

F. Scott Barker's Access 2002 Power Programming

SAMS



"Scott knows development! I have worked with Scott for the last 10 years. His talent, knowledge, and passion for Access and Visual Basic are reflected in his writing."

Meng Phua

Group Program Manager
Microsoft® Office Developer Team
Microsoft®

"Scott is a seasoned developer and an outstanding trainer, and his experience and expertise really shine through in this book."

Stan Leszynski

Author and Developer
www.1dfinfo.com

I am pleased to see Scott continuing to update and add to his *Access Power Programming* books. I can't count the number of times Scott has personally helped with nagging Access programming issues, and I'm pleased to see he's continuing to share his expertise with his many readers.

Over the years Access has increased in both depth and scope. Access is now used to develop applications only dreamed of a few years ago. It's a rare developer who doesn't need help now and then to master new Access technologies or to respond to an application's requirements. I'm sure you'll find *F. Scott Barker's Microsoft Access 2002 Power Programming* an indispensable aid to your programming projects. There are very few experts I trust more than Scott.

Michael Groh

Technical editor, *Access/VB/SQL Server Advisor Magazine*

Way back when I started programming Access, I bought and read Scott Barker's *Access 95 Power Programming* book. It was truly one of the books that turned me into a real programmer. Since then Scott has continued to provide authoritative information and excellent "real-life" examples in every new edition of his book. *F. Scott Barker's Microsoft Access 2002 Power Programming* is no exception. This is a must-have for any Access programmer interested in improving his programming abilities and saving huge amounts of time developing Access 2002 applications.

Tom Howe

Chief Technology Officer, Fios, Inc.

Co-author of *Access 2000 Development Unleashed*

If you intend to become an Access database developer, *F. Scott Barker's Microsoft Access 2002 Power Programming* is your best choice for an advanced tutorial and reference. Scott, an accomplished Access developer, delivers detailed coverage of all the new and upgraded Access 2002 features: XML export/import, Data Access Pages, and client/server applications with SQL Server (MSDE) 2000. Even if you take advantage of only a few of Scott's programming tips and techniques, your investment in this book will repay itself many times over.

Roger Jennings

Author of *Special Edition Using Access 2002*

and *Special Edition Using Windows 2000 Server*

F. Scott Barker's Microsoft® Access 2002 Power Programming

F. Scott Barker

SAMS

800 East 96th St., Indianapolis, Indiana, 46240 USA

F. Scott Barker's Microsoft Access 2002 Power Programming

Copyright © 2002 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32102-5

Library of Congress Catalog Card Number: 00-109715

Printed in the United States of America

First Printing: September 2001

04 03 02 01 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the code or programs accompanying it.

EXECUTIVE EDITOR

Rosemarie Graham

ACQUISITIONS EDITOR

Angela C. Kozłowski

DEVELOPMENT EDITOR

Susan Shaw Dunn

MANAGING EDITOR

Charlotte Clapp

PROJECT EDITORS

Elizabeth Finney

Debra Sexton

COPY EDITOR

Rhonda Tinch-Mize

INDEXER

Eric Schroeder

PROOFREADER

Linda Seifert

Plan-it Publishing

TECHNICAL EDITORS

Pai Chung

Pete Klein

TEAM COORDINATOR

Lynne Williams

MEDIA DEVELOPER

Dan Scherf

INTERIOR DESIGNER

Anne Jones

COVER DESIGNER

Aren Howell

Overview

Introduction 1

Part I The Root of Power Programming

- 1 Macros Are for Weenies; Code Is Cool! 9
- 2 Coding in Access 2002 with VBA 21
- 3 Making Access Project and Data Technology Choices 71
- 4 Working with Access Collections and Objects 79
- 5 Introducing ActiveX Data Objects 109
- 6 Using XML with Access 2002 133
- 7 Handling Your Errors in Access with VBA 147

Part II Manipulating and Presenting Data

- 8 Using Queries to Get the Most Out of Your Data 187
- 9 Creating Powerful Forms 241
- 10 Expanding the Power of Your Forms with Controls 265
- 11 Creating Powerful Reports 313
- 12 Working with Data Access Pages 355

Part III Extending Access with Interoperability

- 13 Driving Office Applications with Automation 389
- 14 Programming for Power with ActiveX Controls 423
- 15 Extending the Power of Access with API Calls 461
- 16 Extending Your VBA Library Power with Class Modules and Collections 491
- 17 Creating Your Own Wizards and Add-ins 511
- 18 Manipulating the Registry with VBA 543
- 19 Using Access with the Internet 573

Part IV Managing Databases

- 20 Securing Your Application 597
- 21 Handling Multiuser Situations 651
- 22 Welcome to the World of Database Replication 691
- 23 Moving Workgroup Applications to Client/Server 741
- 24 Developing SQL Server Projects Using ADPs 779

Part V Adding Finishing Touches

25 Startup Checking System Routines Using ADO **829**

26 Creating Maintenance Routines **861**

Index **889**

Part VI Appendixes on the Website

A Debugging Code in Access 2002

B Getting Started with ActiveX Controls

C Working with Data Access Objects

D Programming Office Command Bars and Other Office Components

E Access 2002 and Jet 4 Errors

Table of Contents

A Letter from the Author 1

Who Is This Book For?	1
What's Covered in This Book?.....	2
Part I: The Root of Power Programming	2
Part II: Manipulating and Presenting Data	3
Part III: Extending Access with Interoperability.....	4
Part IV: Managing Your Databases	4
Part V: Adding Finishing Touches.....	5
Using This Book's Examples	6
Conventions Used in This Book	6

PART I The Root of Power Programming

1 Macros Are for Weenies; Code Is Cool! 9

Understanding Where Macros End and Code Begins	10
When Are Macros Necessary?	10
When Is Code Necessary?.....	11
Looking at Macro-to-Code Changes	12
Using the DoCmd Object	12
Code Equivalents of Macro Commands	14
Converting Existing Macros to VBA Code	15
Summary.....	19

2 Coding in Access 2002 with VBA 21

Getting Started with Programming.....	22
Using Code Modules	23
Declaring Variables	27
Declaring Procedures.....	31
Controlling Program Flow.....	39
Commenting Code.....	42
Handling Errors	42
Getting Started with VBA.....	43
Introducing Objects	44
Using the Object Browser	44
Programming with Objects	47
The Public Keyword	47
The Private Keyword.....	47
Using Properties and Methods	48
Using Existing Properties	48
Using Existing Methods	50
Specifying Named Parameters	51
Assigning Objects to Variables	52

Using Collections	54
Counting the Number of Elements	55
Accessing Elements of the Collection.....	56
Iterating Over Collections	57
Customizing a Form	58
Writing Custom Properties	59
Writing Object-Valued Properties	63
Writing Custom Methods	63
Coding Class Modules	63
Creating the Support Objects	64
Creating the Class Module	65
Using the Class Module	66
Summary	69
3 Making Access Project and Data Technologies Choices 71	
Using Microsoft Database Versus Access Database Project	72
Looking at the Objects Used in Each	73
Using DAO Versus ADO Versus XML.....	75
Summary	77
4 Working with Access Collections and Objects 79	
Creating Custom Collections	80
Defining a New Collection	80
Adding Items to the Collection	81
Removing Items from the Collection	82
Comparing Custom Collections to Arrays	83
Creating a Collection of Integers	83
Creating an Array of Integers	84
Understanding Advanced Uses of Collections	86
Accessing the Access Object Model.....	86
Using the Application Object	86
Looking at the References Collection	99
Specifying and Manipulating Printers.....	99
Working with the Forms, Reports, and Data Access Pages Collections	101
Programming Multiple Copies of the Same Form	104
Supporting Multiple-Form Instances.....	105
Examining the frmEmployees Form's Code	106
Closing the frmEmployees Forms Automatically	107
Summary	108

5 Introducing ActiveX Data Objects 109

Looking at ADO's Object Models	110
The ActiveX Data Objects 2.5 (ADODB) Object Model	111
The ADO Extensions 2.5 for DDL and Security (ADOX)	
Object Model	111
Jet and Replication Objects 2.5 (JRO) Object Model	113
Referencing the Type Libraries.....	113
Opening a Connection to a Database	115
Connecting to the Current Database	115
Connecting to Another Database.....	116
Creating a Recordset.....	116
Opening a Simple Recordset	117
Looping Through and Editing Recordsets	119
Creating Persistent Recordsets	120
Using the RecordCount, BOF, and EOF Properties	122
Checking to See What Operations a Recordset Will Support.....	122
Cloning Recordsets	123
Storing Bookmarks	123
Working with Queries.....	123
Creating a New Query.....	124
Creating a Parameterized Query	124
Opening a Recordset Off a Parameterized Query	124
Executing Bulk Queries.....	125
Modifying an Existing Query	126
Deleting a Query	126
Working with Tables.....	127
Creating a New Table with Fields and Indexes.....	128
Modifying an Existing Table by Adding an Index	130
Summary.....	131

6 Using XML with Access 2002 133

Getting to Know XML	134
Looking at XML's History	134
Examining the Files That Make Up XML Documents.....	135
Working with XML with the Access User Interface	139
Exporting from Access to XML	139
Differences in Exporting Between Access User	
Interface and ADO	141
Importing an XML Document.....	142
Coding with XML and VBA in Access 2002	142
Taking Advantage of the Other Office Applications'	
XML Support	144
Introducing the Excel XML Spreadsheet Schema (XML SS)	144
Summary.....	145

7 Handling Your Errors in Access with VBA 147

Examining Access's Runtime Error Handling	148
Using the On Error Command	149
Using the Exit Sub/Function Command	150
Using Resume, Resume Next, and Resume LineLabel	150
Working with the Err and Error Objects	153
The Err Object's Clear Method	154
The Err Object's Raise Method	154
Working with the ADO Errors Collection	157
Creating User-Defined Errors	161
Using Custom Error Logs to Track Errors	163
An Example Error Handler Calling the Error Log	164
The Actual Error Log Code	165
Creating a Centralized Error-Handling Routine	171
A Last Look at Error-Handling Issues	176
Watching for Environment Changes	176
Using Your Error Handler to Roll Back Transactions	178
Using a Form's On Error Event	179
Nesting Error Handlers	181
Looking at Some New Options for Error Handling	182
Summary	182

PART II Manipulating and Presenting Data**8 Using Queries to Get the Most Out of Your Data 187**

Understanding Where Queries Are Used in Access	188
Using Queries with Form and Report Record Source Properties	189
Giving Users Access to Queries	190
Using Naming Conventions and Query Documentation	191
Working with Select Queries	193
Joining Tables	195
Using the Same Table Twice (Self Joins)	196
Using the Access AutoLookup Feature	198
Working with Action Queries	199
Make Table Queries (SELECT INTO)	199
Append Queries (INSERT INTO)	200
Update Queries (UPDATE..SET)	201
Delete Query (DELETE)	201
Performing Advanced Query Operations	201
Summary Queries	201
Union Queries	203
Nested Queries	204

Subqueries	205
DDL Queries	205
Adding More Power with VBA	205
Building Faster Queries	205
Using Query by Form	206
Creating Temporary Command Objects	215
Using the DoCmd Object's RunSQL Method.....	215
Issuing Parameter Queries	216
Driving Reports and Forms with Queries.....	217
Solving Problems with Queries	217
Grouping to Get Percentages.....	218
Finding and Deleting Duplicate Records	220
Nesting Groups to Get the Complete Solution	221
Distinguishing Between New and Old Records	223
Creating a Total Row for Crosstab Queries	223
Examining the Architecture of the Query Resolution Process	227
Defining the Query	228
Compiling the SQL Statement	228
Preparing the Execution Plan (Optimization)	228
Optimizing Queries with Jet	229
Using Rushmore Technology	229
Examining the Clustered Primary Index	230
Working with Read-Ahead	232
Understanding Optimization Techniques	233
Increasing Performance with Table Relationships	233
Adding Indexes	234
Tweaking the Database Structure to Affect Performance	235
Optimizing Join Performance	236
Using Unconventional Optimization Techniques	236
Understanding Some Performance-Tuning Pitfalls.....	236
Diagnosing Slow Queries	236
Resolving Ambiguous Field References	238
Using the Analyzer Wizards	238
Table Analyzer Wizard	238
Performance Analyzer	239
Database Documentor	239
Looking at Access 2002's New Query Features.....	239
Using ANSI 92 SQL Mode	239
Viewing Data Using PivotTable and PivotChart Views	239
Summary	240

9 Creating Powerful Forms 241

Increasing Form Performance.....	242
Looking at Access 2002's New Form Features	242

Looking at Access 2002's New Form Features	243
New Base Form Events	243
Using the New PivotTable and PivotChart Views	243
Taking Advantage of Other Form Features	245
Using the Form Recordset Property	245
Using the Dirty Event	246
Specifying a Splash Screen Form at Startup.....	247
Using Form Background Properties	248
Reusing Forms to Perform Standard Tasks	251
Increasing Tabbed Form Performance	260
Summary	263

10 Expanding the Power of Your Forms with Controls 265

Setting Up a Field's Lookup Properties for Use on Forms	266
Tapping into the Power of Combo Boxes.....	268
Using the Combo Box Wizard	268
Programming Combo Boxes Beyond the Wizard	270
Using a Union Query to Give the Choice of One or All.....	272
Using a UNION SQL Statement to Requery All in a Subform.....	275
Displaying Combo Box Columns Outside the Control.....	277
Adding New Combo Box Items Based on User Input	280
Working with the Access Tab Control	282
Creating and Editing a New Tab Control	284
Moving Pages in the Tab Control	286
Adding Controls to the Tab Pages.....	286
Using Code with the Access Tab Control	287
Morphing Access Controls	289
Morphing Controls at Design Time.....	289
Morphing Controls with VBA at Runtime	289
Programming Multiselect ListBox Controls.....	292
List Box Properties Dealing with Multiple Selection	293
Manipulating Items Selected in a Multiselect List Box with VBA	294
Getting Relief with the Subform/Subreport Wizard.....	299
Giving Controls Spreadsheet-Type Cursor Movements	300
Looking at the Problem	301
Creating a Solution	301
Manipulating Controls Through Code	304
Examining the Pieces of the Option Group Menu Form	305
Introducing the ManipulatingControlsExample Form	305
Looking at the Code Behind the Form	306
Summary	310

11 Creating Powerful Reports 313

Creating Summary, Detail, and Summary/Detail Reports	
from the Same Report	314
Creating Dynamic Groupings for the Same Report with QBF	320
The Elusive Feature: Creating Snaking Reports.....	323
Looking at the Before Report	325
Working with the After Report	328
Printing Multiple Topics Through a MultiSelect	
List Box	328
Looking at the rptMultiSelectCategoryExample Report.....	329
Looking at the MultiSelect List Box Form	330
Code Listings for the MultiSelect List Box Form	330
Creating a Wizard-Like Interface for Selecting Group-By Items	335
The Core Tables: WizExReports and WizExElements	336
Working with the frmWizExReports Form	338
Using the Group Element Wizard with a New Report	345
Formatting Reports Dynamically	347
Looking at the rptDynamicFormattingExample Report	347
Using Conditional Formatting in Reports	349
Summary.....	354

12 Working with Data Access Pages 355

Why Data Access Pages?	356
Understanding How Data Access Pages Are Structured	356
Understanding the Navigation Control	360
Comparing Data Access Pages to Forms and Reports	362
Understanding What Users Need for Data Access Pages	363
Saving Time with the Data Access Page Wizards	364
Using AutoPage: Columnar.....	364
Taking Off with the Page Wizard	365
Creating and Enhancing Simple Data Access Pages	369
Looking at the Data Access Page Field List	369
Adding Hyperlinks	370
Using Expressions on Data Access Pages	374
Using Bound Combo and List Boxes	374
Formatting with Themes	377
Using Additional Controls on Data Access Pages	378
Grouping Data Access Pages: Reports for the Web	379
Creating the Base Page	379
Creating a Relationship on a Data Access Page	381
Creating Group Levels with Promote	382
Adding a Caption Section	383
Viewing Your Data Hierarchically with Banded Data Access Pages.....	383
Using a Combo for a Group Filter Control.....	383

Finding Additional Resources	384
Summary	385

PART III Extending Access with Interoperability

13 Driving Office Applications with Automation 389

Working with Automation	392
Declaring Object Variables in VBA	394
Using the CreateObject() Function	395
Using the GetObject() Function	396
Cleaning Up When Done with an Object	397
Running Other Applications from Access with Automation	397
Driving Word from Access	398
Driving Excel from Access	402
Driving Microsoft Project from Access	405
Driving Outlook from Access	407
Driving Access from Another Application with Automation	418
Summary	422

14 Programming for Power with ActiveX Controls 423

Understanding the ActiveX Common Controls	424
Using the TabStrip Control	425
Using an ImageList Control with the TabStrip Control	425
Programming the Standard Access Tab Versus the ActiveX TabStrip Control	426
Taking a Closer Look at the ImageList Control	428
Adding Images During Design Time	428
Adding Images to the ImageList Control at Runtime	431
Emulating the Windows Explorer with the ListView Control	434
Looking at the Different Views of the ListView Control	434
Seeing the Major Groupings of the ListView Control Properties	435
Setting Up a ListView Control Manually	436
Creating and Filling a ListView Control Using VBA	437
Displaying a Task's Progress with the ProgressBar Control	439
Displaying the Access Progress Bar with SysCmd()	439
Using the ActiveX ProgressBar Control	441
Sizing Text Boxes at Runtime with the Slider Control	443
Telling It Like It Is with the Rich Textbox Control	445
Properties of the Rich Textbox Control	446
Code Behind the Microsoft Rich Textbox Control	448
Creating Status Bars for Individual Forms with the StatusBar Control	450
Properties of the StatusBar Panels Collection	451
Setting Status Bar Properties at Runtime	452

Docking Toolbars on Forms Using the	
ToolBar Control	453
Viewing Data File Manager Style with the TreeView Control	456
Summary	460
15 Extending the Power of Access with API Calls 461	
Understanding Dynamic Link Libraries	463
Examining the Syntax for API Calls	463
Finding API Declarations	466
Viewing the Possible API Calls	466
Using the API Viewer to Locate Calls	467
Finding API Calls in the Win32api.txt File	468
Considering Some Issues When Using API Calls	470
Creating Your Own API Declarations from Scratch	470
Converting 16-Bit to 32-Bit API Calls	471
Looking at Some Examples of API Calls	471
Finding an Executable Application Associated with a File	472
Connecting and Disconnecting Network Drives from	
Within Access	476
Displaying the Current User and Computer Name	482
Displaying Pertinent Folders from Within Your Application	483
Using the Open File Dialog API Call	486
Summary	490
16 Extending Your VBA Library Power with Class Modules and Collections 491	
Setting Up a Bookmark Tracker	492
Feature Set of the Bookmark Tracker	492
Basic Objects of the Bookmark Tracker	493
Let's Have a Little Class...Modules, That Is	494
Managing Multiple Instances of the Same Form	506
Looking at the Feature Set	506
Looking at the Forms Used to Open Copies of the	
Same Form	506
Examining the Code for Managing Multiple Copies	507
Summary	510
17 Creating Your Own Wizards and Add-ins 511	
Understanding Access Wizards, Builders, and Add-Ins	512
Looking at the Wizards and Add-In Registry Entries	513
Creating Your Own Add-Ins	515
Working with the Bookmark Tracker Wizard	515
Installing Add-Ins in Access	518
Programming the Bookmark Tracking Wizard	522
Finishing with the Wizard	532

