters with, 477–481
 in Ribbons, 462
- E**
- early binding**
 defined, 8
 vs. late binding, 389–390
- edit boxes in Ribbons, 458**
- e-mails**
 e-mailing reports, 382–383
 sending with default client, 60
 sending with default mail program, 394–395
- encapsulation, defined, 65**
- encodings, file, 160**
- encryption**
 of databases, 527–528
 of databases with DAO, 553
 Encrypt with Password button, 486
- enumerations**
 For Each enumeration, 91–92
 error handler class, 125–126
 fundamentals of, 65–67

EnumPrinterConnections method

- EnumPrinterConnections**
 method, 408
- EOF property**, 268–271
- Equals method**, 648, 669
- equi-joins**. See *inner joins (equi-joins)*
- error handling**, 123–135
 categorizing errors, 124
 Err object (VBA), 18
 ERR_AUTOMATIONEXCEPTION
 error, 407
 ERR_PRINTERNAMEINVALID
 error, 407
 ERR_REMOTE SERVERUNAVAIL
 error, 407
 error display in Ribbon
 customization, 442
 error events, defining, 103
 Error events (forms), 78–80, 304
 error handler class, creating,
 125–129
 error messages from API functions,
 18–19
 error messages, suppressing,
 304–305
 Err.Raise statement (VBA),
 102–103
 and file I/O, 148–149
 File Not Found error, 148, 151
 generating Call Stacks with,
 113–115
 HRESULT error codes, 125
 inline error handling, 123–124
 LastDLError property, 18–19
 logging errors. See *logging errors*
 Permission Denied error, 148, 151
 raising errors in class modules,
 102–103
 system error codes, 124–125
- Eval function (Access Application object)**, 641
- events**. See *also form events*
 adding to options form (example),
 514–516
 canceling, 98
 custom, 96–99
 database, Outlook appointments
 from, 429–432
 defining error, 103
 Error (forms), 78–80, 304
 Event keyword, 97
 Form_Error, subclassing, 78–80
 Form_Load, subclassing, 80
 fundamentals of, 72
- Initialize, 70, 72–73
 Load (forms), 80
 NotInList, 470–472
 Paint, 368–370
 Private, 96
 Public, 96
 RaiseEvent keyword, 97–98
 responding to form, 77–80
 similar to control callbacks, 451
 Terminate, 73
 Timer, 306–307, 350–351
 Undo, 304
 WithEvents keyword, 76–77
- Excel, Microsoft, automating**, 421–429
- Exec method (WshShell object)**, 404–405
- exit button (BuildAccApp example)**, 569
- ExpandEnvironmentStrings method**, 398
- expiration dates, application**, 586–588
- explicit method**, 73
- expressions**
 adding in field lists, 199–201
 used with crosstab queries,
 249–250
 Watch Expression, 107
- F**
- F1 key, trapping**, 623–624
- Factory pattern (Gang of Four)**, 667
- field lists**
 expressions, adding, 199–201
 overview of, 197
- field names**
 adding fields with, 197–198
 aliases, 202–203
 fully qualified, 198
 square bracketing, 200–201
- fields**
 adding with asterisk (*), 198–199
 adding with wildcard characters,
 198–199
 avoiding calculated, 661
 custom sort order, 273–276
 MVF, 289–293
 sorting on binary or memo, 218
- File DSN**, 607
- files**
 browsing for (BuildAccApp example), 568–569
- creating HTML, 187–191
 encodings, 160–164
 File Not Found error, 148, 151
 formats, determining, 160–164
 formats, selecting, 557–558
 input/output. See *input/output statements (VBA)*
 installing from attachment fields,
 579–581
- FileSystemObject**
 and Microsoft Scripting Runtime
 reference, 24
 SpecialFolder Enumeration
 Constants, 401–402
 using, 157–159
- filtering**
 data. See *HAVING clauses; WHERE clauses*
 Filter property, 194
 managing with dynamic menus,
 477–481
 multi-select list boxes for criteria,
 318–319
 reports with controls, 360–363
- Find method (Collection object)**, 93, 264–266
- FindWindow function**, 29
- flag values, combining**, 66–67
- folders**
 determining special, 400–401
 opening with Windows Shell
 object, 392
 SpecialFolders strings, 400–401
 temp, 401–402
- FollowHyperlink method**, 94
- FONT tags**, 186
- footers in task dialogs**, 43–44
- For Each enumeration**, 91–92
- ForEach method (Collection object)**, 94
- form events**
 added data, determining, 300–301
 backups with Timer event,
 306–307
 changed data, determining, 301
 ControlTipText property,
 customizing, 308–309
 deleted data, determining,
 301–302
 displaying multiple attachments,
 322–323
 error messages, suppressing,
 304–305

-
- modifier keys, determining use of, 305–306
moving borderless forms, 307–308
progress bars, custom, 319–321
up/down buttons, custom, 321–322
validating form data, 302–304
- formatting**
code, 138
file formats, 160–164, 557–558
Format VBA function, 249
FormatCurrency function (VBA), 495
FormatDateTime function (VBA), 495
FormatNumber function (VBA), 495
StringFormat function, 154, 170–171, 188, 625–627
- forms**
About dialog box, 351
ControlTipText property, 620–621
creating option (example), 513
creating password-protecting object, 545–546
custom navigation, 352–355
design of (BuildAccApp example), 563–569
dynamic menus and dashboards.
See dashboards
Form_Error event, subclassing, 78–80
Form_Load event, subclassing, 80
help forms, creating, 622–623
kiosk forms, 351–352
navigation in Ribbons, 472–477
options form, modifying, 518
parameters, 228
password prompting with, 544–545
recordsets, overview, 194
responding to events, 77–80
setting properties (BuildAccApp example), 566–567
splash screens, 349–351
subclassed, 76–77
themes, creating, 518–523
- FreeFile VBA function, 149**
- FreeLibrary API function, 504–506**
- FROM clauses, 203–205**
- front-end database, 604**
- functions. See also individual function names**
aggregate, 234–236, 238
API. See API functions
callback, 52–54
checksum. See checksum functions
data type conversion, 171–174
flag-related, 67
hash, 279
helper, 561–562, 641–644
Shell. See Shell functions
string handling. See string handling functions, VBA
VBA for global applications, 495–496
- G**
- galleries in custom Ribbons, 461, 465–467**
- Gamma, Erich, 666**
- Gang of Four pattern, 666**
- garbage collector, defined, 6**
- GetClassName API function, 31**
- GetCommandLine API function, 21–22**
- GetComputerNameExA API function, 21**
- GetCurrentProcessId function, 28**
- GetDatabaseProperty method, 571**
- GetDriveType API function, 611**
- getImage and getItemImage callbacks, 465–468**
- GetLocaleInfo function, 492**
- GetObject vs. CreateObject, 389**
- GetSpecialFolder method, 401**
- GetTempName method, 401–402**
- GetUserDefault LCID function, 492**
- GetUserNameEx API function, 19**
- GetWindowThreadProcessId function, 29**
- global applications, VBA functions for, 495–496**
- global image handler, creating, 463–465**
- global object properties, 644**
- globally unique identifiers (GUIDs). See GUIDs (globally unique identifiers)**
- GoF design patterns in C# or VB.NET, 671**
- granularity**
address element, 179–181
phone number, 179–181
- GROUP BY clauses, 234–238**
- groups, Navigation pane, 533–536**
- GUIDs (globally unique identifiers)**
creating, 58–59
matching on values, 214–215
- H**
- hard drive maintenance, 663**
- hash functions, 279**
- HAVING clauses, 237–238**
- headers**
column, 248–249
module (comments), 136
procedure (comments), 136–137
- Heim, Richard, 666**
- help**
adding HelpContextID values to controls, 633–634
alternatives to floating window, 624–625
Compiled HTML Help files, creating, 631–635
F1 key, trapping, 623–624
forms, creating, 622–623
HelpContextID property, 621, 627
HelpFile property, 621, 635
HTML content, writing, 625–627
HTML files, mapping to objects, 627–628
HTML Help Workshop, creating content for, 631–632
LaunchHtmlPage function, 628–630
storing help text in tables, 621–622
- helper functions**
BuildAccApp example, 561–562
object, 641–644
- hexadecimal notation, 671**
- Hidden Members, Show, 390**
- HighlightFrame routine, 345, 347**
- HRESULT error codes, 125**
- HTML (Hypertext Markup Language)**
Compiled Help files, creating, 631–635
content for help, writing, 625–627
creating with code, 187
exporting from Access, 182–187
files, creating, 187–191
files, mapping to objects, 627–628
help pages, opening in browsers, 628–630
Help Workshop, creating content for, 631–632
LaunchHtmlPage function, 628–630
parsing from IE, 435–440
templates, 184
- https protocols, 60**
- hyperlinks, adding in task dialogs, 52–56**

"I Need To" drop-down lists

"I Need To" drop-down lists, 354–355

ICollectionEx interface

members of, 92–94
testing, 95–96

IconLocation property (WshShortcut object), 404

icons in task dialogs, 56–57

ID, determining current process, 28

Dispatch

interface, 387
pointer, 389

IIF VBA function, 211–213, 661

ILog

class, creating instance of, 134–135

interface, creating, 130

images

displaying with attachment fields, 377–378

displaying with paths, 378–379

finding in Office 2007, 443–444

getImage and getItemImage callbacks, 465–468

global image handler, creating, 463–465

imageMso attribute, 463

included with Office, 462–463
loading from attachment fields, 468–469

loading from external files, 463

Immediate window (debugging)

clearing, 652

commands in, 651

uses for, 109–110

implementation inheritance, 81

Implements keyword, 81

implicit method, 73

IN (item1, item2, ... , itemn)

operator, 207

IN clauses, 230–231

IN keyword, 215

IN operator, 215–216, 660–661

IN predicate, 254

indexes

adding to tables, 258

and bulk updates, 662

and Find methods, 263–264

multi-field, 267

query performance and, 658–659

removing from tables, 259

single field vs. multi-field, 659

InetIsOffline function, 27

inheritance in VBA, 81

initialization

Initialize event, 70, 72–73
of objects, 642–643

inline error handling, 123–124

inner joins (equi-joins), 219–220

Input/Output statements (VBA)

error handling, 148–149
Open statement, 149–155
opening files for random access, 151–155
overview of, 148
reading existing text files, 149–151
writing data to text files, 155–157

insert-into query. See **appending**

installing

files from attachment fields, 579–581

Office applications, 409–412

instances

comparing class, 648–649
Instancing property, 74–76

InStr VBA function, 165, 200

instrumentation of code, 112–113

Intellisense

class modules and, 71
displaying class events with, 97
and early binding, 584
finding constants with, 124
in Ribbon customization, 442–445

interfaces, VBA

of derived types, exposing, 101
fundamentals, 80–85
ICollectionEx interface, 92–96
implementing, 81–82
interface inheritance, 81
using, 82–83
writing, 81