Using Access Code Libraries	537
Looking at the Pros and Cons of Code Libraries	538
Considering Where to Put the Library Database	538
Setting a Reference to a Library	539
Viewing Library Routines in the Object Browser.....	540
Looking at Some Library Coding Issues.....	541
Summary	542
18 Manipulating the Registry with VBA 543	
Looking at the Windows Registry's History.....	544
Using the Windows Registry in Your Applications	545
Parts Making Up the Registry	546
Tools Used For Working with the Registry.....	548
Using VBA's Registry Commands	550
Performing Tasks with Registry API Calls.....	555
Looking at the Sample Application.....	557
Working with the Actual Code	558
Summary	572
19 Using Access with the Internet 573	
What's New in Access 2002 for the Internet?	574
Using the Access Hyperlink Features.....	574
Working with Unbound Hyperlink Controls.....	576
Maintaining a Hyperlink Base for a Database	578
Looking at the Hyperlink Data Type.....	579
Using the IsHyperlink Property to Add Hyperlinks to Your Interface	581
Programmatically Using Hyperlinks with the Follow, FollowHyperlink, and HyperlinkPart Methods	582
Working with Hyperlink Options	586
Easily Importing and Exporting Access Objects to HTML Documents	587
Exporting to HTML.....	587
Importing and Linking to HTML Files.....	590
Publishing to Other Web File Formats	592
Summary	594
PART IV Managing Databases	
20 Securing Your Application 597	
Understanding the Purpose of Securing Applications	598
Protecting Sensitive Data: The Client's Perspective	598
Protecting Code: The Developer's Perspective	599
Understanding Access Security	599
Share-Level Security: The Database Password.....	600
User-Level Security: The Real Security System of Access	601

Looking at the Security User Interface.....	608
Working with PIDs, SIDs, WIDs, and Passwords.....	608
Creating a New User	610
Creating a New Group.....	611
Removing Users and Groups.....	612
Adding a User to a Group	612
Adding a Password to a User Account	613
Removing a Password from a User Account.....	613
Setting Permissions on Objects	614
Securing Modules in the VBE.....	615
Setting Database Permissions	616
Changing the Owner of an Object.....	617
Encrypting a Database.....	619
Creating a Workgroup Information File	619
Manually Securing a Database	620
Making Life Easier with Access Security Tools.....	625
Using the Security Wizard.....	625
Printing Users and Groups from Access	625
Reading the Security White Papers	626
Using Other Security Resources	626
Avoiding Common Pitfalls Found in Access Security	626
Planning Security.....	626
Creating Objects with Default Accounts.....	626
Securing Linked Tables in a Multiuser Environment	627
Running with Owner's Permissions	627
Using Security in a Replication Environment.....	629
Distributing Secured Applications with the Microsoft Office Developer.....	629
Distributing Secured Applications Through an .mde File.....	630
Managing Security Through Code	630
Programming Security with DAO	630
Creating a New User Through Code	632
Deleting a User Through Code	633
Setting the Database Password Through Code	634
Creating a Group Through Code.....	635
Deleting a Group Through Code.....	636
Adding a User to a Group Through Code	637
Removing a User from a Group Through Code	638
Changing the Owner of an Object Through Code	639
Setting Permissions for an Object Through Code.....	640
Checking Permissions Through Code	641
Determining Who You're Logged On As Through Code	643
Denying Users the Ability to Create Databases	643
Denying the Creation of Table and Query Objects	644

Compacting, Encrypting, or Decrypting a Database	
Through Code	645
Disabling the Bypass Key Through Code	646
Using the Secured Sample Database: Chap20s.mdb	647
Summary	648
21 Handling Multiuser Situations 651	
Understanding Multiuser Terminology	652
Understanding Multiuser Handling in Access	653
Default Record Locking	653
Default Open Mode: Shared Versus Exclusive	654
Number of Update Retries	656
ODBC Refresh Interval	656
Refresh Interval	656
Update Retry Interval	656
One or Two Database Containers: Knowing Where to	
Put the Pieces	657
Knowing What Should Go Where: An Overview	657
Splitting Databases	659
Looking at the Built-In Locking Modes	661
Understanding Row-Level Versus Page-Level Locking	661
Using the Built-In Locking Modes	662
Using Locking Modes in VBA	665
Using Alternative Locking Schemes	665
Working in VBA with Unbound Forms	666
Creating the Routines for Handling Unbound Forms	667
Using the Sample Form, Step by Step	668
Using Support Routines	670
Coding for Multiuser Error Handling	685
Summary	689
22 Welcome to the World of Database Replication 691	
Understanding Database Replication Concepts	692
The Replication Design Goal	693
Some Typical Replication Applications	694
Working with Jet Replication Tools	694
The Briefcase	695
Access Menus	696
Replication Manager	698
Jet and Replication Objects Programming	699
Converting Databases to Replicas	699
Synchronizing Replicas	704
Understanding the Design Master and Replicas	706
Recovering the Design Master	707
Replication Visibilities	707

Replication System Columns, Tables, and Other Mysteries	708
Using Replica Sets.....	710
Understanding Replica Set Topologies.....	710
Singly Connected List	710
Star and Hub Topologies	711
Automating Star and Hub Synchronization	711
Distributing Replicable Applications	713
Using Replicable and Non-Replicable Objects	714
Creating Partial Replicas	715
Replicating Back-End and Front-End Applications	719
Handling Replication Conflicts	719
Using the Conflict Viewer	720
Using an Alternative Conflict-Resolution Algorithm	723
Understanding Various Replication Conflicts	726
Understanding Replication Synchronizers	728
Synchronization Phases	733
Direct and Indirect Synchronizations	734
Scheduled and On-Demand Synchronizations	734
Synchronizing Replicas over the Internet	734
Handling Counter Fields	735
Using Read-Only Attributes with Replication	736
Performing Replication Identification Fixup	737
Using the Last Synchronization Partner	737
Using the Compact Utility with Replicated Databases.....	738
Deciding Whether to Back Up Your Replicas	738
Upgrading Replica Sets to Access 200x.....	739
Securing Replicated Applications.....	739
Using MDE Files with Replicated Databases	739
Creating Successful Replication Applications	740
Summary	740

23 Moving Workgroup Applications to Client/Server 741

Understanding Client/Server.....	742
Working with Open Database Connectivity	743
Reasons to Use Access for Client/Server	744
Factoring for Client/Server Migration	745
Amount of Data	745
Use and Purpose of Database	747
Database Design	747
Concurrent Use and Number of Users	747
Backup and Recovery	747
Security	748
Data Sharing Among Applications	748
Network Traffic	748

Record Aggregation	749
Bet Your Career on Choosing the Right System	749
Planning for Client/Server	750
Field and Table Names	751
Reserved Words	751
Case Sensitivity	752
Query Processing on the Server	752
Knowing What to Watch for in Application Development.....	753
Limiting Your Data	753
Using Combo Boxes	754
Using Access-Specific and User-Defined Functions.....	755
Creating Heterogeneous and Cross-Database Joins	755
Dealing with OLE Objects	755
Using Local Tables for Static Information	756
Converting Existing Applications	756
Starting with a Well-Designed Database.....	756
Using Timestamp Fields	757
Cleaning Up Queries	757
Reworking Forms	758
Developing Advanced Applications	760
Working with Current Access Security	763
Upsizing Access Databases	764
Using the Upsizing Wizard	768
Distributing a Client/Server Solution	772
Programmatically Setting Up an ODBC Data Source	772
Re-creating a SQL Database with Server Scripts	775
Loading Existing Data into SQL Server	776
Keeping Certain Issues in Mind with Access and SQL Server	776
Summary	777

24 Developing SQL Server Projects Using ADPs 779

Understanding Project File Architecture	780
Understanding OLE DB	780
Linking to Data	781
Data Links and Access Projects	782
Data Links and VBA Code	783
The Microsoft Data Engine.....	786
Objects on a SQL Server.....	787
Working with Projects	789
Creating a New Project	789
Project Properties.....	792
Securing a Project	794

Building a Client/Server Application	795
Working with Tables	796
Naming Conventions for Objects	799
Server Data Types	800
Using Constraints	803
Using Triggers	810
Optimizing Data Access	813
Working with Views	815
Working with Stored Procedures	818
Creating Stored Procedures	819
Comparing Stored Procedure and Access Syntax	822
Summary	824

Part V Adding Finishing Touches

25 Startup Checking System Routines Using ADO 829

Performing Startup System Checks	830
Setting and Retrieving System Settings	835
Notifying and Logging Users Out of an Application	836
Keeping Users Out at Startup Time	836
Logging Users Out in the Middle of the Application	837
Setting the Flag File to Log Users Out of the Back End	841
Testing Table Links at Startup	843
Linking and Unlinking Tables in a Jet Back End in the Application's Folder	845
Finding the Jet Back End with the OpenFile API Call	849
Testing and Repairing Corrupted Jet Back-End Databases	852
Checking and Notifying Users of a New Version	857
Summary	859

26 Creating Maintenance Routines 861

Creating an Export Dialog to Export Tables	862
Examining the Export Utility's Features	863
Examining the Code Behind the Export Utility	865
Compacting and Repairing the Back End on Demand	868
Creating a Generic Code Table Editor	871
Replicating Tables from Back End to Front End for Better Performance	874
Creating a Replicated Table Editor	875
Looking at Startup Routines for Replicating Semi-Static Data	883
Summary	887

Index 889

About the Author

F. Scott Barker holds a B.S. in computer science and has worked as a database developer for more than 15 years, first with Clipper and then for the last eight years with Microsoft Access, Visual Basic, and SQL Server.

While working at Microsoft for two years, Scott was on the Microsoft Access and FoxPro teams. He now contracts with Microsoft and the Access team by developing in-house tools used throughout Microsoft. With his company, Applications Plus, Scott also does contract development for companies in the banking, medical, and insurance industries.

Scott has trained for Application Developers Training Company and others all around the United States. He is a frequent speaker at Access conferences throughout the U.S., Canada, and Europe. Through his classes and conferences, Scott has trained thousands of developers.

Scott has written articles for *Smart Access* (Pinnacle), *Data Based Advisor Magazine* (Advisor), *Access*, *VB*, *SQL Advisor* (Advisor), *Microsoft Office & VBA Developer* (Informant Communications), and the German *VBA Magazin*.

Scott is the author of *Using Access 97*, published by Que Corporation, and of Sams/Que's *Access Power Programming* books for Access versions 95, 97, and 2000. He has also tech edited/reviewed a number of books.

Dedication

To my mom, Nona Barker. I love you.

Acknowledgments

Again, first off I want to thank God for continuing to give me ideas and making all the development and writing fresh and exciting to me.

I want to thank Pai Chung and Pete Klein for tech editing this version, and putting up with me getting sample code to them late.

I want to thank all my friends on the Microsoft Access and Office teams for putting up with my questions all the time: David Gainer and Clint Covington for his help on XML and Data Access Pages. It's neat to work with someone so excited about a technology.

I want to thank all my Access friends: Paul Litwin, Ken Getz, Steve Forte, Roger Jennings, John Viescus, Mike Groh, Mike Hernandez, Mary Chipman, and Andy Baron. It seems as though we see each other only at conferences sometimes.

I want to thank Alison Balter, who besides being a kick to work with and very sharp in the ways of Access and SQL Server, is a great person all around. Alison, always keep that enthusiasm! Also, I want to thank her husband, Dan, who puts up with late-night calls when we have to discuss projects.

Finally, I have to thank my family for again putting up with me working many, many nights. Diana, Chris, Kari Anne, Nichole, and David, I love you.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an executive editor for Sams, I welcome your comments. You can e-mail or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail: feedback@sampublishing.com

Mail:

Sams Publishing
800 East 96th Street
Indianapolis, IN 46240

A Letter from the Author

Dear Reader,

Here we are again. The Microsoft Access team has come out with another winner! With this version of Access, we actually see some enhancements to features that haven't really been updated in three versions.

In addition to giving us an even closer match between Access and SQL Server, richer features in Data Access Pages, and bringing Access up to speed with XML, Microsoft added features to the beloved report writer. Included with report enhancements is the capability to create Web reports using the Access Report Designer. But I'll save that for the report chapter.

One goal behind this edition was to take the code that I originally did in DAO (Data Access Objects) and convert it to the latest standard in data access, ActiveX Data Objects (ADO). Don't worry, though—some chapters that were heavy in DAO, such as “Startup System Routines Using DAO,” are still available on the Web, as are the appendixes that used to be found at the end of this book. You can find these elements at www.sampublishing.com (type this book's ISBN in the Search box).

When I buy a book, I first look at its table of contents. If I find at least five things that I can use from the book, I know I'll get my money's worth. With this book, I hope you can find at least 50 things that you can use. You'll find some techniques that expand your development skill set, as well as items you can plug right into your code without caring about how they work (although if you're like me, you will care).

As a result, I hope you enjoy this book and use it, to paraphrase one reader, “'til the cover comes off.” If you see me at a conference or training, please stop me, say hi, and tell me what you think of the book.

Sincerely,

F. Scott Barker

Author, *Microsoft Access 2002 Power Programming*

Who Is This Book For?

Again, this book is for corporate developers and independent consultants because both have the same goal of cranking out robust systems in as little time as possible. For example, routines for startup system checking allow you to control when users can get onto the system. These routines not only check for a good back end (and locate it if not found), but also repair the back-end database, if necessary (of course, after logging out all other users). Other situations are covered as well.

If you are reading this after using Access macros all your life, don't be put off by the first chapter, "Macros Are for Weenies; Code Is Cool!" You've just entered a new dimension in your development career and, just as college seniors harass underclassmen, VBA developers must give those who exclusively use macros a hard time. Seriously—you will find in using VBA an environment where you more or less have total control.

If you're switching from another language, the first few chapters are what you have to learn to get a handle on VBA. If you've been developing in Visual Basic, Chapter 2, "Coding in Access 2002 Using VBA," will give you an idea of the language differences.

If you're an advanced developer, you'll appreciate the chapters on the changes to the development environment such as when to use ADO versus DAO, and ADP versus MDB. In addition to new material on using XML from Access, you can see example code that uses ADO for all your common tasks.

What's Covered in This Book?

F. Scott Barker's Microsoft Access 2002 Power Programming is broken into five parts, which follow a logical progression—from power programming basics to routines you can use to add robustness to an application.

Part I: The Root of Power Programming

Part I begins with a quick chapter showing you how to switch from macros to Visual Basic for Applications (VBA). Then we get right into the meat for those of you getting into Access from other languages, with a quick study on VBA so that you can understand what's going on in the rest of the book. Part I also examines the two virtual environments Access 2002 provides to develop solutions, as well as the two data-access methods that you can use to work with data from code. Last, various error-handling techniques are presented.

- Chapter 1, "Macros Are for Weenies; Code Is Cool!" gives power users information on what's needed to switch from using macros to coding in VBA.
- Chapter 2, "Coding in Access 2002 with VBA," gives a complete overview of VBA's commands and features.
- Chapter 3, "Making Access Project and Data Technology Choices," introduces the latest Access environment choices. You see the paths you can go down in developing your application and the differences between them.
- Chapter 4, "Working with Access Collections and Objects," shows the various collections and objects that make up the Access object model, including ADP and MDB.

- Chapter 5, “Introducing ActiveX Data Objects,” explains ADO, Microsoft’s preferred method for working with data, so you can get going with the latest technology used throughout the Microsoft products.
- Chapter 6, “Using XML with Access 2002,” explains what XML is, how it’s used, and what you can do with this latest HTML language with Access.
- Chapter 7, “Handling Your Errors in Access with VBA,” explains how to take full advantage of error trapping, as well as use a centralized error routine for logging errors to disk.

Part II: Manipulating and Presenting Data

Although the term *power programming* sounds as though it could relate only to cranking out code, Access has powerful tools in their own right that you access through the user interface. That’s what the chapters in Part II are all about.

Underlying every great form or task is a query, and making sure that your queries are optimized as much as possible can make or break an application. Next, creating reusable forms is discussed, as are new form properties and methods. You’ll see some powerful ways to use combo and list boxes, as well as how to manipulate the controls in code. Various techniques also are shown for creating versatile reports in VBA. Last, you look at Data Access Pages, which give you the best of both worlds in forms and reports.

- Chapter 8, “Using Queries to Get the Most Out of Your Data,” shows how to create powerful queries for tasks, forms, and reports, as well as what optimization techniques can be used and when. Examples are given for using advanced query by form and for creating totals for crosstab queries.
- Chapter 9, “Creating Powerful Forms,” explains how to create reusable forms, use paging techniques, and use some of the new developer features found in Access 2002.
- Chapter 10, “Expanding the Power of Your Forms with Controls,” provides useful examples on creatively using combo and list boxes and creating forms with spreadsheet-type cursor movement. Included are techniques for creating forms and controls on-the-fly to build data-driven applications through VBA.
- Chapter 11, “Creating Powerful Reports,” shows how to create dynamic reports based on user requests—including one form that does the job of three—and how to create a report that reflects changing months even while using a crosstab query for the record source. Last, new report properties are discussed.
- Chapter 12, “Working with Data Access Pages,” shows how to take advantage of this interactive report feature. This chapter not only shows how to get going with DAPs, but also how to enhance them with ActiveX controls and extend them with code.

Part III: Extending Access with Interoperability

Easily my favorite part of the book (and Access in general), Part III shows how to automate the various Microsoft Office applications from each other through VBA. Some ActiveX controls that you can use with Access are explored with example code, including the Windows Common controls, which are included in the Microsoft Office Developer (MOD).

Part III includes discussions on creating class modules, collections, and wizards to increase your programming productivity. Access's Internet features also are looked at, with plenty of examples.

- Chapter 13, "Driving Office Applications with Automation," includes abundant sample code that shows how to drive various Office applications from Access, including Word, Excel, Project, Outlook, and Graph. Working the other way, other examples show how to drive Access from Excel and Project.
- Chapter 14, "Programming for Power with ActiveX Controls," discusses the Windows Common Controls that come in the Microsoft Office Developer. Sample code shows how to use the properties and methods for each control.
- Chapter 15, "Extending the Power of Access with API Calls," shows how to use Windows API calls to add features to Access that aren't in the base product.
- Chapter 16, "Extending Your VBA Library Power with Class Modules and Collections," shows you how to really take advantage of some of the more advanced features of VBA, and bring them to a level where they can greatly increase development productivity. This chapter includes a cool example of tracking multiple bookmarks for a user's form by using class modules and collections.
- Chapter 17, "Creating Your Own Wizards and Add-Ins," continues where Chapter 16 stops, discussing how to create real-world wizards to use during development. A couple of builders are included to help you add the new features discussed in Chapter 16 to any of your forms in only a couple of keystrokes.
- Chapter 18, "Manipulating the Registry with VBA," explains how to take advantage of using the Windows Registry to store values that you can use to set up your application's environment.
- Chapter 19, "Using Access with the Internet," discusses the Internet features built into Access. Included are the HyperLink datatype, importing from and exporting to HTML tables, publishing forms and reports on the Internet from Access, and more.