Internet

connection (Shell function), 27

Internet Information Services (IIS), 592

Internet Movie Database (IMDB), 279

Internet Explorer 7, automating

parsing HTML from, 435–440
viewing Web pages in multiple tabs, 432–433

Web queries, 433–434

Is operator (VBA), 85

ISAMs (Installable Sequential

Access Method drivers), 230

IsEmpty function (VBA), 85

IsNothing function, 85, 585

IsNull VBA function, 211–212

IsNumeric VBA function, 85, 282

IsObject VBA function, 85

IsTrusted property, 529

IsUserAnAdmin API function, 27

IsValidData function, 302–303

Item method (VBA Collection object), 86

ItemSize attribute (menus), 459

Iterator pattern (Gang of Four), 667

J

Jet database engine, Microsoft, 193–194, 261

Johnson, Ralph, 666

joins, table. See **table joins**

K

keyboards

keyboard-driven navigation, 355
selecting list box items with, 319

KeyCodeConstants module, 306

KeyDown event, 305–306, 355

KeyDown property, 623

KeyPress event, 306

KeyPreview property (forms), 306, 623

keys

alpha-numeric for registration, 591

key codes, 306

numeric for registration, 590
testing generation of (registration), 591

keywords

ASC, 258

ByRef, 168

DESC, 258

event, 97

Implements, 81

IN, 215

Me, 70

New, 87

ParamArray keyword (VBA), 170–171, 671

RaiseEvent, 97–98

UNIQUE, 258

WHERE, 205

WithEvents, 76–77

kiosk forms, 351–352

- L**
- labels boxes in Ribbons**, 458
language packs, 500
LanguageSettings object, 494
LastDLLError property, 18–19
late binding
 application dependencies example, 584–585
 defined, 7–8
 vs. early binding, 389–390
LaunchHtmlPage function, 628–630
LCIDs (locale identifiers), 494
left outer joins, 220–222
Left VBA function, 165
Left\$ VBA function, 167, 200
library, checksum, 278
licensing applications
 limited-use applications. See limited-use applications
 registration. See registering applications
LIKE operator, 207, 272
limited-use applications
 expiration dates, 586–588
 overview of, 585
 restricting launches, 589–590
 restricting records, 588
line-continuation character (code), 138
list boxes
 deleting items in, 317–318
 multi-select, selection with keyboard, 319
 multi-select for filter criteria, 318–319
 up/down movement in, 311–316
loading
 images from attachment fields, 468–469
 images from external files, 463
 Load events (forms), 80
 LoadCustomUI method (Access 2007), 448–449
 loadImage callback attribute, 463–465
 LoadLibrary API function, 504–506
 LoadString API function, 504–506
 preventing startup, 443
 Ribbon customizations, 446–449
localization of applications
 changing user locale, 493–494
 determining locale with API functions, 492–493
 Office, determining current language of, 494
 overview, 491
 reading resources with LoadString API, 504–506
 resource DLLs, creating, 501–504
 resource DLLs, deploying, 504
 string tables for localized text, 496–501
 system locale settings, 492
 user locale settings, 492
 VBA global application functions, 495–496
Locals window (debugging)
 displaying, 652
 uses for, 110
locking
 computers, 548–551
 LockWorkstation API, 548
 records based on logon, 552–553
logging errors
 creating instance of ILog class, 134–135
 to database tables, 130–132
 error handler class and, 135
 ILog interface, 130
 overview, 129
 to text files, 132–134
login
 forms, creating, 536–544
 logon names, user, 547
LogMessages property, 260
m_LogType variable, 135
loops
 looping through dates, 645–646
 retry, 649–650
 timeout, 650–651
Luhn algorithm. See also Mod-10 checksums, 280
- M**
- macros**
 in Disabled Mode, 529
 security settings, 530
mailto protocol, 60
Make Table queries, 239–240
managed code
 basics, 5–6
 referencing Access object model from, 7
 writing libraries for Access, 6–7
manifest files, modifying, 35–36
maps, navigating to, 365–366
matching patterns. See pattern matching
MD5 algorithm, 279
MDB file format, 527
MDE files, 557
Me keyword, 70
members, class module. See class module members
Members, Show Hidden, 390
memos
 fields, sorting data on, 218
 Memo data type, 285
menus
 dynamic in Ribbons, 462
 forms. See forms
 managing filters with dynamic, 477–481
 Office menu modification, 484–485
 in Ribbons, 458–460
message box replacement example, 37–39
metadata properties, 70
methods
 CallByName to invoke, 640
 ErrorHandler class, 128–129
 fundamentals, 70–71
 ILog interface, 130
 Move methods, 268–271
 similar to control callbacks, 451
Microsoft. See also Office applications (Microsoft)
 Excel. See Excel, Microsoft, automating
 Office 12.0 Access Database Engine Object Library, 561
 Office 12.0 Object Library, 494
 Office Access database engine, 263
 Outlook. See Outlook, Microsoft, automating
 Shell Controls and Automation library, 392
 Visual SourceSafe (VSS) version control system, 139
 Word. See Word, Microsoft
 Word Object Library, 412–413
Mid VBA function, 165, 421
Mod-10 checksums
 fundamentals of, 279–280
 using VBA to calculate, 281–285
modifier keys (form events), 305–306
modules
 headers (comments), 136
 refactoring code, 139–140