Part IV: Managing Your Databases

No matter how advanced the software is, you must always take extra steps in programming when using your application on a network. Access is no different; although you can get by

with simply throwing an application on the server, you still need to know how to optimize it for the network environment and to handle different situations that can occur.

Replication is fully explained, showing not only how to use it and to optimize for it, but the theories behind it.

SQL Server is so popular nowadays—it's even included in the Access box!—that Access developers have to know something about it. One chapter in Part IV helps you think ahead for what's in store when developing an application that works with SQL Server. Another chapter discusses using Access right from the workstation where you do your development.

- Chapter 20, “Securing Your Application,” thoroughly introduces the Access Security Model through the user interface and code. This chapter also presents tips, tricks, and VBA code examples for handling some of the most asked-about situations in Access security.
- Chapter 21, “Handling Multiuser Situations,” gives an overview on the multiuser environment within Access, including locking methods and error handling. You'll also see sample code for using unbound forms, a popular method for increasing performance in certain situations. Options for row-level locking are also discussed.
- Chapter 22, “Welcome to the World of Database Replication,” discusses how to use database replication to its full advantage. Topics range from distributing replicated databases, understanding how conflicts are handled, and troubleshooting replicated databases.
- Chapter 23, “Moving Workgroup Applications to Client/Server,” gives an overview of what it takes to use Access with SQL Server. This chapter will help prepare you for making sure that your applications don't have to be gutted when the boss says, “Let's use SQL Server.” Also covered is how to convert your legacy Access applications to SQL Server, as well as steps for using the Upsizing Wizard.
- Chapter 24, “Developing SQL Server Projects Using ADPs,” shows how to use Access Data Projects for developing SQL Server projects.

Part V: Adding Finishing Touches

Think of the chapters in Part V as being “Scott's idea of what all applications should have in them.” Have fun!

- Chapter 25, “Startup Checking System Routines Using ADO,” covers not only checking the back end to make sure it's still there (for Jet and SQL Server), but also checking for valid versions of the application, even if the front end should be running at this time. Actual code for logging users out of the back end is given.

- Chapter 26, “Creating Maintenance Routines,” continues with what I deem as useful routines that allow you to generically export tables to any supported format, compact and repair the back end from the front end (again, logging users out first), replicate semi-static tables for performance enhancements, and more.

Using This Book's Web Site

You can find the examples of this book on this book's Web page at www.sampublishing.com (type this book's ISBN in the Search field). To use the examples from this book, download them from the Web site onto your hard drive.

You also can find this book's appendixes on this Web site. (As with most development books, I'm limited as to how much information I can fit in the actual book.) You will find appendixes for debugging Access 2002 code (Appendix A), working with ActiveX controls and Data Access Objects (Appendixes B and C), and programming office components such as command bars and the Office Assistant (Appendix D). Also, Appendix E, “Access 2002 and Jet 4 Errors,” lists all Access and Jet 4.0 errors.

Conventions Used in This Book

As with other routines created for system or generic purposes, many routines in this book are prefixed with `ap_`, the initials for Applications Plus, my company. This isn't for copyright purposes but to ensure that they don't conflict with other routines you might have with the same name.

Names of all dialogs and dialog options use initial capital letters. New terms are introduced in *italic* type.

Messages that appear onscreen, as well as all program code and Access commands, appear in a special monospace font, as in the following example: `Variable undefined`. Text that you are to type appears in **monospace boldface**; syntax variables that you need to replace with an appropriate value appear in *monospace italic*. When a choice is given for parameters, a pipe symbol (|) is used. Finally, the Leszynski Naming Conventions are used for all code examples.

Although you can use an underscore character (`_`) in VBA code to indicate the continuation of a line, you might occasionally see a ➤ character in code. This character also indicates that a code line (usually a query or SQL statement) is continued from the line above it.

Occasionally you will encounter tips, notes, and cautions. Tips suggest easier or alternative methods of executing a procedure. Notes indicate additional information that may help you avoid problems or that you should consider when using the described features. Cautions warn you of hazardous procedures (for example, activities that delete files).

The Root of Power Programming

PART

I

IN THIS PART

- 1 Macros Are for Weenies; Code Is Cool! 9
- 2 Coding in Access 2002 with VBA 21
- 3 Making Access Project and Data Technology Choices 71
- 4 Working with Access Collections and Objects 79
- 5 Introducing ActiveX Data Objects 109
- 6 Using XML with Access 2002 133
- 7 Handling Your Errors in Access with VBA 147

Macros Are for Weenies; Code Is Cool!

CHAPTER

1

IN THIS CHAPTER

- Understanding Where Macros End and Code Begins 10
- Looking at Macro-to-Code Changes 12
- Converting Existing Macros to VBA Code 15

Many of you reading this book have been using Access for quite a while and automating it quite nicely with macros. Macros work quite well for applications used by one or maybe a few people in a controlled environment. But when it comes to creating powerful, robust applications, VBA is the only way.

Microsoft Access is unique in that the macro language it uses isn't the "recording" of commands, as in Excel or the Windows Recorder, but instead is made of the commands created from a set list of actions, each with its own set of arguments. Figure 1.1 shows an example of several macros that perform several actions, including `OpenReport` and `OpenDataAccessPage`.

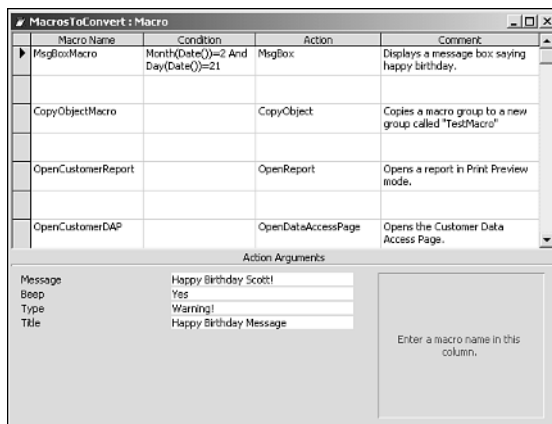


FIGURE 1.1

Macros give users a jump-start in automating their applications and perform commands, such as `OpenReport`, `OpenDataAccessPage`, and other actions shown here.

Understanding Where Macros End and Code Begins

Although macros are very versatile and can take care of quite a few tasks that need to be performed, they do have limitations. For example, you can't perform Automation with macros, loop through recordsets, or set up error handling. These actions all require the use of VBA.

When Are Macros Necessary?

The Access team at Microsoft has whittled down the need for macros. In Access 2002, there is only one reason to use macros: to create key combinations. A common example is using `^P` to print (where `^` is used for the `Ctrl` key). Third-party vendors and developers who create "Office-compatible" applications use Autokeys to include necessary key combinations.

TIP

An AutoExec macro used to handle whatever actions you wanted to perform when the database was opened. A Startup Form property is available from the Tools menu's Startup command. On the form you specify, program an initialization routine for one of the form's events, such as the unload event, in the form specified in Startup dialog's Display Form property. Chapter 26, "Startup Checking System Routines Using ADO," shows an initialization routine in action.

NOTE

In versions before Access 97, menus were created by using a combination of the AddMenu macro action and the actions to perform for the menu choices. Macro menu actions still exist in Access 2002 for backward compatibility. However, you can now use command bars set to the type Menu.

When Is Code Necessary?

A single book really can't list the total number of tasks that require code, because you're limited to your imagination in creating tasks using VBA. These are just a few of the tasks requiring the use of VBA:

- **Error handling.** When you use macros and an error occurs in the Access retail product, because you have no control whatsoever, you get a couple of gray error boxes and are sent back to where you started that command. With the runtime application that comes with the Package and Deployment wizard, you get dumped out to Windows when errors occur.

In VBA, on the other hand, you can use error-handling commands to trap errors and handle them gracefully. (For more information on error handling, see Chapter 7, "Handling Your Errors in Access with VBA.")
- **Automation,** used for communicating with other applications. For more information on the power of Automation, see Chapter 13, "Driving Office Applications with Automation."
- **Transaction processing.** This gives developers control over completing a task or rolling it back.

- **Declaring and calling Windows API routines.** For more information on Windows API routines, see Chapter 15, “Extending the Power of Access with API Calls.”
- **Performing replication tasks.** For more information on replication, see Chapter 22, “Welcome to the World of Database Replication.”

Looking at Macro-to-Code Changes

Power users looking to become developers sometimes feel nervous about switching to code because they fear getting themselves into a complex world without all the commands they have available with macros. This is far from true. Although getting into code can be confusing at first, switching to VBA gives you not only more flexibility, but also more control. Most macro commands are covered with the DoCmd object (described in the following section); others either have equivalent commands or aren’t needed in code.

Using the DoCmd Object

In code, most macro actions can be performed with the DoCmd object. The actions are used as methods that can be called with arguments. Most arguments have intrinsic constants, which can be used in place of numbers. An example of this is using the acCmdSaveRecord constant with the RunCommand method to save the current record in a form:

```
DoCmd.RunCommand acCmdSaveRecord
```

NOTE

In versions of Access before Office 97, the DoMenuItem command performs the same commands as RunCommand. Access automatically changes DoMenuItem commands to RunCommand when converting applications.

acCmdSaveRecord is an *intrinsic constant* and represents a value needed for a parameter. Access uses intrinsic constants often, thus saving you development time; you don’t have to remember numbers because you have the intrinsic constants listed.

Another is the OpenForm method:

```
DoCmd.OpenForm "SplashScreen", acNormal, , , acFormEdit, acWindowNormal
```

Table 1.1 lists the macro actions that are used as methods of the DoCmd object.

TABLE 1.1 DoCmd Object Method Alternatives to Macro Actions

<i>Method</i>	<i>Visual Basic Alternative</i>
AddMenu	
ApplyFilter	
Beep	Beep statement
CancelEvent	Use the Cancel argument in event procedures
Close	
CopyObject	
DeleteObject	Use the Delete method of the object collection
Echo	Application.Echo
FindNext	
FindRecord	
GoToControl	<i>ControlName</i> .SetFocus
GoToPage	<i>Form</i> .GoToPage
GoToRecord	
Hourglass	
Maximize	
Minimize	
MoveSize	
OpenDataAccessPage	
OpenDiagram	
OpenForm	
OpenFunction	
OpenModule	
OpenQuery	<i>DatabaseVariable</i> .Execute for action queries
OpenReport	
OpenStoredProcedure	
OpenTable	
OpenView	
OutputTo	
PrintOut	
Quit	Application.Quit
Rename	Rename the object by using the DAO <i>object</i> .Name property
RepaintObject	
Requery	<i>ObjectName</i> .Requery

TABLE 1.1 Continued

<i>Method</i>	<i>Visual Basic Alternative</i>
Restore	
RunCommand	
RunMacro	
RunSQL	<code>DatabaseVariable.Execute "SqlString"</code>
Save	
SelectObject	
SendObject	MAPI commands (SendObject is more efficient)
SetMenuItem	
SetWarnings	
ShowAllRecords	
ShowToolbar	
TransferDatabase	
TransfersSpreadsheet	
TransferSQLDatabase	
TransferText	

In addition to the DoCmd object, many macro actions have alternative methods and statements available. In most cases, these commands are more efficient to use than the DoCmd method. This generally isn't the case when DAO or ADO alternatives are used.

Code Equivalents of Macro Commands

Rather than being simply a straight port over from macro action to VBA command, some commands are much more flexible in the action they perform. It's generally better to use a code command over a macro whenever possible. The macro actions with code equivalents are as follows:

- `MsgBox`. In place of this macro action, use the `MsgBox()` function either to retrieve an answer from the user or simply to display a message box.

TIP

One feature of the `MsgBox()` function is that it can include other buttons in the message box, such as Yes, No, and Cancel. Another feature is that it returns what the user pressed or clicked. For information about `MsgBox()`, look up *MsgBox Function* in the Answer Wizard.

- **RunApp.** Use the `Shell()` function to run another application.
- **RunCode.** This action allows you to use the string name of a function or subprocedure to run it. With code, simply type the function or subroutine name, with any options necessary.
- **SendKeys.** You can use a `SendKeys` statement in code.
- **SetValue.** You can set the value directly in Visual Basic.
- **StopAllMacros and StopMacro.** These statements aren't necessary because macros aren't used. When a macro is converted to VBA, the `End` statement replaces `StopAllMacros`, and the `Exit Sub|Function` statement replaces `StopMacro`.

Converting Existing Macros to VBA Code

One feature as of Access 97 is the capability to convert your current macros into VBA code. This is great for two reasons:

- You can learn how to code VBA.
- You can use error handling around the commands.

TIP

Before converting your macros, fill in the `Description` property for as many macros as possible. Access will carry these descriptions over into comments. Also, when naming your macro, if you use all 64 allotted characters, the conversion will fail because it wants to add converted macro - in front of the macro name.

To show you how to convert macros, I'll use the `MacrosToConvert` macro group. You can find this group (see Figure 1.2) on this book's Web page at www.sampublishing.com, in `\Examples\Chap01\Chap01.mdb`.

Either from the open macro group or from the highlighted macro group in the database container, you can convert a macro group to a code module by following these steps:

1. From the **File** menu, choose **Save As**. The **Save As** dialog appears (see Figure 1.3). In addition to the choice of names for the new object, you also are offered a drop-down list with two choices: **Macro** or **Module**.
2. Select **Module** and click **OK**. The dialog in Figure 1.4 appears.

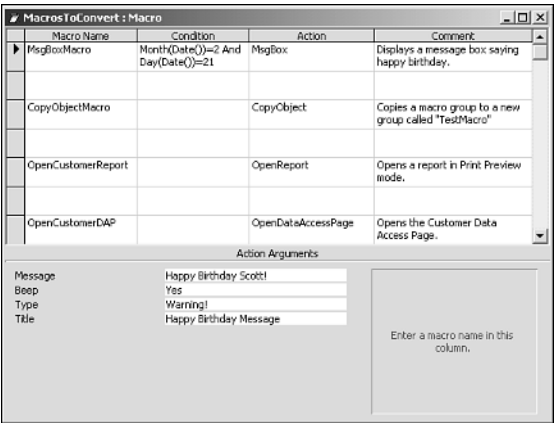


FIGURE 1.2
The MacrosToConvert macro group will be converted into VBA.

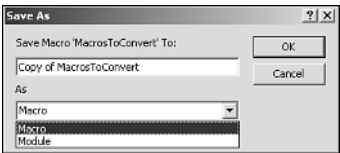


FIGURE 1.3
The Save As dialog selected from within the macro design converts macros to VBA.

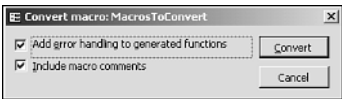


FIGURE 1.4
Error handling is offered when converting macros to VBA.

TIP

Leave the Add Error Handling to Generated Functions and Include Macro Comments options checked so that Access will place error-handling code in the created routines and comments above the converted macros.

3. Click Convert. The macros are converted and a new module is created.
4. A message box appears when the conversion process is completed; click OK.

The module will be named *Converted Macro-NameofMacroGroup*. In this case, the new module was named *Converted Macro-MacrosToConvert*. Listing 1.1 shows the new code.

LISTING 1.1 Chap01.mdb: Code Listings for the Converted MacrosToConvert Macros

```
Option Compare Database
' .....
' MacrosToConvert_MsgBoxMacro
' .....
Function MacrosToConvert_MsgBoxMacro()
On Error GoTo MacrosToConvert_MsgBoxMacro_Err
If (Month(Date) = 2 And Day(Date) = 21) Then
    ' Displays a message box saying happy birthday.
    Beep
    MsgBox "Happy Birthday Scott!", vbExclamation, "Happy Birthday Message"
End If

MacrosToConvert_MsgBoxMacro_Exit:
Exit Function

MacrosToConvert_MsgBoxMacro_Err:
MsgBox Error$
Resume MacrosToConvert_MsgBoxMacro_Exit

End Function

' .....
' MacrosToConvert_CopyObjectMacro
' .....
Function MacrosToConvert_CopyObjectMacro()
On Error GoTo MacrosToConvert_CopyObjectMacro_Err
' Copies a macro group to a new group called "TestMacro"
DoCmd.CopyObject "", "TestMacro", acMacro, "MacrosToConvert"

MacrosToConvert_CopyObjectMacro_Exit:
Exit Function

MacrosToConvert_CopyObjectMacro_Err:
MsgBox Error$
Resume MacrosToConvert_CopyObjectMacro_Exit

End Function
```

LISTING 1.1 Continued

```

' .....
' MacroToConvert_OpenCustomerReport
' .....

Function MacroToConvert_OpenCustomerReport()
    On Error GoTo MacroToConvert_OpenCustomerReport_Err
    ' Opens a report in Print Preview mode.
    DoCmd.OpenReport "SampleReport", acViewPreview, "", ""

MacroToConvert_OpenCustomerReport_Exit:
    Exit Function

MacroToConvert_OpenCustomerReport_Err:
    MsgBox Error$
    Resume MacroToConvert_OpenCustomerReport_Exit

End Function

' .....
' MacroToConvert_OpenCustomerDAP
' .....

Function MacroToConvert_OpenCustomerDAP()
    On Error GoTo MacroToConvert_OpenCustomerDAP_Err
    ' Opens the Customer Data Access Page.
    DoCmd.OpenDataAccessPage "dapCustomer", acDataAccessPageBrowse

MacroToConvert_OpenCustomerDAP_Exit:
    Exit Function

MacroToConvert_OpenCustomerDAP_Err:
    MsgBox Error$
    Resume MacroToConvert_OpenCustomerDAP_Exit

End Function

```

Now that the macros are converted to Visual Basic code, you need to change the macro calls into function calls as well. For example, when you change a macro call to a function call on the Click event of a command button, the command goes from being

MacroGroup.MacroName

to

=FunctionName()

TIP

One of the best ways to learn the most common commands used in VBA is to examine the functions created by the macro conversion process. Better yet, convert the macros yourself, by going through them line by line and creating the VBA code by hand. That's how I got a great handle on them—by converting nine pages of one macro group to code.

NOTE

In Access 2002, you can also convert macros by using the Tools menu's Macros command. Besides being able to convert regular macros from this menu, you can choose to convert menus, toolbars, and shortcut menus.

Summary

Macros are great for jump-starting an application, but they take you only so far; you need to turn to code for a more robust, powerful application. You can convert most macro commands to a method of the DoCmd object. Access includes an easy way to convert your macros so that you keep all the work you've done and move your macros over to VBA.

For more information on VBA commands and error handling, see these chapters:

- Chapter 2, “Coding in Access 2002 with VBA,” gives an overview of using VBA in Access.
- Chapter 7, “Handling Your Errors in Access with VBA,” explains how to use error handling to make your applications more robust.