MonthName VBA function

MonthName VBA function, 495

Move methods, 268–271

MoveNext method, 150

MsgBox function, 37–38

multi-valued fields, defined, 377, 670

MVF (multi-valued lookup) field, 289–293

MZ-Tools, 136

N

naming

conventions for code, 138–139
field names. See field names
interfaces, 81
splitting names, 178–180

navigation

form (Ribbons), 472–477
“I Need To” drop-down lists, 354–355

keyboard-driven, 355

to maps dynamically, 365–366
navigation bars, 352–353

navigation pane

customizing, 533–536
Disable Design View shortcuts option, 539
restricting view of, 536–539

nested joins, 222–224

.NET

Framework/references, 6–7

network traffic, minimizing, 662–663

New keyword, 87

NewPassword method, 554

NoMatch property, 265, 268

North American Numbering Plan, 182

Northwind Traders sample database, 194

notifications, computer locking, 548–551

NotInList event, 470–472

Null vs. vbNullString, 210

numbers

formatting as hexadecimal, 671
updating version, 567–568

numeric keys for registration, 590

numeric values for primary key fields, 658

Nz (Null-to-Zero) VBA function, 167, 212–213

O

objects

backup of, 653–654
defined, 5
destruction pattern, 668
form, password-protected, 545–546
global object properties, 644
helper functions, 641–644
initialization of, 642–643
instantiation pattern, 668
Object Browser (VB), 389–391
Object variable, 100
Object-oriented programming (OOP). See OOP (Object-oriented programming)
Ribbon, 449
termination of, 643–644
testing for type, 83–85

ObjPtr VBA function, 37, 70, 649, 671

Office 2007

Color scheme option, 506–507, 518
determining current language of, 494

Office applications (Microsoft). See also individual applications

adding references to, 8–9
automating, 409–432
determining installations, 409–412
Excel chart with data, creating, 421–429

images included with, 462–463

Office menu modification, 484–485

Office Ribbon Developer Center, 444

Office Trust Center, 529–531

Outlook appointments from events database, 429–432

Word document with Access data, creating, 412–421

onAction attribute, 446, 450

one-to-many table joins, 659

OOP (Object-oriented programming), 4–5

Open statement (VBA), 149–155

OpenFolder method, 392

OpenProcess function, 29

operators

< (less than), 660–661

> (greater than), 660–661

BETWEEN . . . AND, 660–661

AddressOf operator (VBA 6), 550

ALIKE (ANSI-Like), 261

BETWEEN...AND, 209–210

comparison, 206–207

IN, 215–216, 660–661

Is (VBA), 85

LIKE, 207–208

And Not, 66–67

Or, 66

Seek method, 266–268

TypeOf (VBA), 83–84

OptionExists method, 509–510

options, application. See application options

Or operator, 66

ORDER BY clauses

indexing fields in, 659

sorting data with, 216–218

outer joins, 220–222

Outlook, Microsoft, automating, 429–432

P

packages, signed, 531–532

Paint event, 368–370

ParamArray keyword (VBA), 170–171, 671

parameters

accidental, 230

declaration, 228–229

form and report, 228

prompting for data with SQL, 227

parentheses () to group conditions, 206

parsing

HTML from IE, 435–440

ParseAddress function, 366

text file into table (example), 174–178

text into data types, 171–174

Partition function, 249–250

Pass-Through queries, 259–260

password-protecting objects

ALTER DATABASE PASSWORD statement, 259

form object, creating, 545–546

login form, creating, 540–544

prompting with forms and reports, 544–545

user table, creating, 540

paths

database (BuildAccApp example), 563

displaying images with, 378–379

to special folder Shell function, 24–25

- pattern matching**
 with ANSI-92 syntax, 261
 data in defined ranges, 209–210
 on dates, 214
 on GUID values, 214–215
 to items in lists, 215–216
 on null and empty string values, 210–214
 on string patterns with wildcard characters, 207–209
 on strings, 207–209
- patterns**
 Attachment fields, 670
 creational, 667–668
 destruction, 668
 Equals method, 669
 Factory, 667
 formatting numbers as hexadecimal, 671
 Gang of Four, 666
 Iterator, 667
 matching. See pattern matching
 object destruction, 668
 object instantiation, 668
 overview, 665–666
 Prototype, 666
 reasons for using, 666
 recordset destruction, 669
 recordset instantiation pattern, 668
 seek, 267–268, 669–670
 sources for, 671
 strings, 670–671
 token replacement, 670–671
 VBA object, 667–669
- PeekToken VBA function, 166–167**
- PeelToken VBA function, 167–169**
- percent sign (%) ANSI wildcard, 261**
- Performance Analyzer, Access, 663**
- Permission Denied error, 148, 151**
- personalization**
 basics, 527
 personally identifiable information, 526
- phone number granularity, 182**
- P/Invoke (platform invoke), 12**
- pivot queries. See crosstab queries**
- Platform SDK, 12**
- Pop method, (stacks), 111–113**
- Portland Pattern Repository, 671**
- postal codes, 181**
- predicates**
 ALL, 231
 DISTINCT, 233–234
 IN, 254
- selection, 231–234
 TOP, 231–233
- previewing reports, 382–383**
- primary keys**
 fields, numeric values for, 658
 Primary Key Indexes, 658
 setting on tables, 256–257
- printers**
 adding, 406–407
 enumerating, 408–409
 removing, 408
- printing**
 bulk reports, 384–385
 reports, 382–383
- private conditional compilation constants, 116**
- private data (BuildAccApp example), 561–562**
- Private events, 96**
- Private variables, 68–69**
- procedure headers (comments), 136–137**
- procedures, refactoring code, 140–142**
- processes and threads**
 awaiting process completion, 32–34
 defined, 28
 determining current process ID, 28
 pausing code execution, 28
 quitting processes, 29–32
- ProgID (programmatic identifier)**
 for automation servers, 389
 values for Office applications, 8–9
- progress bars, custom, 319–321**
- properties**
 accepting partial SQL strings, 194
 adding to ErrorHandler class, 126–128
 build (BuildAccApp example), 563–564, 574–577
 DAO, 614
 debug builds with database properties, 122–123
 fundamentals, 68–70
 global object, 644
 Instancing, 74–76
 keywords, 68
 metadata, 70
 partial SQL, 194
 property dialog box Shell function, 25–27
 read-only, 69
- setting form (BuildAccApp example), 566–567
 setting form properties (BuildAccApp example), 566–567
- similar to control callbacks, 451
 values, retrieving, 638–639
 values, setting, 639
 write-only, 69
- Prototype pattern (Gang of Four), 666**
- public conditional compilation constants, 116–117**
- Public events, 96**
- Push method (stacks), 111–113**