Coding in Access 2002 with VBA

CHAPTER

2

IN THIS CHAPTER

- Getting Started with Programming 22
- Getting Started with VBA 43
- Programming with Objects 47
- Using Properties and Methods 48
- Using Collections 54
- Customizing a Form 58
- Coding Class Modules 63

Since Office 97, Access has supported *Visual Basic for Applications (VBA)*. VBA is a common programming language component now shared among all of Microsoft Office, as well as Visual Basic 4 and later versions.

Because most application development involves writing code to process information, you naturally need an introduction to the vast number of features available in VBA. This information provides you with a solid foundation before you dig deeper into designing powerful applications with the help of this book.

NOTE

This chapter refers to the programming language features as *VBA features* rather than Access features, because the programming language within Access is a separate component. The concepts and features explained here don't just apply to Access, but also apply to all applications that support VBA. The *VBE (Visual Basic Editor)* is even in a separate window from Access; it's the same editor used in all Office products. VBA is also used in third-party applications such as AutoCAD.

VBA 6, first used in Access 2000, coincided with the feature set of Visual Basic 6. One goal of Microsoft was to keep the two products in sync. This changed with Access 2002 because it uses VBA 6.3, and the next version of Visual Basic (Visual Basic.NET) uses VBA 7.0.

Although VBA is a Visual Basic component, Visual Basic has many more components that make it the powerful development language that it is.

NOTE

As of this writing, VSA, the .NET version of VBA, is looming on the horizon. Don't worry, though—VSA is strictly for developing applications for the Internet, whereas VBA will still be used for LAN-based applications.

Getting Started with Programming

The following sections introduce you to programming in Access 2002. First, you're introduced to the concept of code modules, which is where you write your code. Here, you design and write a very simple form. Then you're introduced to declaring variables and subroutines, and how to control your program's flow. This information is the basis for programming in Access 2002. Along the way, you learn about different techniques for writing maintainable code.

Using Code Modules

You can write your code in three places in Access 2002: in class modules that can be one of three types, in code behind forms and reports, and in standalone class modules. The last module type is a self-contained standard code module. You should write all your code that processes a form or report in the form's or report's code module, and you should write any generic routines not specific to forms or reports in separate code modules. The independent class modules let you create custom objects that can include their own properties and methods.

By organizing common functions into separate code modules, you can more easily maintain your application. For example, you can write all your network routines in a Network Utilities code module, routines for manipulating strings in a Basic Utilities code module, and generic form code in a Form Utilities module.

Coding Behind Forms

To write any code behind forms in Access, you first must recognize the difference between Design, Form, and Datasheet view. When a form is in Design view, you can add and remove controls and write code. No code written behind the form is run.

NOTE

The steps for editing code behind forms also apply to editing code behind reports.

A form in Form or Datasheet view can have code running behind it. The primary differences between being in Form and Datasheet view is the way the form is displayed, and how much control you have over the data presented. Datasheet view displays a form on a grid and is limited in how you can customize it; Form view, on the other hand, lets you create complex forms and fully customize their appearance. Form view allows you to create forms that aren't limited to grids, and to display one record at a time or a continuous list.

NOTE

You can have different forms in your application in different modes at the same time. Some forms can be running while you're designing other forms. This is important to note, especially when debugging, because you might get an error if a running form tries to reference another form that's in Design mode.

You're now going to create a new form. This example introduces you to the code editor and to switching form views. Follow these steps:

1. From the database window, click the Forms tab and then the New button.
2. The New Form dialog appears. Select Design View and click OK.
3. You should see the form in Design view (see Figure 2.1). If you don't also see a toolbox or the property sheet, choose Toolbox and Properties from the View menu.

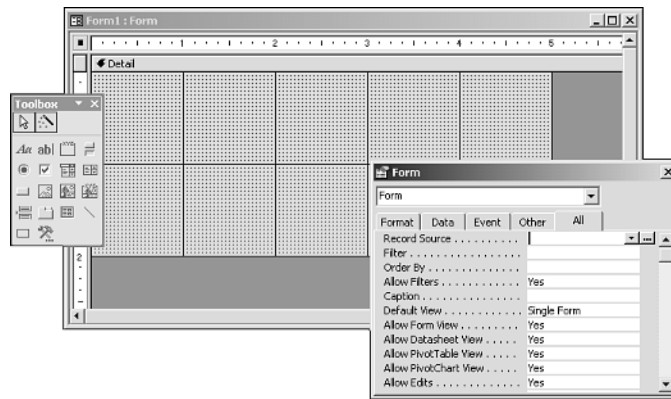


FIGURE 2.1

With the form in Design mode, you can add, remove, and modify form controls as well as write code behind the form.

4. To add a control to the form, select the control from the toolbox. Then click and drag a region on the form to draw the control. Select a command button and add it to the form.

NOTE

If the Command Button wizard appears, click Cancel. To ensure that you understand how to create and customize forms in Access, you won't use the wizards in this chapter. Although the wizards let you work quickly, they hide complexities that you should understand. To disable wizards from appearing every time you add controls, click the Wizard toggle button (the magic wand) in the upper-right corner of the toolbox. When the button is raised, the control wizards are turned on; when the button is down, the wizards are off.

You should be viewing the command button's property sheet. If not, be sure that the command button is selected.

NOTE

If you highlight the button's text, the property sheet becomes blank. When this happens, click the form's background, and then re-click the control's border to select it.

5. On the property sheet, click the All tab. Change the name of the control to cmdHelloButton, and then change the caption to Say Hello.
6. Now write an *event handler* to display a message when the button is clicked. (Event handlers, also known as *event procedures*, are pieces of code that execute when an action occurs.) Right-click the button and choose Build Event. From the Choose Builder dialog, select Code Builder and then click OK.

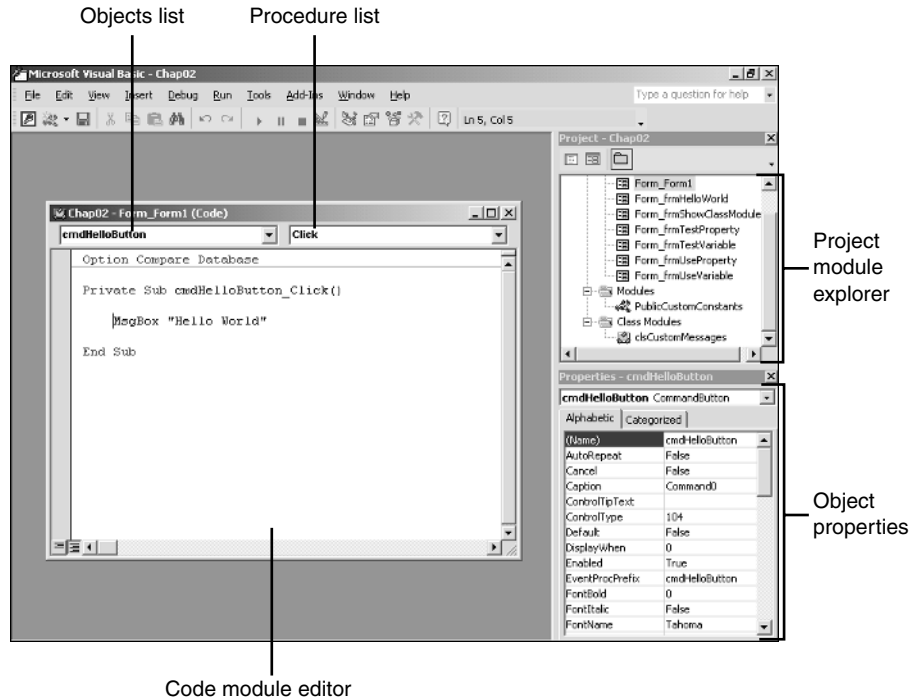
TIP

Because you will be using event procedures instead of macros from now on, you can set up Access to use event procedures by default. To do so, choose Options from the Tools menu. Then click the Forms/Reports tab and select the Always Use Event Procedures check box. Now when you choose Build Event, you will move directly into the code module.

You should be in the VBE's code module editor, into which you type your code (see Figure 2.2). It's organized by object and procedure. Look in the (General) object to find any custom procedures you might have written behind this form as well as variable declarations available to the entire form.

Variables available to the entire form are found under the (General) object's Declarations section. To view procedures relating to other controls or objects on the form, drop down the Objects list and select the object. To change the procedure you're viewing, drop down the Procedure list.

7. In the editor window, type the following, which will display a message box when the button is clicked:
MsgBox "Hello World!"
8. You're ready to run your first form. Close the VBE and select the form. From the View menu, choose Form View.

**FIGURE 2.2**

VBA capitalizes keywords automatically in the code module editor.

TIP

An easy way to switch views is to click the Design button (the first button on the left on the Form toolbar). This button automatically switches you between Design view and Form or Datasheet view, depending on the form type. You can force Access to switch to a different view by clicking the button's down arrow. When you edit a module in the VBE and want to switch back to Form view, choose Object from the View menu.

9. Click the Say Hello button to get a message box displaying your message.

Coding in custom class modules is discussed later in the section “Coding Class Modules.” More advanced examples are in Chapter 16, “Extending Your VBA Library Power with Class Modules and Collections.”

Coding in Standard Modules

Any code or variables not specific to an individual form are written in *standard modules* (code modules that don't exist behind a form or report). With standard modules, it's a good idea to group related procedures into their own standard modules. You don't want to create a single module that contains all your generic code.

You can create and view standard modules on the Modules page of the database window.

TIP

When viewing a subroutine in the code module, you can jump to another subroutine that's being called by pressing Shift+F2 or by clicking the subroutine's name, opening the View menu, and choosing Definition. You can also jump back to the calling subroutine by pressing Ctrl+Shift+F2. Both key combinations can be very useful when working with code.

2

Declaring Variables

The form you just created displays a message box. You didn't have to store or manipulate data with this form. As you develop applications, you often will need to store data temporarily while your application is running. This data can be the result of a calculation or a user entry.

To store this data, you use a *variable*. To create a variable, you must answer the following questions:

- How long and where do you need the information?
- What do you want to name the variable?
- What type of data will you store in the variable?

Defining the Life and Location of Variables

How long and where you're going to use a variable are referred to as the variable's *scope* and *visibility*. You can scope variables to either the procedure level or module level:

- *Procedure level* refers to variables defined within a subroutine. An example of a subroutine is the click event you used in the Hello World example. Variables declared within a subroutine are visible only to that subroutine.
- Variables declared at the *module level* are available to the entire code module—and possibly the entire application.

To specify whether a variable is available only to its code module or the entire application, you use the `Private` (or `Dim`) and `Public` keywords.

NOTE

Unlike the `Public` keyword, `Private` isn't an outright replacement for an existing keyword from Access 2. For module-wide variables (those in the declaration section), you can now use `Private` instead of `Dim`. However, variables still must be declared by using `Dim` when they're contained within subroutines.

Use `Dim` to declare variables in a procedure or module. A variable declared by using `Dim` is available only to the subroutine or code module where it's defined. Variables dimmed within procedures exist only during the procedure's execution. So when you exit the subroutine, all variables declared within it are released from memory. Variables dimmed at a module level live until you exit or recompile the application. For example,

```
Dim strUser as String
```

TIP

At the module level, it's a good idea to use either `Private` or `Public` rather than `Dim`, simply because you then know exactly where you are with the variable. Although perfectly legitimate, `Dim` is actually a leftover from previous versions of VBA.

At the module level, you can create variables that are available outside the module. These variables are defined with the `Private` and `Public` keywords. `Public` variables can be accessed from anywhere within your application; private variables, on the other hand, are similar to those declared with `Dim` and are available only within that code module. (`Private` and `Public` are discussed in much greater detail later in this chapter.) You can declare variables this way in a form's code module and a standard code module. For example,

```
Private pstrUser as String  
Public pstrUser as String
```

You can declare private variables within the module to store information required by each routine within the module. Any information that might be necessary to other modules in your application should be stored in public variables.

Finally, you can declare static variables in procedures. *Static variables* are similar to module-level variables in that they exist until an application is exited or recompiled. This way, you can create subroutines that can keep a value in memory for subsequent calls. For example,

```
Static strUser as String
```

Naming Variables with the Leszynski Naming Conventions

After determining your variable's scope, you need to give the variable an appropriate name. To write more maintainable code, it's a good idea to follow a standard naming convention. The naming convention used in this book is derived from the Leszynski Naming Conventions. This standard prefixes all variable declarations with an abbreviated form of the variable type. This prefix should be lowercase, followed by a name that describes the variable's use. For example, string-type variables are prefixed with `str`. A string variable that holds a user's name can be declared as follows:

```
Dim strUserName as String
```

Any developers looking at this declaration can correctly determine that the variable must contain a string that represents the username. Another benefit of supplying the data type as part of the variable name is that you don't have to find the variable's declaration to determine its type.

Forcing a Variable Declaration with `Option Explicit`

The final step to declaring a variable is always to specify the variable's data type. You should do this even though VBA lets you write code without declaring any variables. Although not declaring variables might sound great at first, in the long run you'll find out that you created a debugging nightmare. Therefore, it's highly recommended that you declare all your variables with the correct data type at the beginning of the subroutine, before writing any other code.

Notice my emphasis here on specifying a data type. Not only should you declare variables, you should always assign them an appropriate data type. Because VBA isn't a strongly typed language (which means you can legally use undeclared variables), it's very easy to write unmaintainable code. When running your code, VBA recognizes undeclared variables and automatically allocates memory for them as a Variant data type. You might wonder why it's so important to require declarations of variables. Consider the following code example, which returns a string in reverse order:

```
Function Reverse (strValue)
    For intLoop = Len(strValue) To 1 Step -1
        strCurrentCharacter = Mid$(strValue, intLoop, 1)
        strNew = strNew & strCurentCharacter
    Next
    Reverse = strNew
End Function
```

This example has quite a few errors. First, `strCurrentCharacter` is misspelled on line 4 (it's missing an `r`). In a very large program, this mistake might be very hard to debug. Worse, because no syntax error is generated, your bug might appear as a random calculation error.

Next, because variables or the return type for the function aren't defined, the preceding example allows any data type to be passed in without error.

By using a small, two-word command on top of every code module in the Declarations section, you can avoid all these types of mistakes. This command is `Option Explicit`, which causes VBA to ensure that all your variables are declared. All undeclared variables (that is, typing errors) generate compile-time errors. In the preceding code example, even if you declared all your variables, you would get a compilation error on the misspelling. The preceding function would now be written as follows:

```
Function Reverse(strValue as String) as String
    Dim intLoop as Integer, strNewString as String
    Dim strCurrentCharacter as String
    For intLoop = Len(strValue) To 1 Step - 1
        strCurrentCharacter = Mid$(strValue,intLoop,1)
        strNewString = strNewString & strCurrentCharacter
    Next
    Reverse = strNewString
End Function
```

NOTE

This routine is strictly to show the issue of `Option Explicit`. VBA 6.x includes a function for reversing a string call, `StrReverse`.

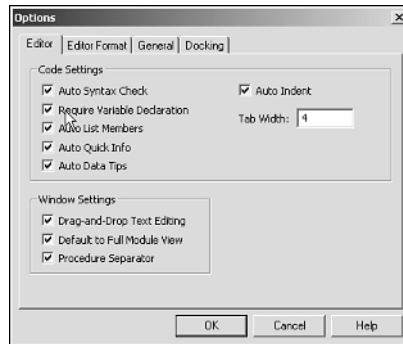
To ensure that you never forget to add `Option Explicit` to your modules, Access provides a convenient option that automatically adds the statement to the beginning of every code module. To turn on this option, follow these steps in the VBE:

1. From the Tools menu, choose Options.
2. Click the Editor tab to see all the settings relating to a code module and the editor.
3. Under Code Settings, make sure that the Require Variable Declaration option is selected (see Figure 2.3).

CAUTION

In Access 97, the Require Variable Declaration option was True by default. To follow other product standards, Access 200x (actually, VBA 6.x) leaves it False by default, so you have to turn it on yourself. Not doing so can cause hours of grief from misspelled variables.

Also, this option affects only new modules. You must manually add the `Option Explicit` statement to any existing code modules.

**FIGURE 2.3**

You can find a detailed explanation of the Editor page in Appendix A, “Debugging Code in Access 2002,” on this book’s Web page at www.sampublishing.com.

CAUTION

Access 2002 strongly types object properties; in versions of Access before 97, properties were considered `Variant`. Because properties are strongly typed, prior versions of your code could potentially break, but the performance advantage of using strongly typed variables is well worth it.

NOTE

By default, the top of your modules will say `Option Compare Database`. This statement—used only within Access, not in other Office applications—specifies how Access does string comparisons within VBA. To learn more about this statement, look up `Option Compare` in Help.

Declaring Procedures

You use procedures to organize your code within code modules. In procedures, you call statements and methods that perform a certain function. You can create two types of procedures: subroutines and functions. *Subroutines* perform an operation without returning a value, whereas *functions* manipulate data and return a value.

You wrote a subroutine earlier in this chapter when you wrote the Hello World button’s click event. All event handlers are subroutines. The difference between an event and your custom

subroutine is that Access calls events automatically in response to some action; on the other hand, your code must call custom subroutines.

An example of a function is the `Reverse` string function used in the preceding section.

Creating a subroutine or function involves four steps:

1. Determine the scope of the procedure (`Private` or `Public`).
2. Declare the subroutine with the `Sub` or `Function` keyword, followed by the name of the routine or function.
3. Determine what information you need to pass to the subroutine. These bits of information—called the subroutine’s *arguments*—are specified in parentheses after the subroutine’s name.
4. Write code that performs the desired operation.

Determining a Procedure’s Scope

Similar to declaring variables, you can also specify whether a procedure is available only to the current code module or to the entire application. This is done by using the `Public` and `Private` keywords introduced earlier in the section “Defining the Life and Location of Variables.”

By default, all procedures are public except for Access event handlers, which are private (you’ll see the `Private` keyword before their declarations). If a procedure is required only within the code module, you should declare it as private. Using `Private` is valuable because it prevents calling a procedure that should never be directly called. For example, a utility function specific to a procedure in the same code module should be made private. With your own class modules, make sure that you use the `Public` keyword with your properties.

The concept of static variables is also explained in “Defining the Life and Location of Variables.” You can also create static functions or subroutines. If you use the `Static` keyword before the `Sub` or `Function` keyword, all variables declared within the subroutine automatically become static. Here’s an example of how to declare a public static function:

```
Public Static Function Increment(intValue as Integer) as Integer
```

You can find a more detailed discussion of the `Public` and `Private` keywords later in the section “Programming with Objects.”

Specifying Arguments in Procedures

Procedures also can take any number of arguments, which pass additional information to procedures. In most cases, when calling a procedure, you must supply every specified argument.

You already saw how to declare arguments in the `Reverse()` string function. For the `Reverse()` example, you needed to pass a single argument, `strValue`. However, nothing

prevents you from declaring no arguments or even many arguments. Arguments are specified with a comma-delimited list. For example, the following function, `SaveFile()`, would take three parameters:

```
Function SaveFile(strFileName as String, strFileExtension as String, _  
    strPath as String) As String
```

NOTE

The underscore character (`_`) is a line-continuation character. You can use it to break up a long line of code into multiple lines.

Again, as when declaring variables, you should define your arguments with an appropriate argument name and an appropriate data type. (Notice that the Leszynski naming standard is also used by subroutines defined in this book in the arguments' names.)

In the `SaveFile()` function, `As String` follows the argument list to specify the type of data that the function is returning. If this part is omitted from the function declaration, the return value is assumed to be `Variant`. Therefore, always specify the appropriate return value.

NOTE

You might come across functions written in VB and specified as follows:

```
Function Reverse$ (strValue as String)
```

The `$` following the function name specifies the return data type of `String`. VBA exposes special characters that can be used as shortcuts to define the data type. This method isn't recommended because anyone reading your code—including you—might not know all the different characters that can be used to declare data types.

Using Enumeration to Control Parameters

Enumeration has been used as far back as Access 95 with Access's and VBA's built-in functions and commands. An example of this is seen in the `DoCmd` object's `OpenForm` method, mentioned in Chapter 1, "Macros Are for Weenies; Code Is Cool!" The `DoCmd.OpenForm` method uses various built-in enumerations, as shown in the following code line:

```
DoCmd.OpenForm "frmTest", acNormal,,, acEFoemdit, acDialog
```

The three parameters used here each have multiple choices. For instance, the second parameter is Form Mode. The possible choices are `acDesign`, `acFormDS`, `acFormPivotChart`, `acFormPivotTable`, `acNormal`, and `acPreview`. These choices are then listed as you type the comma preceding the parameter (see Figure 2.4.)

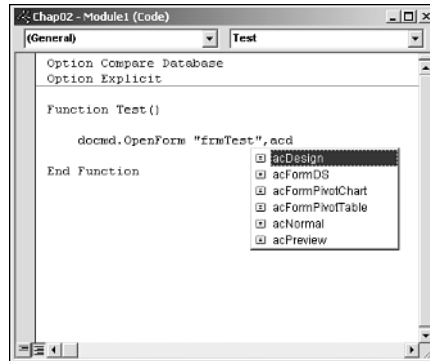


FIGURE 2.4

VBA uses enumeration for some parameters in built-in commands and functions.

The really cool thing with VBA 6.x is that you can have enumerators for your own routines. The syntax for declaring enumerators is

```
Public|Private Enum Name
    Enumerators...
    ...
End Enum
```

An example for using enumerators is lists such as tracking quarters in the year. Here's the syntax for such an enumerator list:

```
Public Enum apQuarter
    qtrFirst = 1
    qtrSecond = 2
    qtrThird = 3
    qtrFourth = 4
End Enum
```

NOTE

If all you need is a list of unique values and don't care exactly what they end up being, you don't need to use the `= 1, = 2` syntax.

The routine call that would use this enumerator would resemble this:

```
Sub ShowQtr(intQuarter As apQuarter)
    MsgBox intQuarter
End Sub
```

Now, when this subroutine is called and the parameter begins, Figure 2.5 shows you how it will look.

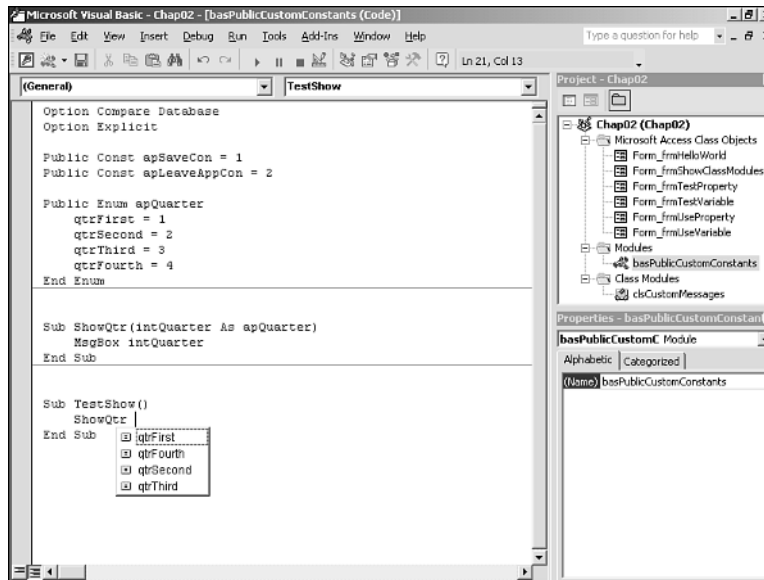


FIGURE 2.5

Using enumerators is easy and can save time trying to remember possible parameters for user-defined routines.

NOTE

Enumeration variables are type equivalent to Longs, so they can be used with Long-type variables, passed as arguments, and returned from functions.

Taking Advantage of Optional Parameters

Sometimes you might need to have one or many optional arguments. VBA supports subroutines and functions that can take any number of optional arguments. Optional parameters must be the last declarations in the prototype. You can declare multiple optional parameters in succession.

You can specify optional parameters in a couple of ways. One way is to specify the optional parameter to be a Variant data type. When you specify the optional parameter as Variant, you can use the `IsMissing()` function to tell whether the parameter was supplied. The following is an example of using an optional argument with the Variant data type and `IsMissing()` function:

```
Function Salutation(strFirst as String, strLast as String, _  
    Optional varSalutation as Variant) as String  
    If IsMissing(varSalutation) Then  
        Salutation = strFirst & " " & strLast  
    Else  
        Salutation = varSalutation & " " & strFirst & " " & strLast  
    End If  
End Function
```

`IsMissing()` checks whether an optional argument is being supplied. When using optional arguments, you should always verify whether the parameter was supplied and respond appropriately; otherwise, you'll experience errors if you try to reference a missing parameter.

You can now call the preceding function as follows:

```
Debug.Print Salutation("Diana", "Barker", "Mrs.")  
Debug.Print Salutation("Diana", "Barker")
```

The other alternative in optional arguments is to specify what the default should be. That declaration would resemble this:

```
Function OpenFile(Optional strSkeleton As String = "*.*) as String
```

Thus, if no argument is specified for `strSkeleton`, the `"*.*"` is used.

TIP

To specify a strong data type for optional parameters without explicitly giving a default value, use `""` for String, `0` for numbers.

Taking Advantage of Parameterized Arrays

VBA provides support for all subroutines and functions to take various numbers of arguments. For example, the following concatenate function appends two strings to each other:

```
Public Function Concat(sStart as String, sAppend as String) as String  
    Concat = sStart & sAppend  
End Function
```

This function is limited to concatenating only two strings. It would be great if you could write a function that could concatenate any number of strings. With VBA, this is possible:

```
Public Function Concat(ParamArray avarArray() as Variant) as String
    Dim varElement as Variant, strTemp as String
    strTemp = ""
    For Each varElement in avarArray
        strTemp = strTemp & varElement
    Next
    Concat = strTemp
End Function
```

In this function, the keyword `ParamArray` specifies that anywhere from 0 to n number of parameters will be specified. `ParamArrays` must be declared as `Variants` and must be the last argument of any declaration. One difference between `ParamArray` and `Optional` arguments is that you can define only a single `ParamArray` at the end of an argument list.

Also, the `For Each` loop iterates automatically over all the array's elements. This functionality, also present in collections, is discussed in detail later in the section "Getting Dizzy with Loops."

Calling this function can now be done with any number of parameters:

```
? Concat()                ' Returns an empty string
? Concat ("Hi ", "Out ", " There", "!") ' Returns Hi Out There!
? Concat (1,2)             ' Returns 12 (not 3)
```

NOTE

You can't use `Optional`, `ByRef`, or `ByVal` with `ParamArrays`. `ParamArrays` are optional by definition, as you can provide anywhere from 0 to n values. `ParamArrays` are implicitly `ByVal`, which means that you're working on a copy of the variable.

Passing Arguments ByRef or ByVal

The effect of passing arguments to subroutines or functions isn't as obvious as you might expect. When you pass a variable to your subroutine or function, what happens if you modify the variable's value? Does this change the value in the calling subroutine? The following code demonstrates the importance of distinguishing between `ByVal` and `ByRef`:

```
Sub WhatsMyValue()
    Dim intX as integer
    intX = 10
```

PART I

```
SquareIt(intX)
    MsgBox intX
End Sub

Sub SquareIt(intSquare as Integer)
    intSquare = intSquare * intSquare
End Sub
```

When the line `MsgBox intX` is executed, what's the value of `intX`? Is it 10 or 100? In this example, the answer is 100. This change in value can have important implications on your program. You might not want a subroutine to modify its parameters. This can create another common pitfall and debugging nightmare: As with using undeclared variables without `Option Explicit`, no visible error is generated.

You can define the behavior of your subroutine parameters. By default, all parameters you define are considered by reference. Simply defined, this means that the subroutine is operating directly on the passed-in variable.

The other option, by value, causes your subroutine or function to operate on a copy of the variable. When a parameter is specified by value, a copy of the variable is made in memory and discarded at the end of the subroutine. Thus, if a parameter is specified `ByVal`, modifying the value of the parameter doesn't change the source variable.

NOTE

One disadvantage to `ByVal` is the small amount of overhead needed to create the extra copy of the variable. Memory is also needed to store the extra copy of the variable. Also, you can use `ByVal` only with the basic data types; you can't use it with user-defined types or object-valued variables.

Using `ByVal` parameters is as simple as supplying the `ByVal` keyword in the parameter list. For example, in the `SquareIt` procedure earlier, if you want `intSquare` to be specified by value, you would rewrite the subroutine's header as follows:

```
Sub SquareIt(ByVal intSquare as Integer)
```

NOTE

The same rules hold true for parameters in functions. By default, all parameters are passed by reference.

Controlling Program Flow

VBA provides different ways to control your application's flow. You can control your program's flow with loops and decision-making commands. You've already seen one type of loop in the `Reverse()` function, another in the `Concat()` function, and decision-making code in the `Salutation()` function earlier in this chapter.

VBA also supports the capability to branch your code by using `Goto` and `GoSub` statements. Extensive use of these statements, however, often leads to unmaintainable code and isn't recommended unless they're used for error handling. Error handling is discussed in Chapter 7, "Handling Your Errors in Access with VBA."

Making Decisions in Your Code

Decision-making code is when you run different code based on different situations. For example, in the `Salutation()` function shown earlier in the "Taking Advantage of Optional Parameters" section, you checked whether the optional parameter was specified. VBA exposes two different decision-making statements: `If...Then` and `Select...Case`.

The `If...Then` statement is very useful for running different code depending on an expression's value. Before dividing two numbers, for example, you should first check to see whether you're dividing by zero:

```
If intB <> 0 Then
    intReturn = intA / intB
Else
    ' Generate an Error, or perform special processing.
End If
```

You can also nest `If...Then` statements to check for numerous conditions. VBA provides an easier mechanism for nesting conditional code that results in cleaner, more maintainable code—the `Select...Case` statement. Imagine that you're writing code depending on the date of the week:

```
Select Case strWeek
    Case "Saturday", "Sunday"
        ' Process the weekend
    Case "Friday"
        ' Process Friday
    Case Else
        ' All other days.
End Select
```

Getting Dizzy with Loops

Writing loops allows you to repeat a block of code until a certain condition is met. The following types of loops are supported:

- Do...Loop
- For...Next
- For Each...Next

The loop you use depends on the situation. Do...Loop repeats until a specific condition is reached. With Do...Loop, you can have three different syntaxes: one with the condition at the bottom of the loop, another with the condition at the top, and one without a condition. For example, you can continue to loop until the user enters a username. This condition might be met after one iteration or after many. Thus, this loop might be written as follows:

```
Do
    strUserName = InputBox("Please enter your User Name", "Logon Screen")
Loop While strUserName = ""
```

This is one of a few different syntaxes Do...Loop supports. One checks the condition before the code block executes, as in the preceding logon example; another checks the syntax after the code block executes.

Checking the condition before the loop executes has a slightly different syntax than checking the condition at the end of the loop. The following syntax checks for the condition before the loop executes. In this case, the code within the loop might never be executed, depending on whether the condition is met.

```
Do While | Until condition
    statements
Exit Do
    statements
Loop
```

In the following syntax, the loop always executes at least once because the condition isn't checked until the end of the loop:

```
Do
    statements
Exit Do
    statements
Loop While | Until condition
```

NOTE

In the preceding syntax, you can supply a condition only with the Do or the Loop keyword, but not both at the same time. The condition is optional. You can write a loop that loops unconditionally. When this occurs, the loop continues infinitely or until an Exit Do statement is executed.

There's a difference between the `While` and `Until` keywords. You use `While` when you want the loop to continue while a certain condition is being met. You use the `Until` keyword when you want to loop until a condition is met. For example, the condition

```
Loop While x = 10
```

is the same as

```
Loop Until x <> 10
```

To force an exit from `Do...Loop` before the condition is met, use the `Exit Do` statement. This forces the loop to exit without executing any additional lines of code.

NOTE

VBA supports an additional type of loop, `While...Wend`, which is functionally equivalent to `Do While...Loop`. However, `Do...Loop` is more flexible because it can loop until a condition is or isn't met, and it allows you to force the loop to occur at least once. `While...Wend` is limited to looping only while a specific condition is met.

The `For...Next` loop is useful when you know the number of times you need to repeat. For example, to loop 10 times, you use

```
For intLoop = 1 To 10  
    ' Run Code  
Next intLoop
```

TIP

Although the `intLoop` on the `Next` statement is optional, including it each time saves a lot of confusion.

Similar to the `Exit Do` statement in `Do...Loop`, you can force a `For...Next` loop to exit at any point by using an `Exit For` statement.

Loops can also run in reverse. You already saw one example of a backward-stepping `For...Next` loop with the `Reverse()` function earlier in the section "Forcing a Variable Declaration with `Option Explicit`." To run the preceding loop backward, use

```
For intLoop = 10 To 1 Step -1  
    ' Run Code  
Next intLoop
```

Without the `Step` statement, the loop increments one at a time. You can also use `Step` with integers larger and smaller than 1 to cause the loop to iterate in multiples.

VBA also supports a `For Each...Next` Loop statement. This special type of loop iterates over collections as discussed later in the section “Using Collections.”

Indenting Rules for Control-Flow Code Blocks

To make your code more readable, indent all control-flow-related blocks. Such code includes all types of loops and `If...Then` statements. For example, the following code loops and checks the state of a variable:

```
For intLoop = 1 To 10
    If intLoop = 1 Then
        ' Do something
    End If
Next intLoop
```

As with comments, indenting code has no effect on your code’s execution. Adding tabs doesn’t slow down your code’s execution, but it does make following your control flow much easier.

Commenting Code

All comments in the code are specified by an apostrophe. Comments are ignored during application execution. Comments such as the following explain the flow of the program and make your code more maintainable:

```
' The strUserName variable is used to cache the email name
' of the current user.
```

Compared with writing code, comments are often considered wasted time. However, they are one of the most important ways to make your code more maintainable. For the short term, you probably can remember how you wrote your code, but imagine looking at your code after weeks or months pass. Using comments is one way to ensure that you still can understand decisions and algorithms well into the future.

Handling Errors

Unfortunately, many predictable and unpredictable errors might occur in your application. Your application must be prepared to respond to errors at all times. If you write an Access application without any error handling, depending on the type of error, your user might get an indecipherable error message, or the whole program might shut itself down without warning.

In Access, two types of errors can occur: synchronous, which occur during code execution, and asynchronous, which occur outside code execution. The following sections introduce these error types. Chapter 7 provides a deeper discussion on responding to errors.

NOTE

Many errors that can occur when your code isn't running can also occur when your code is running. For example, an Out of Memory error can occur while your code is being executed (causing a synchronous error) or when no code is running (causing an asynchronous error).

Errors While Code Is Running

Synchronous errors occur during code execution. If no error handler is present, these errors can often halt your application. Again, because it's almost impossible to anticipate all errors that can occur, you should do your best to trap the most common errors and generically handle the rest. If an error occurs that you didn't anticipate, it's usually a good idea to notify the user that an error occurred and you're shutting down the application. A graceful exit is always much cleaner than having users read a confusing Access error message.

To trap synchronous errors, use the `On Error` command at every entry point to your code. This usually is any event-handler code. Unless you want to do special processing within functions or subroutines called by the event handler, it's not always necessary to have an event handler in every routine.

Errors While Code Isn't Running

Asynchronous errors occur while your code isn't running. There might be very little you can do to handle these errors. Also, they often might be unpredictable because you usually have no control over when they occur. All you can do is respond to the error and, in many cases, gracefully close your application. Examples of asynchronous errors are when a network error occurs or when you run out of disk space. Such errors can occur at any time and are independent of any code you might have written.

When asynchronous errors occur in a form, an Error event is fired to give you an error identifier and an error message. By using the error identifier, you can determine the error that occurred and respond appropriately. You can also use the Error event to display a clearer error message than the one Access provides.

Getting Started with VBA

VBA supports object-oriented features, a browser for viewing an object's functionality, and the capability to define custom public and private properties and methods. The rest of this chapter introduces you to these powerful features.

Introducing Objects

VBA adds some basic object-oriented features. Your application consists of many objects. An *object* can be thought of as a single unit that contains code and data that serve a common purpose. The following are object examples:

<i>Object</i>	<i>Description</i>
Text box	All controls are objects.
Form	Every form in your application is a separate object.
Table	Each table in your application contains other objects, such as fields and indexes.
Excel	Other applications that support Automation are also objects.

Objects have properties and methods, which are the exposed parts of the object. Exposed properties and methods can be either public or private. For example, the capability to reference a control's value is done through a public property, `Value`.

All functionality internal to the object is considered private. All private members (properties and methods) are accessible only to the object itself. For example, writing a private method on a form means that the method can be called only from within the form. No other form knows about nor can call that method.

All code behind forms used to be private. Before Access 95, you had to resort to poor and complex programming techniques to expose your form functionality. Often, this was done through global and static variables and subroutines written in global modules. The disadvantage to this approach was that your functionality wasn't encapsulated. By exposing the internal guts of your forms through global variables and subroutines, you make it very easy for someone to accidentally change the state of your form. By being able to hide, or *encapsulate*, all the internal workings within the form, you avoid this problem. You now can selectively expose any parts of your form by defining those parts as public.

What's more, Access lets you create multiple instances of objects, so you can display and manipulate multiple copies of the form. Imagine displaying a customer form a few times, one for each customer. The manipulation of multiple instances is introduced in Chapter 4, "Working with Access Collections and Objects," and discussed again in Chapter 16.

Using the Object Browser

Access 2002 supports a large number of different objects. Every control that you can insert on a form is also an object. What's more, through Automation, you can use VBA to program other Automation objects, such as Microsoft Word or Excel. You can even use C++ or Visual Basic versions 4 and later to develop your own Automation applications.

Because you can access a large number of objects, you need a mechanism to manage them. VBA provides this through the Object Browser, which makes it easy to view and obtain information about all the public properties and methods of any object (see Figure 2.6). The Object Browser can

- Show all objects available to your applications, as well as the properties and methods associated with each object. You can also browse any other object available on your system.
- Allow you to quickly locate and go to different procedures in your code.
- Enable you to get help via the Help button on generally any property, method, or object.
- Allow you to paste the syntax of an object's method or property into your code. This helps you remember the list of parameters and helps prevent typing errors.