Q

- queries, SQL**
 action, 238–239
 Append, 240–243
 automating Web, 433–434
 bulk updates and, 662
 calculated fields, avoiding, 661
 creating multi-record APPEND, 241–243
 crosstab, 246–250
 Data Definition, 254–259
 database maintenance and, 663
 DELETE, 245–246
 expressions used with crosstab, 249
 fixed column headings in CROSSTAB, 661
 hard drive maintenance and, 663
 Make Table, 239–240
 network traffic and, 662–663
 Pass-Through, 259–260
 pivot queries. See crosstab queries
 reducing data to be processed, 659–660
SELECT. See SELECT queries
 stored queries vs. recordset properties, 662
 subqueries, 253–254
 transform. See crosstab queries
 types of, 195–196
 UNION, 251–253
 Update, 243–245
- Query Designer, Access**
 multi-record APPEND query with, 241–243
 query types created by, 195–196

Query Designer (*continued*)

Query Designer, Access (*continued*)

single-record APPEND query with, 243
table name aliases, adding with, 205
UNION queries with, 253
update queries with, 244–245
Query Optimizer, 657–658
Quick Watch, adding, 106–107
quotation marks ("") for delimiting strings, 207

R

radio buttons in task dialogs, 50–52

RaiseEvent keyword, 97–98

random access to files, 151–155

read-only properties, 69

RecordCount property, 292–293

records
appending multiple with SELECT queries, 241
appending with value lists, 243
limiting number of, 588
locking based on logon, 552–553
multiple, 670
multi-record APPEND queries, 241–243
record level security, simulating, 551–553
searching with attachments, 292–294
single-record APPEND query, 243

recordsets

destruction pattern, 669
form and report, 194
instantiation pattern, 668
SELECT query and, 196
RecordSource property, 194
refactoring code, 139–141, 637
ReferenceEquals method, 648–649
references
adding (BuildAccApp example), 560–561
adding to Office applications, 8–9
finding with References dialog box, 8
overview, 7–8
testing reference fix-up, 583–584
updating, 581–582
REG_EXPAND_SZ type values, 398
REG_MULTI_SZ type values, 398–400

RegDelete method (WshShell object), 398

Regional and Language Options dialog box (Vista), 493–494

Regional and Language Options (Vista), 493–494

RegisterDatabase method, 608

registering applications

alpha-numeric keys, 591
numeric keys, 590
registration Web service, 592–596
testing key generation, 591

registry, Windows

and application expiration dates, 586–588
determining Office installations and, 408
managing application options in, 509–513
reading and writing, 395–400
registry keys, 590
RegRead method (WshShell object), 397
RegWrite method (WshShell object), 395–396
storing application options in, 508–509

Remove method (VBA Collection object), 87

RemovePrinterConnection method, 408

Replace VBA function, 169–170

report manager

bulk printing, 384–385
categorization of reports, 382
e-mailing reports, 382–383
overview, 379–380
previewing reports, 382–383
printing reports, 382–383
reports table, creating, 380–381
tracking report usage, 384
user custom reports, 385–386

reports

calendar, creating, 370–376
ControlTipText property, 620–621
creating with data bars, 366–370
displaying images with attachment fields, 377–378
displaying images with paths, 378–379
drill-down, 363–365
filtering with controls, 360–363
navigating to maps dynamically, 365–366