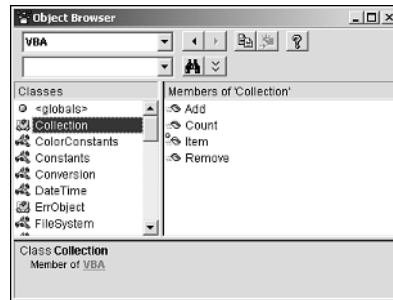


FIGURE 2.6

The Object Browser lets you view information about all objects your application references.

Displaying the Object Browser

The Object Browser is available only in the VBE. In the following steps, you open a code module so that you can use the Object Browser:

1. Create a new code module by going to the Modules page on the database window and clicking New. You will now be placed in the VBE.
2. From the View menu, choose Object Browser, or press F2.
3. Select the different objects from the Projects/Libraries drop-down list (the top drop-down list). Selecting a library displays all the objects within the library.
4. Clicking a class (object) in the Classes list displays all members (methods and properties) in the class. If a Help file is available for the object, clicking the ? button displays the associated Help topic.

By selecting a library and an object, you can view all the object's public methods and properties. Selecting an individual method or property gives you the appropriate syntax. At this point, you can paste the syntax into your code.

Adding References to Other Objects

By adding references to any object on your system, you can use the referenced object in your application and the Object Browser to view the object. To add references, follow these steps:

1. In the VBE, choose References from the Tools menu.
2. In the References dialog, select the name of the application or object (see Figure 2.7).

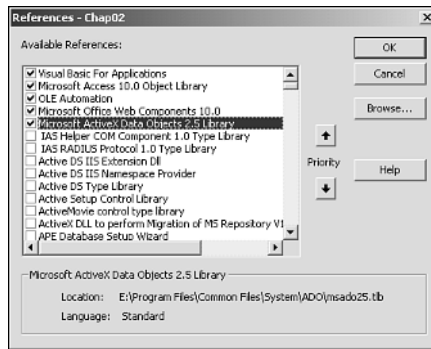


FIGURE 2.7

By setting references ahead of time, you can access all objects included in that reference.

If the object or application isn't listed, you can browse for it. Object libraries have .tlb or .olb file extensions. Because executable files, DLLs, and OCXs also can contain object libraries, those extensions are also available.

You also can reference other databases you have created. When you do this, you can open the forms and call the functions and procedures of the other database. This way, you can create a common procedure library that all your applications can share.

CAUTION

The order of the objects defines the order in which they're searched. This can cause problems with your code if two libraries contain objects of the same name.

After you make a reference to an object, you can call the methods and properties available in that object.

Programming with Objects

Earlier, the section “Defining the Life and Location of Variables” introduced you to the concept of variable and procedure scoping by using the `Public` and `Private` keywords. You also can create and reference your own custom properties and methods on an object. *Custom properties* are essentially variables for which you can write custom routines when they’re assigned or retrieved. For example, the `Visible` property on the form functions like a variable, but when you assign a value to it, it either hides or shows the object. With VBA, you can now create your own variables that provide special behaviors.

NOTE

Method is essentially another term for a subroutine or function. Methods act on the object they’re contained in. For example, the `Repaint` method on the form tells the form to repaint itself.

2

CODING IN
ACCESS 2002
WITH VBA

The Public Keyword

As a quick review from the earlier discussion, you make variables and procedures public with the `Public` keyword. A public variable is available anywhere within your application. If the variable or procedure is in a standard module, you don’t have to use the `Public` keyword. If it’s behind a form or report, or part of another object, you first need to reference the object. If you expose a public variable `intTest` on a standard module, for example, you can reference it as follows:

```
Debug.Print intTest
```

However, if the same variable is exposed on a form named `Form1`, you must first reference the form, as follows:

```
Debug.Print Form1.intTest
```

Public variables are declared only in the general declaration section of a code module. Variables declared within a procedure are available only to that procedure. Because variables and procedures are public in standard modules by default, the `Public` keyword is optional.

The Private Keyword

By default, all variables and procedures are public except for event handlers, which are located in a form’s or report’s class module. Private variables and procedures are available only within the procedure.

Using Properties and Methods

The capability to program an object's properties and methods isn't new to Access. Access forms and controls have always had both. For example, the `Visible` property on a control causes a control to be either visible or invisible. The `Requery` method on a form restarts the underlying query.

Using Existing Properties

Access provides a large number of properties. Almost all actions in Access involve setting a property. However, the user interface hides the complexities surrounding properties. For example, a form consists of many properties that establish a caption on the title bar, Minimize and Maximize buttons, a background color, and a certain size.

Access supplies property sheets that let you view and set values for each property for the form and controls. In the beginning of this chapter, when you created the Hello World form, you used the form's property sheets. To display a property sheet, follow these steps:

1. Select the Forms tab from the database window.
2. Select the Hello World form and click the Design button to open the form in design view.
3. Right-click the black box in the upper-left corner. From the pop-up menu, select Properties if it isn't already selected. (Choose Select Form from the Edit menu if the property sheet is already open.)

You should now see a form onscreen with the title Form and a number of tabs. The property sheet is divided into a number of categories. You can view properties by categories, or click the All tab to view all properties grouped.

The property sheet provides an easy way to set properties. To change a property, simply find it on the property sheet and enter the new value. The new value takes effect immediately.

Setting Property Values

Setting an object's properties from code is very simple. Use the following syntax:

```
ObjectName.PropertyName = expression
```

The dot separates the object from the property being called on the object. Therefore, to set the caption property of the form from within the form, use the following code line:

```
Me.Caption = "My Form"
```

The `Me` operator references the form itself when you're writing code within the form. Chapter 4 provides a more detailed discussion of the `Me` operator.

Getting Property Values

You retrieve property values when you want to perform actions depending on the state of the object. For example, you can check the state of a check box and then run code based on whether the check box is now selected. You get the value of properties with the following syntax:

```
VariableName = object_name.property_name
```

If you want to run code based on the state of the check box every time it's clicked, you can do it as follows:

```
Sub CheckBox1_Click()  
    If CheckBox1.Value Then  
        ' Run code when checkbox1 is true  
    Else  
        ' Run code when checkbox1 is false  
    End If  
End Sub
```

In this example, the value of the check box isn't stored in a variable. Instead, the returned value of the property is being used directly. If you're writing code in which the value of the property is being used multiple times, it's more efficient to store the value in a variable.

Using the Default Property

Many objects also provide a default property, which you can use to simplify your code. The default property is returned when you reference the object without specifying a property. For example, what would happen if you had a text box named Text1 inserted on a form and wrote the following line of code?

```
Me.Text1 = "Hello Out There"
```

Assigning a value to the control actually is setting the Text property. This line of code is the same as typing

```
Me.Text1.Value = "Hello Out There"
```

For the TextBox control, the Value property is the default property.

Every object has a different default property. For most controls, the default property relates to the value being displayed. On the form, the default property is actually another object, the Controls collection. (Collections are discussed in greater detail later in the section "Using Collections.")

TIP

To determine the default property for any object, look for the little icon with the blue dot in the Object Browser.

Setting Multiple Properties with the With Statement

VBA supports writing more readable code for performing multiple actions on a control. For example, if you want to set three properties of a text box named `text1` on your form, you would need to write the following code:

```
text1.BackColor = 0  
text1.Width = 200  
text1.Height = 400
```

Rather than write all this code, you can write a `With...End With` code block to accomplish the same thing:

```
With text1  
    .BackColor = 0  
    .Width = 200  
    .Height = 400  
End With
```

This purely cosmetic feature doesn't change how your code is executed. The advantage to using the code block is that it becomes more obvious that you're modifying many properties of the same object.

Using Existing Methods

Objects are also composed of any number of methods. A *method*, as defined earlier, is similar to a function or subroutine. As with properties, methods have existed for some time on forms and controls. Again, as with properties, VBA allows you to define your own additional public methods.

Referencing methods is similar to referencing properties. The primary difference is that a method is either a subroutine or function and therefore can't be assigned a value. Calling methods follows this syntax:

ObjectName.MethodName

For example, to call `Requery` on a form, type the following:

Me.Requery

Some methods might take parameters or supply return values. If you're calling a method that doesn't have a return value or you don't care about the return value, separate the arguments with a comma. For example, calling the `GotoPage` method on a form requires that you specify to which page you want to move; you also can optionally specify a right and down offset. Calling this method on the form is as follows:

```
Me.GotoPage 2,100,100
```

If the method returns a value that you want to use, you must enclose the arguments in parentheses. This works similarly to calling a function.

Specifying Named Parameters

All methods provided in Access allow referencing of an argument by its name rather than its position. You can identify methods that support named arguments in the Object Browser.

Normally, when you call a method, you pass the arguments in the order specified by the method. Now, you can provide the parameters in any order by providing the argument's name. The syntax for using named parameters is as follows:

```
ObjectName.MethodName Parameter1:=expression, _  
    Parameter2:=expression,...Parametern:=expression
```

TIP

You can use named parameters and find their syntax with user-defined routines as well. Use the Auto Statement Builder to take full advantage of this.

For example, the `GotoPage` method takes three parameters. Two parameters, offset right and down, are optional. By using named parameters, you can omit the right parameter and still specify a down parameter. To call the `GotoPage` method to move to page 2, down 100 twips, you would write the following method call:

```
Me.GotoPage PageNumber:=2, Down:=100
```

NOTE

Access exposes all screen measurements through VBA in twips. There are 1,440 twip units in an inch. A *twip* is a special unit of measurement that's screen-resolution independent. Therefore, regardless of your screen's resolution, elements are displayed correctly.

You can use named parameters rather than leave blank commas. Suppose that you have five parameters but want to use only the first and last. With named parameters, you use only the two instead of all five. Named parameters also help make your code self-documenting because remembering what the parameters do is a lot easier when you can see their names right there.

Assigning Objects to Variables

Assigning an object to a variable uses a slightly different convention than assigning a value to a variable. When using objects, you assign objects to a variable by using the `Set` statement:

```
Set variable = ObjectReference
```

To understand the difference between object assignment and variable copying, think back to the `ByVal` and `ByRef` discussion earlier in the section “Passing Arguments `ByRef` or `ByVal`.” Normally, you’re copying variables. Consider the following code fragment:

```
Dim intX as integer
Dim intY as integer
intX = 10
intY = intX
intX = 20
MsgBox intY
```

The value of `intY` is 10 when it appears in the message box. When the line `intY = intX` is executed, the current value of `intX` is copied into `intY`. Any subsequent changes to `intX` aren’t reflected in `intY`. However, when the `Set` statement is used, a reference pointing to the object is made, not a copy. Consider the following code fragment:

```
Dim txtName as TextBox
Me.UserName = "Bill"
Set txtName = Me.UserName
Me.UserName = "Joe"
MsgBox txtName
```

The value `txtName` displayed in the message box is `Joe`. Furthermore, any changes made to the `txtName` variable are immediately reflected in the `UserName` text box. Assigning an object to a variable is useful when you want to perform a number of actions on a single object.

NOTE

Always use the `Set` statement for object variables, never for data variables.

Using Generic Object Types

Access and VBA also supply generic object types. You declare generic-type variables when you're unsure of the object type to assign. VBA supports the following generic object types:

<i>Type</i>	<i>Description</i>
Object	Supplied by VBA, this is a generic object. Any type of object can be assigned to this type.
Control	Supplied by Access, this variable can take any type of control.

The following code uses the generic `Control` object type to align all controls to the specified position. (To loop over all the controls, this example also uses a special syntax, which is discussed in greater detail later in the section “Using Collections.”) You must declare the variable as type `Control` because the form might consist of a number of different control types.

```
Sub AlignLeft(intLeft as Integer)
    Dim ctlCurrent as Control
    For Each ctlCurrent in Me
        ctlCurrent.Left = intLeft
    Next
End Sub
```

NOTE

Using the generic objects such as `Control` can be convenient, but a performance hit is involved. Always use specific types where possible. If you know a control will be a text box, for example, declare the variable as `TextBox`.

Identifying the Object Type

When using a generic object type, you often want to know what type of object is actually being referenced. Identifying control types is very important because you might want to perform an action that only certain control types support. There are three ways to determine an object's type. The first and second ways work for all objects; the third can be used only with control objects.

To identify an object's type, you can use the `TypeOf` operator, which works only within an `If...Then` statement:

```
If TypeOf Object is ClassName Then
    ...
End If
```

NOTE

You can't use the `OfType` operator in a `Select...Case` statement. You must use a nested `If...Then` statement if you want to check whether the variable contains one of a number of different objects. There's a more convenient way to check the control type being referenced. Every control has a property, `ControlType`, that returns an integer identifying the control. You can use this property in a `Select...Case` statement.

The second way is to use the `TypeName (variable)` function. `TypeName` passes the variable to be tested; then a string representing the variable's type is returned.

Lastly, in the following example, the code iterates over all the controls on a form and appends a colon (:) to all labels:

```
Sub FixLabels()  
    Dim ctlActive as Control  
    For Each ctlActive in Me  
        If ctlActive.ControlType = acLabel Then  
            ctlActive.Caption = ctlActive & ":"  
        End If  
    Next  
End Sub
```

Access has defined constants that represent each built-in control. To find the list of constants, press F2 in the VBE and look up `acControlType` in the Classes list. You then see the constants in the Members list.

Using Collections

As mentioned earlier, every form in your application usually has a number of controls. To reference a control, you first have to reference the form. Because a form can consist of many controls, these related controls are grouped in a *collection* object.

Collections are a special type of property that consist of related objects. For example, your database consists of a collection of tables, and each table has a collection of fields and indexes. There are two data access methods for accessing these collections: *Data Access Objects (DAO)* and *ActiveX Data Objects (ADO)*. ADO is discussed in Chapter 5, "Introducing ActiveX Data Objects." Information on DAO is available at www.sampublishing.com; just type this book's ISBN in the Search field.

Collections are object-valued properties, meaning that the collection property's data type is a specific object type. Because collections are objects, they can contain properties, and some collections provide methods. Most collections provide support for iterating over their elements, as well as provide a count property that returns the number of elements.

A user-defined collection works very much like an array. However, unlike an array, you don't need to dimension the collection because you can add to and remove from the collection dynamically. You also don't need to worry about garbage collection if you remove an item from the middle of the collection. With arrays, such garbage collection and memory management are often quite complicated. (Programming your own custom collections and a comparison between them and arrays can be found in Chapter 4, with more advanced examples in Chapter 16.)

To understand collections, you need to examine a form's control collections. First, you need to prepare a new form and insert a few controls on it. Follow these steps:

1. Open the database window by pressing F11.
2. Click the Forms tab.
3. Click the New button.
4. Select Design View from the dialog and click OK.
5. If the toolbox isn't visible when the form appears, choose Toolbox from the View menu.
6. Use the TextBox control on the toolbox to create three text boxes on the form. Set the Name property of the first text box to `txtMyName`, the second to `txtLocation`, and the third to `txtControlCount`.
7. Use the CommandButton control on the toolbox to create a command button on the form. Set the Name property to `cmdControlCount` and Caption to `Control Count`.

Counting the Number of Elements

Collections provide a convenient mechanism to determine the number of elements. Unlike arrays, which require you to calculate the elements by using the upper and lower boundaries, collections expose a `Count` property, which you can use to find out how many controls you created on the form in the preceding section. Follow these steps:

1. Right-click the `cmdControlCount` command button and select Build Event. Next, select Code Builder and click OK.
2. Type the following line of code in the event handler:

```
Me.[txtControlCount] = Controls.Count
```

NOTE

Although they aren't needed here, you must place square brackets around controls that have spaces in their names when referencing them from code.

3. Click back on the form. Switch to Form view by clicking the Design View button or by choosing Form from the View menu.
4. Click the Count button to display the number of controls on your form. The displayed count will include any labels—those that are independent as well as text box labels.

Accessing Elements of the Collection

Accessing individual elements of a collection is quite similar to accessing array elements. You will now access different controls on your form:

1. Switch your form back to Design view.
2. Create a second command button. Set its name to `cmdGetName` and its caption to `Control Name Demo`.
3. Right-click the button and choose Build Event. Next, choose Code Builder and click OK.
4. Type the following line of code:

```
Me.txtMyName = Me.Controls(1).Name
```

TIP

You actually reference controls on the Controls collection through the `Item` property as `Me.Controls.Item(1)`. However, because the `Item` property is the default property, you don't need to specify the `Item` property but can directly access elements by using `Me.Controls(1)`.

5. Click back on the form and switch to Form view.
6. Click the Control Name Demo button to display the name of the first control on your form.

Another advantage to collections is that they can have a unique string name. On an Access form, each control *must* have a unique name. This name is also used as a key in the collection. Thus, you can rewrite the preceding code to access the `txtMyName` control, as follows:

```
Me.txtMyName = Me.Controls("txtMyName").Name
```

In this example, the string `txtMyName` is returned.

TIP

When referencing a control, it's almost always better to reference it by name rather than its ordinal index. This is because as you add and remove controls from a form, code that references a control by index isn't always guaranteed to point to the same control.

Although preceding examples demonstrate different ways to reference controls, there's an even easier way. The default property of the form is the `Controls` collection. Therefore, you can directly reference controls on the form as follows:

```
Me.MyControl.Name
```

Iterating Over Collections

VBA also provides an easy technique to iterate over each element in a collection. This way, you can write code that can perform an operation on every element in the collection. You can either write a loop, or use a new syntax provided by VBA. To loop over the collection of controls, you could type the following code:

```
Dim ctlCurrent as Control, intLoop as Integer
For intLoop = 0 to Me.Controls.Count - 1
    Set ctlCurrent = Me.Controls(intLoop)
    ' Process the control ctlCurrent
Next
```

A simpler way to iterate over a collection is to use the `For Each` syntax:

```
Dim ctlCurrent as Control
For Each ctlCurrent in Me.Controls
    ' Process the control ctlCurrent
Next
```

This code automatically iterates over every control in the collection. Notice that the first example uses the `Set` statement, which you should remember is required to assign an object in VBA to a variable. Omitting the `Set` statement would generate a runtime error.

Back to your sample form. You need to add a button that causes all text boxes to align left. Follow these steps:

1. Switch your form back into Design view.
2. Create a third command button. Set its name to **cmdAlignLeft** and caption to **Align Left**.

3. Right-click the command button and choose Build Event. Next, choose Code Builder and click OK.
4. Type the following code:

```
Dim ctlCurrent as Control
If Me.txtLocation >=0 Then
    For Each ctlCurrentIn Me.Controls
        ' Process the control ctlCurrent
        If ctlCurrent.ControlType = acTextBox Then
            ctlCurrent.Left = Me.txtLocation
        End If
    Next
Else
    MsgBox "You need to supply a location to align to."
End If
```
5. Click back on the form, and then switch the form to Browse view.
6. Enter a value into the Location text box. Click the Align Left button. All your text box controls will align left at that location.

NOTE

Collections built into Access are *0-based collections*, meaning that you must subtract one from the number of elements in the collection if you iterate over the collection in a loop. On the other hand, collections *you* define are all 1-based, and you should loop over your collections from 1 to the actual count of the collection. Other collections in Office products are also 1-based, so you have to double-check various collections when you use them.

Customizing a Form

In Access 2, you had to live with a predefined set of properties and methods. Now, VBA lets you define your own additional form properties. With VBA, you can create properties and methods that can be referenced just as though they were built in. You're now ready for the next step of adding public properties and methods to the form.

NOTE

The forms created in this section are on the book's Web page at www.sampublishing.com in the Chap02.mdb file, located in the \Examples\Chap02 folder.

Writing Custom Properties

You can expose a property of a form in two ways. The first way is simply to expose a form variable as public by placing the keyword `Public` in front of the variable's declaration. Follow these steps:

1. Create a new form in Design view. From the View menu, choose Code to display the code window. In the code module's declaration section, type the following line:
`Public intLockForm as Integer`
2. Create a button on the form. Type **Show Lock** for the caption; then right-click the button and select Build Event. Click Code Builder and then click OK.
3. Type the following line of code in the event handler:
`MsgBox intLockForm`
4. Name the form **frmUseVariable** and save it.
5. Open the form in Form view.
6. Create a new form and place a command button on it. Name the button **cmdLockForm** and set its caption to **Lock Form**.
7. Right-click the button and select Build Event. Click Code Builder and then click OK.
8. Type the following code in the event handler:
`Form_frmUseVariable.intLockForm = _
Not Form_frmUseVariable.intLockForm`
9. Save the form as **frmTestVariable**.
10. Switch the form to Form view. Click the Locked button.
11. Activate the `frmUseVariable` form and click the button. The value of `intLockForm` is displayed. If you go back to the other form and click the button, `intLockForm` changes.

You've created your first public form property. However, you have no way to respond to changes in the property's value. VBA provides such functionality through two special subroutines: `Property Get` and `Property Let`. These subroutines allow you to write code that's executed every time the variable is set or returned.

Now, create a new form called `frmUseProperty`. You will create a `LockForm` property that has a little more intelligence than the `intLockForm` variable saved on the `frmUseVariable` form in step 4. The `LockForm` property, when set to `True`, locks all the controls that can be locked on the form; when set to `False`, it unlocks all the controls on the form.

NOTE

When you use the Leszynski Naming Conventions, internal object properties don't include the data type in the name. Hence, the variable named `intLockForm` becomes the property `LockForm`.

Follow these steps to create the `LockForm` property:

1. Declare a private variable to store the state of the lock. Under the declarations section of the form, type the following code line:

```
Private intLock as Integer
```

2. Write the procedure that returns the value of `intLock`. Returning values of properties is done by using the `Property Get` statement:

```
Property Get LockForm as Integer  
    LockForm = intLock  
End Property
```

3. When setting the `LockForm` property, you need to lock all controls on the form. The code for assigning properties uses the `Property Let` statement:

```
Property Let LockForm(intValue as Integer)  
    Dim ctlCurrent as Control  
    On Error Resume Next  
    For Each ctlCurrent in Me  
        ctlCurrent.Locked = intValue  
    Next  
    intLock = intValue  
End Property
```

The `On Error` statement in this code ensures that errors aren't generated for controls that don't provide a locked property. When such a control is encountered, it's skipped.

4. In the form's `Load` event, initialize `LockForm` to false with the following code:

```
LockForm = False
```

5. Add a few more text box controls to the form.

NOTE

Because I'm using the `Locked` property to demonstrate using custom properties, you must have bound controls to lock. Therefore, I've created a table named `tblUseProperty` with three `Text` type fields in it: `Text1`, `Text2`, and `Text3`. Figure 2.8 shows this table structure and the `frmUseProperty` form in Design view.

Another issue is that you can't lock unsaved bound forms. So in the routine attached to the OnDeactivate event of the frmUseProperty, you want the following code:

```
Private Sub Form_Deactivate()
    Application.RunCommand acCmdSaveRecord
End Sub
```

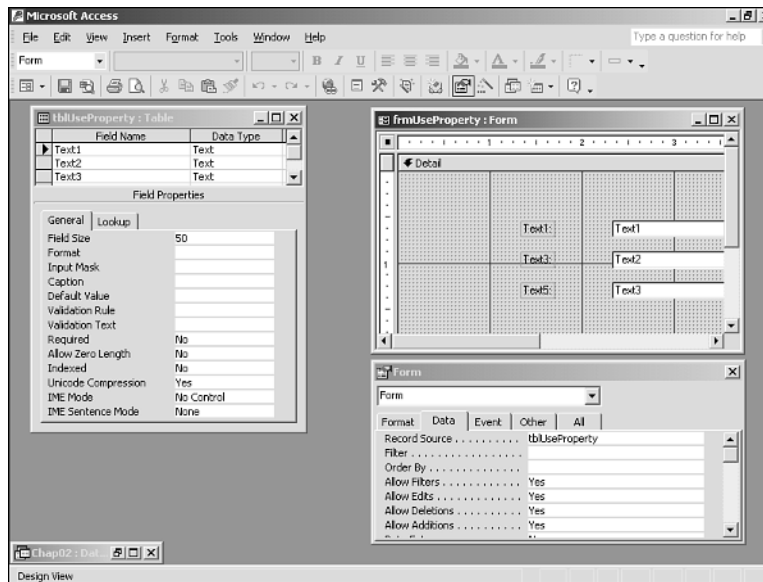


FIGURE 2.8

The table tblUseProperty supplies the text boxes for the frmUseProperty form.

6. Save the form and switch it to Form view.
7. Type something in the text boxes.
8. Copy the frmTestVariable form to one called frmTestProperty. Switch to the frmTestProperty form in Design view.
9. Modify the line of code behind the cmdLockForm command button's OnClick event from


```
Form_frmUseVariable.intLockForm = _
    Not Form_frmUseVariable.intLockForm
```

 to


```
Form_frmUseVariable.LockForm = Not Form_frmUseVariable.LockForm
```

10. Add the following code after the line you just changed to have the `cmdLockForm` command button's caption reflect the lock state of the `frmUseProperty` form accordingly:

```
If Forms!frmUseProperty.LockForm Then  
    Me.cmdLockForm.Caption = "Unlock Form"  
Else  
    Me.cmdLockForm.Caption = "Lock Form"  
End If
```

The `cmdLockForm_Click` routine would then resemble this:

```
Private Sub cmdLockForm_Click()  
    Forms!frmUseProperty.LockForm = _  
        Not Forms!frmUseProperty.LockForm  
    If Forms!frmUseProperty.LockForm Then  
        Me.cmdLockForm.Caption = "Unlock Form"  
    Else  
        Me.cmdLockForm.Caption = "Lock Form"  
    End If  
End Sub
```

11. Open the `frmTestProperty` form in Form view and click the button.
12. Return to the `frmProperty` form. You should no longer be able to type in any of the controls, and the `cmdLockForm` caption should say `Unlock Form`.

You just created your first custom property. You can set as well as retrieve the property's value.

TIP

To create a read-only property, just write a `Property Get` subroutine. Without the corresponding `Property Let` routine, no value can be assigned to the property, but a value can be returned.

NOTE

When writing code within the form, reference the `intLock` variable directly. Because assigning a value directly to the internal variable doesn't cause the form to get locked, you should call the property instead of the private variable, even when writing code within the form.

Writing Object-Valued Properties

Object-valued properties return a value or can be assigned an object. For example, when you use the `Set` statement to assign a control to a variable, you're using an object-valued property. You also can write code that returns objects by using the `Property Set` statement. Chapter 4 provides an example of returning object-valued properties.

Writing Custom Methods

The `LockForm` property demonstrated earlier also can be easily created as a method as follows:

```
Public Sub LockForm(intLock as Integer)
    Dim ctlCurrent as Control
    On Error Resume Next
    For Each ctlCurrent in Me
        ctlCurrent.Locked = intLock
    Next
End Sub
```

This method works as expected. Calling `LockForm(True)` locks the form, and calling `LockForm(False)` unlocks the form. However, there are fundamental differences in the calling code. Using the `LockForm` method:

```
LockForm True
```

Accessing the `LockForm` property:

```
LockForm = True
```

With a property, you're assigning a value to the property. If you use a method, there's no way to easily determine the value of `LockForm`. Rather than create a `Public Sub`, you can create a `Public Function` that returns a value. This value can be used to indicate the success or failure of the operation. However, this still doesn't provide a way to determine the current lock state.

Public methods and functions are useful when you want an action to occur that doesn't remember any state. Examples of methods in Access are the `Repaint` method on the form, which causes the form to paint itself, and the `Requery` method on a list box, which causes the contents of the list to be retrieved from the database.

Coding Class Modules

Thus far, you've seen how to use standard and form modules, which are also referred to as *class modules*. You can also create your own class modules that you can use as objects, complete with properties and methods.

The next example gives you an idea of how to start using your own class modules, but it's up to you to figure out when your applications can take full advantage of them. As with other examples in this chapter, you can find this example in the Chap02.mdb file in the \Examples\Chap02 folder on this book's Web page at www.sampublishing.com.

TIP

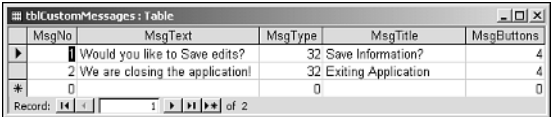
After you work your way through this discussion, you can check out Chapter 16 to see how to use class modules in a real-world application.

As with forms, you can use Get, Set, and Let for adding custom properties to your class module object, and you can use Sub and Function commands for creating methods. The best way to show you this is to jump right into the example. Class modules are mainly for you, the developer, whereas many other features in Access allow you to create something cool for users.

This example allows you to add custom error messages to your application into a table. Then, by using a constant, you call a method of the class module object clsCustomMessages, which is created to display the message desired. Next, you retrieve the message box button that was clicked. This class module has only one property, UseBeep, which turns the beep on and off when messages are displayed through the CustomMessages type object. I'll point out other features as the example progresses.

Creating the Support Objects

Although it would be nice to always create a totally black box object with no outside support objects, you sometimes end up creating a more unusable object that way. Such is the case here. For creating custom messages (in addition to the class module you'll create), you use a table for the message information, called tblCustomMessages (see Figure 2.9) and a standard module to create corresponding constants for the messages.



MsgNo	MsgText	MsgType	MsgTitle	MsgButtons
1	Would you like to Save edits?	32 Save Information?		4
2	We are closing the application!	32 Exiting Application		4

FIGURE 2.9
This table supplies the messages to use with the class module.

The constants used are kept in the PublicCustomConstant module. Here's the module listing:

```
Option Compare Database
Option Explicit
```

```
Public Const apSaveCon = 1
Public Const apLeaveAppCon = 2
```

When you need to add a new message, add an entry to `tblCustomMessages` and create a new public constant.

Creating the Class Module

Creating separate class modules is basically a combination of creating standard modules (in the module tab of the database window) and creating code behind forms, adding custom methods and properties. If you look at the Modules page of the database window for `Chap02.mdb`, you see the `clsCustomMessages` class module along with `basPublicCustomConstants`, the standard module. Notice that they have different icons beside their names, based on which type of module they are (see Figure 2.10).

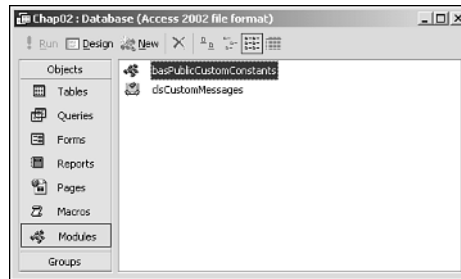


FIGURE 2.10

Two types of modules can now be seen in the Modules page of the Database window.

To create a class module, from the Insert menu choose Class Module. At the top of the module window, the words `Class Module` appear beside the name; in the standard module, the word `Module` appears.

The `clsCustomMessages` class module consists of one method and one property. The method, `DispMessage`, takes the constant passed in and looks up the message number in the `tblCustomMessages` table. The `MsgBox()` function is then called by passing the fields that were entered for the desired message. The return value of the `MsgBox()` function is passed back.

NOTE

When a method passes back a value, a function is used. If no value is to be passed back, a routine can be used instead.

Listing 2.1 shows the code for the DispMessage method.

LISTING 2.1 Chap02.mdb: Displaying a Custom Message

```
Function DispMessage lngMsgNo As Long) As Integer
    Dim rsMessages As New ADODB.Recordset
    With rsMessages
        '-- Retrieve the desired message
        .Open "Select * from tblCustomMessages where MsgNo = " & _
            lngMsgNo, CurrentProject.Connection, adOpenStatic
        '-- If the UseBeep property is set to True, beep
        If blnBeep Then
            Beep
        End If
        '-- Display the message
        DispMessage = MsgBox(!MsgText, _
            !MsgType + !MsgButtons, !MsgTitle)
    End With
End Function
```

This method also looks at the Boolean value of blnBeep. This variable is actually the internal value of the UseBeep property that has been specified by the following Property Get and Let routines; the Let routine sets blnBeep to the value passed in, and the Get routine merely returns the current status of blnBeep.

```
Property Get UseBeep() As Boolean
    UseBeep = blnBeep
End Property
```

```
Property Let UseBeep(blnBeepArg As Boolean)
    blnBeep = blnBeepArg
End Property
```

The blnBeep variable is declared in the class module's declarations section:

```
Option Compare Database
Option Explicit
Dim blnBeep As Boolean
```

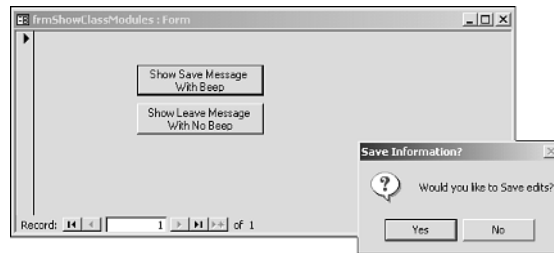
Using the Class Module

Now that the class module is created, it's time to test it. To do so, I created a form in Chap02.mdb named frmShowClassModules. This form has two command buttons, which correspond with the two custom messages that have been included. In each button's OnClick event, a reference has been created to the CustomMessages object. Listing 2.2 shows the code for the first button, which calls the DispMessage method with the apShowCon constant.

LISTING 2.2 Chap02.mdb: Calling a Custom Class Module's DispMessage Method, Setting the UseBeep Property First

```
Private Sub cmdShowMsg1_Click()  
    Dim clsMessages As New CustomMessages  
    clsMessages.UseBeep = True  
    If clsMessages.DispMessage(apSaveCon) = vbYes Then  
        MsgBox "Yes, I will save"  
    Else  
        MsgBox "No, I won't save"  
    End If  
End Sub
```

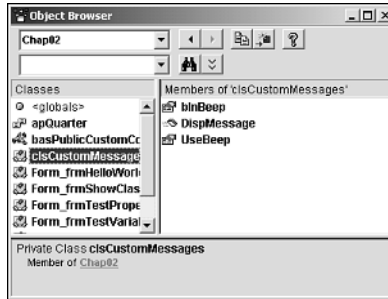
The first statement uses the `New` keyword to declare a variable to be a new instance of the `CustomMessages` class module. Next, the `UseBeep` property of the `CustomMessages` type object is set to `true`. Then the `DispMessage` method is called, and the return value is examined. Figure 2.11 shows the object in action.

**FIGURE 2.11**

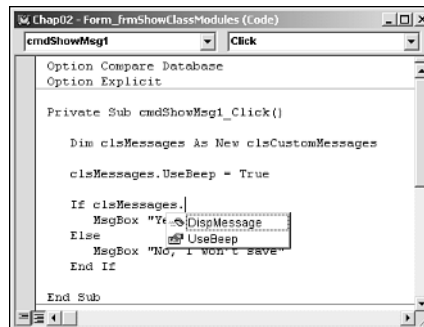
Creating objects to perform specific tasks is now possible, as with the `CustomMessages` type object.

One benefit of using class modules is that the object now shows up in the Object Browser (see Figure 2.12). Another benefit is that Access uses the object and displays the properties and methods as you use them in the module editor (see Figure 2.13).

Listing 2.3 shows the last routine, used with the second command button on the form. The only notable difference between this routine and the first button's (shown in Listing 2.2) is that the `UseBeep` property isn't set. By not setting this property, it defaults to `false`.

**FIGURE 2.12**

You can view properties and methods of class module objects inside the Object Browser.

**FIGURE 2.13**

Getting Auto Tip information on custom objects can save a great deal of time.

LISTING 2.3 Chap02.mdb: Not Setting a UseBeep Property

```
Private Sub cmdShowMsg2_Click()
    Dim clsMessages As New CustomMessages
    If clsMessages.DispMessage(apLeaveAppCon) = vbYes Then
        MsgBox "Yes, I will leave"
    Else
        MsgBox "No, I won't leave"
    End If
End Sub
```

NOTE

When you click the first button (and therefore set the UseBeep property to true), it doesn't stay set for the second button. As two separate instances of the object, they don't see each other or the properties set.

Summary

You've seen Visual Basic for Applications, a common language used within the Microsoft Office products. You now have the foundation for programming Access 2002 and for the rest of the book. To build on the foundation, see these chapters:

- Chapter 4, "Working with Access Collections and Objects," shows the various collections and objects that make up the Access object model.
- Chapter 5, "Introducing ActiveX Data Objects," explains ActiveX Data Objects (ADO), Microsoft's preferred method for working with data.

Making Access Project and Data Technologies Choices

CHAPTER

3

IN THIS CHAPTER

- Using Microsoft Database Versus Access Database Project 72
- Using DAO Versus ADO Versus XML 75

Before Access 2000, when you created an application, you had to plan whether you wanted to use a Jet (the database engine Access uses natively) or a client/server back end. Depending on the type of back end chosen, you would use various methods to create the application. Although this is still the case, you now have more choices to make when starting on a project. Now, in addition to the back-end choice, you must decide what type of database container to use: the traditional MDB or ADP.

The other choice to make is the method of accessing the data while working in VBA: DAO (Data Access Objects), which has been used since Access 2, or ADO (ActiveX Data Objects), the data-access technology now used throughout a number of Microsoft products, including Visual InterDev. You now have an additional choice of XML (Extensible Markup Language), which transfers data between systems and applications.

Helping you decide which technology to use in which case is what this chapter is all about. When you first look at the choices, it's hard to see when one should be used over another.

NOTE

This chapter isn't intended as an in-depth discussion of these technologies. Each is discussed at length in various parts of the book. At the end of this chapter you can see where each technology is discussed further.

Using Microsoft Database Versus Access Database Project

The choice of whether to use MDB versus ADP actually follows the Jet versus client/server choice pretty closely. ADP, used as a front end for SQL Server, contains tools for editing not only Access objects such as forms and Data Access Pages, but also views and stored procedures.