parameters, 228

password prompting with, 544–545

recordsets overview, 194

report manager. See report manager

Report view (Access 2007), 357

ReportCategory fields, 382

sorting with controls on, 358–359

subclassing, 76–77

repurposing controls, 485–486

resource DLLs

creating, 501–504

deploying, 504

resources, reading with LoadString API, 504–506

restoration of data, 276–278

retrieve command line function, 21–23

retry loops, 649–650

return computer name function, 21

return user name function, 19–20

ReturnsRecords property, 260

reviews, code, 142

RHS (right-hand side) argument, 89

Ribbon customizations

buttons, 455

callback routines for, 449–452

check boxes, 456–457

combo boxes, 457–458

contextual tabs, 453–454

dialog box launchers, 460–461

displaying options in, 516–518

drop-down controls, 457–458

dynamic menus, 462

error display in development, 442

filters using dynamic menus,

477–481

finding controls and images, 443–444

form navigation, 472–477

galleries, 461

images. See images

Intellisense usage, 442–443

labels and edit boxes, 458

loading, 446–449

menus, 458–460

NotInList event, 470–472

Office menu modification, 484–485

Office Ribbon Developer

Center, 444

overriding commands, 485–486

overview, 441–442

preventing startup loading, 443

refreshing controls, 455

- repurposing controls, 485–486
 split buttons, 462, 482–484
 tabs, 453
 toggle buttons, 455–456
 writing, 444–445
- right outer joins, 220–222**
- Right VBA function, 165**
- ROT (running object table), 389**
- Run method (Application.Run), 641**
- S**
- SaveAsText method (Access Application object), 653**
- SaveSetting statement (VBA), 508**
- screens, splash, 349–351**
- Scripting Host, Windows. See Windows Scripting Host**
- searching**
 Find methods, 264–266
 optimization of, 271–272
 for records with attachments, 292–294
 Searches variable, 87–88
 Seek method, 266–268
 for specific attachments, 294–296
- security**
 ACCDR files, 557
 AllowBypassKey property, 556–557
 authentication, 526–527
 authorization, 526–527
 database encryption, 527–528
 database passwords, adding with DAO, 554–555
 database passwords, changing with ADO, 555–556
 database passwords, changing with DAO, 553–554
 digital signatures, 531
 disabled mode, 528–529
 encrypted databases with DAO, 553
 fundamentals, 525–526
 locking computers, 548–551
 logon names, determining, 547
 macro settings, 530
 Office Trust Center, 529–531
 password-protecting objects. See password-protecting objects
 protecting data with client-server database, 556
 protection with ACCDE or MDE files, 557
 Security Account Manager (SAM), 526
- selecting file formats, 557–558
 signed packages, 531–532
 simulating record level, 551–553
 simulating Windows integration, 546–551
 Trusted Locations page, 531
- Seek method (recordset object), 266–268, 669–670**
- Select Case block, 151**
- SELECT queries**
 appending multiple records with, 241
 FROM clauses, 203–205
 field lists. See field lists
 ORDER BY clauses, 216–218
 overview, 196–197
 WHERE clauses. See WHERE clauses
- selection predicates, 231–234**
- self joins (self-referential joins), 224–225**
- self-signed certificates, creating, 532**
- SendError method (error handler class), 128–129**
- SendKeys VBA function, 652**
- SendMessage function, 29**
- separators, menu, 460**
- servers, automation, 389, 391–392**
- SetObjectProperty helper method, 616**
- Shell functions**
 basics, 388
 ShellExecute API function, 629–630
 ShellExecute function, 59–60
 ShellExecute method, 392–395
 ShellExecuteEx API function, 25–27
 Windows, 21–23
- Shell objects, Windows, 392–395**
- SHGetFolderPath API function, 24**
- Shift argument (KeyDown event), 306**
- shortcuts, creating desktop, 402–404**
- Show Hidden Members function, 390**
- signatures, digital, 531**
- signed packages, 531–532**
- SimpleTaskDialog function, 38–39**
- Sleep function, 29**
- sorting**
 reports with controls, 358–359
 UNION queries, 252–253
- sorting data**
 on binary or memo fields, 218
 custom sort order fields, 273–276
 ORDER BY clause and, 216–218
 setting direction of, 216–218
- spaces in databases, 201**
- special folders**
 determining with Windows Scripting Host, 400–401
 path function, 24–25
- specialization, interface, 83**
- splash screens, 349–351**
- split button controls, 462, 482–484**
- splitting strings, 164–182**
 address element granularity, 180–181
 formatted token replacement, 170–171
 names, splitting, 178–180
 parsing text file into table (example), 174–178
 parsing text into data types, 171–174
- PeekToken function (VBA), 166–167
 PeekToken function (VBA), 167–169
 phone number granularity, 182
 Replace function (VBA), 169–170
 replacing token in strings, 169–170
 Split function (VBA), 164–165
 string handling functions overview, 165–166
 StringFormat function (VBA), 170–171
 VBA data type conversion functions, 171–174
- SpyXX tool, 31**
- SQL (Structured Query Language)**
 action queries, 238–239
 adding data from different database formats, 230–231
 Aggregate functions, 234–236
 ANSI-89 and ANSI-92 syntax, 261
 Append queries, 240–243
 IN clauses, 230–231
 crosstab queries, 246–250
 Data Definition queries, 254–259
 data types used with DDL, 255–256
 DELETE queries, 245–246
 GROUP BY clauses, 234–238
 HAVING clauses, 237–238
 Make Table queries, 239–240
 parameters, prompting for data with, 227–230

SQL (Structured Query Language) (continued)

SQL (Structured Query Language) (continued)

Pass-Through queries, 259–260

properties accepting partial SQL strings, 194

queries. See queries, SQL

selection predicates, 231–234

server, 551, 556, 592

SQLConfigDataSource API function, 608

subqueries, 253–254

table joins. See table joins

UNION queries, 251–253

Update queries, 243–245

use in Access, 194

square brackets ([])

with LIKE operator, 207–208

in SQL code, 200–201

stacks

Call Stacks, 110–115

defined, 111

StackTrace property (.NET Framework), 111

startup loading, preventing, 443

Static variables, 368

StatusBarText property (form and report controls), 619–620

storage

stored queries vs. recordset properties, 662

and VBA Collection object, 86

string handling functions, VBA

overview, 165–166

PeekToken function, 166–167

PeelToken function, 167–169

StringFormat function, 154, 170–171, 188, 625–627

strings

in C and C++, 15–17

pattern matching with ANSI-92 syntax, 261

patterns, 670–671

replacing tokens in, 169–170

retrieving localized strings from tables, 497–501

splitting. See splitting strings

String class (.NET Framework), 670–671

variant vs. string function versions, 170

StrPtr function, 36–37

structures in C and C++, 17

subclassing

forms and reports, 76–80

windows, 548

subform controls, reusing, 310–311

subqueries, 253–254

SysCmd function, 319, 574

SysCmd method, 619–620

system database (MDW) files, 527

System DSN, 607–610

system error codes, 124–125

system information, retrieving, 19–23

system locale settings, 492

T

table joins

Cartesian products, 225–227

indexes and, 659

inner joins, 219–220

nested joins, 222–224

one-to-many joins, 659

outer joins, 220–222

overview, 218–219

self joins, 224–225

table of contents (HTML Help files), 633–634

tables

adding data from multiple. See table joins

adding indexes to, 258

ALTER TABLE statement, 257–258

choosing with FROM clauses, 203–205

CREATE TABLE statement, 255

creating debug builds with, 121–122

dashboard (examples), 333–335

deleting data from (BuildAccApp example), 577

designing for performance, 658–659

DROP TABLE statement, 258

joining multiple with nested joins, 222–224

list of (BuildAccApp example), 564–565

logging errors to database, 130–132

managing application options from, 509–510

name aliases, 204–205

parsing text files into (example), 174–178

re-linking automatically, 604–607

removing indexes from, 259

reports tables, creating, 380–381

setting primary keys on, 256–257

storing application options in, 508

storing help text in, 621–622

storing theme data in, 519

string tables for localized text, 496–501

user name/password tables, 540

tabs

in Ribbon customizations, 453

viewing Web pages in multiple, 432–433

Tag property (Access objects), 514, 620

TargetPath property (WshShortcut object), 403

task dialogs

message box replacement

example, 37–39

TaskDialog and TaskDialogIndirect functions, 40

TaskDialog class, 64, 68

TASKDIALOG_SYSTEM_ICONS, 41

task dialogs, Windows Vista, 35–57

command links, 46–48

custom button text, setting, 44–46

expanded section feature, 40–43

footer section feature, 43–44

hyperlinks, adding, 52–56

icons, 56–57

overview, 35–36

radio buttons, adding, 50–52

StrPtr function, 36–37

TaskDialog and TaskDialogIndirect functions, 37

verification text, adding, 48–50

temp folders and files, 401–402

templates, HTML, 184

termination

of objects, 643–644

Terminate event, 73

TerminateProcess API function, 29

testing

ICollectionEx interface, 95–96

key generation (registration), 591

objects for type, 83–85

reference fix-up, 583–584

routines, 34

versioning server, 604

text

storing help text in tables,

621–622

Text Format property, 364

verification in task dialogs, 48–50

text files

fixed-width, 148

logging errors to, 132–134

- opening for random access, 151–155
parsing into tables (example), 174–178
reading existing (Open statement), 149–151
writing data to, 155–157
- Thawte certificate authority**, 531
- themes for forms**
applying at runtime, 520–523
storing theme data in tables, 519
- threads and processes**. See **processes and threads**
- timeouts, adding to code blocks**, 650–651
- Timer event**
backups with, 306–307
to close splash screens, 350–351
- Timer VBA function**, 650–651
- tbexp.exe tool**, 7
- toggle buttons in Ribbons**, 455–456
- tokens**
formatted replacement of, 170–171
HTML template, 184–187
PeekToken function, 166–167
PeelToken function, 167–169
replacement patterns, 670–671
replacing in strings, 169–170
- tooltips**, 308–309
- TOP predicate**, 231–233
- transform queries**. See **crosstab queries**
- Trim\$ VBA function**, 167, 213–214
- Tristate enumeration values**, 158
- Trusted Locations page**, 531
- typedef struct keywords**, 17
- TypeName VBA function**, 84–85
- TypeOf operator (VBA)**, 83–84
- U**
- underscore (_) ANSI wildcard**, 261
- Undo event**, 304
- Unicode file formats**, 160–164
- unions**
defined, 17
UNION queries, 251–253
- unique indexes**, 659
- UNIQUE keyword**, 258
- unmanaged code**, defined, 6
- Update queries**, 243–245
- updating applications**
bulk updates, 662
checking for latest version, 596
- client application, creating, 603–604
re-linking tables automatically, 604–607
version numbers (BuildAccApp example), 567–568
versioning server, creating, 596–602
versioning server, testing, 604
- up/down buttons (list boxes)**
creating, 311–316
custom, 321–322
- URLs, opening in new tabs (IE7)**, 432–433
- users**
creating user DSN, 607–608
determining logon names, 547
limiting views to, 551–552
name and password tables, 540
user as administrator Shell function, 27
user locale, changing, 493–494
user locale, settings, 492
user name return function, 19–20
- USys prefix (User System object)**, 622
- USysErrorLog table**, 130
- USysRibbons table example**, 447
- UTF-8 file formats**, 160–164
- V**
- validating**
control data, 309–310
form data, 302–304
registration with Web service, 595–596
- value lists, appending records with**, 243
- variables**
m_LogType variable, 135
Object variable, 100
Private variables, 68–69
Searches variable, 87–88
- variant vs. string function versions**, 170
- VarPtr pointer function**, 37
- VBA (Visual Basic for Applications)**. *See also class modules, VBA; individual function names*
background, 3–6
calculate Mod-10 checksums with, 281–285
CallByName VBA function, 637–640
Code Commenter, 136
- code reuse in, 64, 73–76
data type conversion functions, 171–174
data types, C conversions to, 14–18
Extensibility 5.3 object model, 652, 654
functions for global applications, 495–496
inheritance in, 81
Input/Output statement.
 See Input/Output statements (VBA)
object patterns, 667–669
and OOP 63
string handling functions.
 See string handling functions, VBA
variant vs. string function versions, 170
- vbAppWinStyle enumerated constants**, 388
- VBE (Visual Basic Editor)**, 106
- VB.NET language**, 5–6
- vbNullChar constant**, 609
- vbVerticalTab characters**, 417
- verification text in task dialogs**, 48–50
- VeriSign**, 531
- version numbers**
calculating (BuildAccApp example), 577–578
updating (BuildAccApp example), 567–568
- versions, application**
checking for latest, 596
version control systems, 139
version resources, 503
versioning server, creating, 596–602
versioning server, testing, 604
- VSS (Visual SourceSafe) version control system**, 139
- views**
limiting to specific users, 551–552
restricting navigation pane, 536–539
- visibility, controlling with class modules**, 64–65
- Vista, Windows**. *See Windows Vista*
- Visual Basic Editor (VBE)**. *See VBE (Visual Basic Editor)*
- Visual C++ 2005 Express Edition**, 501

Visual Studio

BuildAccApp example, 560
versions of, 6
Visual Studio 2005, 442–443,
501–504

Vlissides, John, 666

VSS (Visual SourceSafe) version control system, 139

W

WaitForSingleObject function, 33–34

Watches window (debugging)

basics, 106–109
displaying, 652

Web browsers, opening default, 60

Web pages

HTML-based help, 625–631
viewing in multiple tabs, 432–433

Web queries, automating in IE 7, 433–434

Web service, registration, 591–596

Web sites, for downloading

HTML Help Workshop, 631
MZ-Tools (module headers), 136
Northwind Traders sample database, 194

sample code for this book, 325

Web sites, for further information

API resources, 12
Checksum algorithms, 280
connection and template strings, 231
hash functions, 279
Internet Movie Database (IMDB), 279
Luhn algorithm, 280
MD5 hash function, 279
Microsoft Excel Object Model, 422
Microsoft Outlook Object Model, 430
Microsoft Word object model, 413
Office Ribbon Developer Center, 444
patterns, 671
U.S. Postal Service, 181
Windows API documentation, 13
Windows security, 526

WeekdayName VBA function, 495

WHERE clauses

comparison operators, 206–207
vs. HAVING clauses, 237–238

indexing fields in, 659
matching data in defined ranges, 209–210
matching on dates, 214
matching on GUID values, 214–215
matching on null and empty string values, 210–214
matching on strings, 207–209
matching to items in lists, 215–216
Null vs. vbNullString, 210
IN operator, 215–216
overview, 205–206

WHERE keyword, 205

wide characters, defined, 13

Wide Strings, 16–17

wildcard characters

adding fields with, 198–199
ANSI, 209, 261
pattern matching on strings with, 207–209
usage with LIKE operator, 207–208

windows

class names, 31
closing code windows, 654–655
window procedure (wndproc), 548

Windows, Microsoft

automating, 391–409
registry. See registry, Windows security integration, 547–551
Shell functions, 21–23
Shell objects, 392–395
Windows Scripting Host. See Windows Scripting Host

Windows Shell objects, 392–395

Windows API

API functions, 12, 14
API resources, 12
controlling processes and threads.
See processes and threads
Declare statements, writing, 13–18
error messages from API functions, 18–19
GUIDs, creating, 58–59
programming, 5
registry API functions, 508–509
retrieving system information, 19–23
ShellExecute function, 59–60
Windows Shell functions, 21–23
Windows Vista task dialogs. See task dialogs, Windows Vista

Windows Scripting Host

desktop shortcuts, creating, 402–404
printers. See printers
registry, reading and writing to, 395–400
Script Host Object Model, 157–158
special folders, determining, 400–401
temp folders, temp files in, 401–402
WshShell object for process completion, 404–406

Windows Vista

control tips, 621
creating system DSN in, 610
defragmentation tool, 663
Digital Certificates for VBA projects, 532
IsUserAnAdmin API function in, 27 and .NET Framework, 6
Regional and Language Options dialog box, 493–494
task dialogs. See task dialogs, Windows Vista

WithEvents keyword, 76–77

Word, Microsoft

automating, 412–421
creating document with Access data, 412–421

WorkingDirectory property (WshShortcut object), 404

wrapper method, 74–75

write-only properties, 69

WshCollection object, 408

WshShell object

awaiting process completion with, 404–406
CreateShortcut method, 402–403
methods of, 395–398
SpecialFolders strings, 400–401

WshShortcut object, 402–404

X

XML schema for Ribbons, 442–443

XMLHTTP object, 595

Y

Year VBA function, 200



Take your library wherever you go.

Now you can access more than 200 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