Microsoft also provides a desktop solution to use with the ADP, known as the Microsoft Data Engine (MSDE). This is aimed at small workgroup applications. Generally, you want to use an ADP with MSDE when you think you will be moving your application to the full functionality of SQL Server 7/2000.

TIP

You can also use the MSDE to test your ADP and then easily modify the connection string to hook you up to the production database. Chapter 24, "Developing SQL Server Applications by Using ADP," covers more about MSDE.

Looking at the Objects Used in Each

Your standard MDB contains the usual objects, plus the Data Access Pages (DAPs) available as of Access 2000. Data Access Pages let you create an application in Access, and then move it over to the Web. You can read more on DAPs in Chapter 12, “Working with Data Access Pages.”

Figure 3.1 shows the standard Access Database window displaying the Northwind sample database that comes with Access.

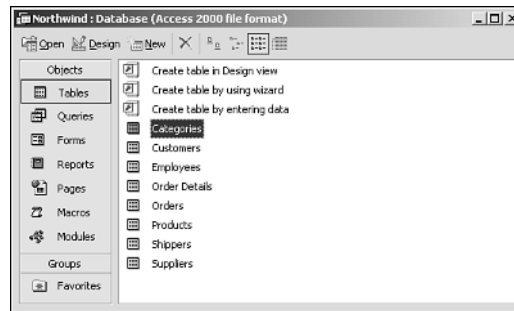


FIGURE 3.1

Here is the good old MDB you’ve come to know and love.

NOTE

Features found in both ADP and MDB are Data Access Pages and the Outlook style toolbar. The Groups feature (refer to Figure 3.1) is great when you’re working on one area of a project and want to group multiple objects for development purposes.

In Access 2002, you can choose to work with Access 2000 or 2002 file formats (notice the title bar in Figure 3.1). Both Northwind.mdb and NorthwindCS.adp are provided in the Access 2000 file format.

You can take the same Northwind database and upsize it to SQL Server and an ADP by choosing Database Utilities, Upsizing Wizard from the Tools menu while in the Database window. (If you decide to do this, you must have the MSDE installed and running or be connected to SQL Server. For more information on how to do this, see Chapter 24.)

After the upsizing is done, you see the ADP in Figure 3.2, which, although it looks similar to the standard MDB, has different objects. Table 3.1 lists the objects found in each and how they correspond with each other.

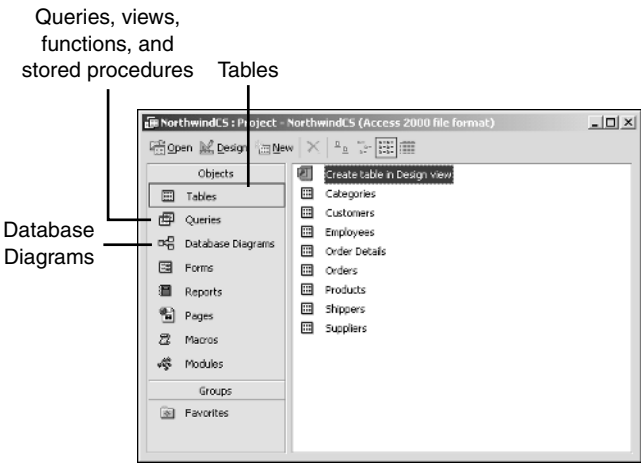


FIGURE 3.2
In a ADP, the objects in Tables, Queries (now consisting of views, functions, and stored procedures), and Database Diagrams are actually stored in the back end.

TABLE 3.1 Comparing Objects Between MDB and ADP

<i>MDB</i>	<i>ADP</i>
Table	Table*
Relationship	Database Diagram*
Select Query	View*
Action Query	Stored Procedure*
Parameterized Query	Stored Procedure*
Form	Form
Report	Report
Data Access Page	Data Access Page
Macro	Macro
Module	Module

* *Stored in the back-end database on the server.*

TIP

Stored procedures can actually buy you everything that views can and more, although they're a little harder to create. One drawback to views is that you can't

specify ordering. You can use the View Designer to create a SELECT statement and then copy the SQL text into the stored procedure editor, where you can add parameters and ordering. For more on this, check out Chapter 24.

Using DAO Versus ADO Versus XML

How you design your application also varies with which front-end type you use. The choice of using DAO versus ADO is fast becoming a moot point because Microsoft is aiming that way, made even more apparent in Access 2002 since the company made only bug fixes to DAO, not really any enhancements.

As mentioned at the beginning of this chapter, DAO has been around for several versions of Access and has pretty well been the standard for database manipulation with Jet from VB and all Office products.

Then along came ADO. ADO came on the scene because DAO was written and optimized mainly for Jet. The idea is that with the way the world is going with the Internet and multiple data sources, an easier, more generic way to access that data was believed to be needed—hence, ADO.

NOTE

In Access 2000/2002, not all functionality is provided in ADO to replace DAO. For example, when using objects and methods such as `Me!subForm.Requery`, the DAO object model is still used under the covers.

Where using DAO requires a reference to one object model, ADO requires three to cover most of the same ground. You can see this in Figure 3.3, by choosing References from the Tools menu in the VBE.

As just mentioned, DAO uses one library, Microsoft DAO 3.6 Object Library, whereas ADO uses three (see Table 3.2). Chapter 5, “Introducing ActiveX Data Objects,” discusses each library in more detail.

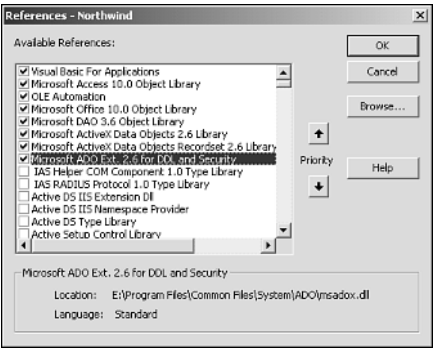


FIGURE 3.3
When converting to ADO from DAO, you can have code that references both in the same application.

TABLE 3.2 ADO Libraries and Their Purpose

<i>Library</i>	<i>Purpose</i>
Microsoft ActiveX Objects 2.6 Library	Data manipulation
Microsoft ADO Ext. 2.6 for DDL and Security	Data definition and security
Microsoft Jet and Replication Objects 2.6	Replication

TIP

Whichever version you want as the default, place it before the other. For example, if I want to use ADO recordsets as the default, I move its library up before DAO's. This is important because when you `Dim` something as a recordset, you want to get the right type. Otherwise, errors can occur. You can also use the `Library.ClassName` syntax:

```
Dim rstX As ADODB.Recordset
Dim dynY As DAO.Recordset
```

NOTE

In revising this book for Access 2002, I decided to fall into line and switch all the code to ADO. However, if you want to see most of the examples in DAO, you can pick up a copy of my *Microsoft Access 2000 Power Programming*. Trying to cover both ADO and DAO to the extent of all the examples would make this book about 2,000 pages long.

Just because you're using SQL Server as a back end doesn't automatically mean that you should use an ADP with ADO. There are still good reasons to use Jet-linked table SQL Server applications in many situations:

- Your application requires storage on the front end, such as user preferences or SQL strings in tables. In an ADP, you have to use the Registry or file system for such storage.
- Your application requires users to be able to create and store ad hoc queries, and you can't or don't want to give the users permissions to create server objects.
- Linked-table applications are still better at joining information from heterogeneous data sources (ISAM stuff).

Although I mentioned that ADO is harder to use than DAO in the previous edition of this book, it really is just a matter of getting used to the new syntax. Microsoft also has enhanced the ADO object model since it was first introduced.

The bottom line is that DAO is fading in future versions of Microsoft products. It probably will be supported in a couple more versions, but even DAO 3.6 hasn't had any enhancements made. So if you're using DAO now, start looking at converting your applications; however, if an application doesn't have major changes in store, I would hold off.

If you're just starting to develop applications in VBA and Office products, or even have to start a new application, jump on the ADO bandwagon. Just know that there are some issues you might have to work around.

Regarding XML, it's not really a choice of using either ADO or XML exclusively, but when to use each. As I mentioned, XML allows you to read and write data to and from other systems and applications. It was created originally to use with the Internet, but is also useful between different file formats locally.

So when you have a database application that needs to interface with another system, especially other file formats, you can use XML to perform this function. When you get the data into Access or SQL Server tables, use ADO. You can read more about XML and ADO in the chapters listed in the Summary section.

Summary

In Access 2002, it seems as though there are more choices to make than ever. These choices are definitely ones that you need to make before you delve into creating an application. For more information on the technologies discussed in this chapter, see these chapters:

- Chapter 5, "Introducing ActiveX Data Objects," gives an overview of the ADO object models and how to use them in your applications.

- Chapter 6, “Using XML with Access 2002,” gives an overview of using XML within Access 2002, both through the interface, and using the MSXML object model.
- Chapter 23, “Moving Workgroup Applications to Client/Server,” talks about what it takes to move your applications to a client/server platform and the issues involved.
- Chapter 24, “Developing SQL Server Applications by Using ADPs,” goes into detail about using the new Access Database Project for a front end.
- You also can read more about the object model for DAO and how to use it at www.sampublishing.com. Just type this book’s ISBN in the Search field and look for Appendix C, “Working with Data Access Objects.”

Working with Access Collections and Objects

CHAPTER

4

IN THIS CHAPTER

- **Creating Custom Collections** 80
- **Comparing Custom Collections to Arrays** 83
- **Accessing the Access Object Model** 86
- **Programming Multiple Copies of the Same Form** 104

In Chapter 2, “Coding in Access 2002 with VBA,” you learned how to create custom methods and properties, and were introduced to the basics of collections. In this chapter, you build on those basics to learn about the other objects available in Access 2002. You can find this chapter’s code examples in the \Examples\Chap04 folder on the book’s Web page at www.sampublishing.com. In the Chap04.mdb database, you find code modules that contain all the sample functions and subroutines, as well as forms to demonstrate event-handling code.

Creating Custom Collections

Chapter 2 introduced you to Access 2002’s object-oriented concepts. In that chapter, you accessed the various properties and methods of Access objects, as well as defined custom properties and methods. You also learned of a special type of object, the *collection*, which allows you to maintain lists of elements without worrying about the complexities involved in managing the list.

Consistent with your ability to create custom properties and methods, VBA also lets you create your own custom collections. The most obvious use of this feature is to replace arrays. Arrays are very difficult to manage, especially for adding and removing elements. By using collections in place of arrays, you can avoid many of these difficulties. Collections offer these advantages over arrays:

- They are dynamically allocated, which means that they take up only as much memory as required. Arrays, on the other hand, require you to predefine a size, and changing the size requires reallocating the entire array.
- Collections automatically keep a count of the elements and support easy addition and removal of elements. Arrays, on the other hand, require you to calculate the size based on the upper and lower bounds, and removing elements is difficult.
- Collections easily support adding elements in any position. Arrays support adding elements only through lots of code.
- Each element of a collection can be a different data type. Only a variant array supports different data types. To avoid bugs, however, making the elements of each collection a uniform data/object type is a good programming practice.

The following section shows how to create collections of basic data types as well as collections of objects. You also see a simple comparison demonstrating the differences between coding for arrays and coding for collections.

Defining a New Collection

Collection is a valid object data type in VBA. Therefore, to create a new collection, you declare it as

```
Dim colExample as New Collection
```

This example creates an object that supports all the features of a collection. This includes iterating over the collection by using `For Each`, by using the `Count` property, and by adding and removing items.

In the preceding declaration, notice the `New` keyword, which VBA uses to create new instances of objects. Because a collection is another type of object, you must use `New` to create a new collection. However, when you're declaring a variable to hold a reference to an existing collection, you declare the variable without the `New` keyword.

Of course, you can also do the following rather than create the collection while dimensioning a variable:

```
Dim colExample As Collection
...
If colExample Is Nothing Then Set colExample = New Collection
```

More examples of the `New` keyword are discussed later in the section “Supporting Multiple-Form Instances,” where you learn how to create and manage multiple instances of forms.

NOTE

Although this chapter introduces custom collections and multiple form instances, you can see more advanced examples in Chapter 16, “Extending Your VBA Library Power with Class Modules and Collections.”

Adding Items to the Collection

To add items to your collection, call the `Add` method on the collection. The syntax for `Add` is

```
object.Add(item, Optional key as String, Optional Before|After)
```

Table 4.1 explains the `Add` method's arguments.

TABLE 4.1 Add Method Arguments

Argument	Description
<i>item</i>	The element to be added to the collection. <i>item</i> can be of any data type.
<i>key</i>	An optional string name used to uniquely identify the element. You're already familiar with unique keys. In the control's collection, the name of the control is the key value. Trying to add a non-unique key to the collection generates a runtime error.

TABLE 4.1 Continued

<i>Argument</i>	<i>Description</i>
Before	Specifies which element to place the item before. You can specify the ordinal position or the unique key.
After	Similar to the Before argument, but specifies which element to place the item after.

You can specify only the Before or the After parameter, not both. To specify After, you need to use a named argument. When using Before or After, specifying a position greater than the number of elements or specifying an invalid key generates a runtime error. Omitting Before or After causes the element to be placed automatically at the end of the collection. For example, to add the value 10 after the third position in a collection called `colIntegers`, use the following code:

```
colIntegers.Add 10, After := 3
```

CAUTION

All built-in collections in Access are 0-based, but user-defined collections are 1-based. This means that the index of the first element in your collection is 1; when writing loops over the collection, you shouldn't subtract 1 from the count. The syntax for using a For Each loop to iterate over the collection is the same in all cases. Not understanding the difference between starting 0-based or 1-based can cause you to miss an element or might result in an error if you try to access an element beyond the boundaries of the collection.

Removing Items from the Collection

Because you can add items to a collection, the reverse must also be true. You can remove items from a collection by using the Remove method. The syntax for the Remove method is

```
object.Remove(index)
```

index must be either the position of the item in the collection or a key of an existing element. Similar to adding elements to specific positions of the collections, specifying an index or key that doesn't exist generates a runtime error.

To clear all elements in a collection, you must write a loop that iterates over each element. Remembering that user-defined collections are 1-based, you can write the following routine to clear a collection:

```
Sub ClearCollection(colClear As Collection)
    While colClear.Count > 0
        colClear.Remove (1)
    Wend
End Sub
```

Notice that a `While` loop was used instead of `For Each`. The `Remove` method removes the element's index position from the `Collection` object. The `For Each` loop would return each element of the collection. Therefore, because you can't remove an element from the collection through the element, you can't use `For Each`. Instead, as demonstrated in the preceding example, you should manually iterate over the elements in the collection to remove items.

Comparing Custom Collections to Arrays

Now you know the basics for declaring a new collection and for adding and removing elements from the collection. In the following sections, you create a simple collection of integers, as well as an array of integers. The array takes three elements, calculates the number of elements, removes an element, and displays all the elements. You should recognize the advantages of collections at the completion of this example.

Creating a Collection of Integers

The code in Listing 4.1 declares a new collection by using the `New` keyword, adds elements to the collection, accesses an element of the collection, lists the elements of the collection, removes an element from the collection, and then displays the number of elements in the collection. In short, Listing 4.1 demonstrates each major feature of collections. Outside remembering to use the variant data type, the code is quite intuitive and easy to follow. You can find the `ShowCollectionFeatures()` function in the `modDisplayCollectionFeatures` module.

LISTING 4.1 Chap04.mdb: Demonstrating a Collection

```
Sub ShowCollectionFeatures()
    '1. Declare the new collection
    Dim colDemo as New Collection, varElement as Variant

    '2. Add 3 elements to the collection
    colDemo.Add 10
    colDemo.Add 20
    colDemo.Add 30

    '3. Print the 2nd element to the debug window
    Debug.Print colDemo(2)      ' Displays 20 in the debug window
```


LISTING 4.1 Continued

```
'4. Iterate over the collection
For Each varElement in colDemo
    Debug.Print varElement
Next

'5. Remove Elements from the collection
colDemo.Remove(2)

'6. Print the Count of elements
Debug.Print colDemo.Count ' Displays 2
End Sub
```

NOTE

You must recognize that the variable holding each element is declared as a variant. In For Each loops, the return type must be an object or a variant. Because this collection consists of integers, not objects, the variant type must be used.

Creating an Array of Integers

Using an array is more cumbersome than using a collection. Declaring an array, setting values to the elements, and retrieving values from the array is no more difficult than using a collection. However, when you need to add more elements than you declared (or you need to remove elements), the array becomes more difficult to use.

Listing 4.2 demonstrates all the concepts performed in the collection example in Listing 4.1. However, Listing 4.2 doesn't demonstrate how you reallocate the array to grow larger. Reallocating arrays is demonstrated after Listing 4.2.

LISTING 4.2 Chap04.mdb: Comparing Arrays to Collections

```
Sub ArraySample()
'1. Declare the array. The Array is set to contain 3 elements
Dim arrIntegers(2) As Integer
Dim intLoop As Integer
Const RemoveElement = 0

'2 Set positions 1 through 3
arrIntegers(0) = 10 'Arrays are 0 based by default
arrIntegers(1) = 20
```

LISTING 4.2 Continued

```
arrIntegers(2) = 30
Debug.Print arrIntegers(1)

'Estimate number of elements
Debug.Print UBound(arrIntegers) - LBound(arrIntegers)

' Remove the first value from the array
' Move subsequent values up in the array
For intLoop = RemoveElement To UBound(arrIntegers) - 1
    arrIntegers(intLoop) = arrIntegers(intLoop + 1)
Next
```

End Sub

In this example, array limitations are apparent:

- If you need more than three elements, you must redimension the array. To define arrays that can be redimensioned, you must declare the array as follows:

```
Dim arrIntegers() As Integer
```

- By declaring an array this way, you can redefine its size as needed by using the `ReDim` keyword:

```
ReDim Preserve arrIntegers(2) as Integer
```

The optional `Preserve` keyword preserves the data in the existing array. You can't change the data type of a resized array, and you're limited to changing only the last dimension of multidimensional arrays.

- The example in Listing 4.2, which shows how to print the number of elements by using the `UBound()` function, isn't accurate because it represents only the size of the array. For example, calling that line of code after removing an element still returns 2.
- Removing an element is inefficient if you want to keep the elements together because you would need to iterate over all subsequent elements and move them up in the order. Depending on whether the item is near the beginning of the list, you might have to move quite a few elements.

If you compare Listing 4.1 with Listing 4.2, you can see that collections are usually superior to arrays, but that doesn't mean you should never use an array. When you need to store only a fixed number of elements, arrays might be faster and more convenient to use. They are also more intuitive, if you need a multi dimensional matrix. However, if the number of elements you're storing varies and elements are frequently added and removed, collections almost always offer you better memory management and efficiency.

Understanding Advanced Uses of Collections

Collections have many more uses than just managing integer lists. Each element of the collection can be of a different data type. Furthermore, collections aren't limited to the basic data types, which means you can have collections of objects. In the later section "Supporting Multiple-Form Instances," you learn how to use custom collections to manage your forms.

Accessing the Access Object Model

You already know about parts of the Access object model. When referencing controls and forms while writing code, you're actually manipulating a piece of the Access object hierarchy. In the following sections, you learn how all the pieces fit together.

Access uses a hierarchy of built-in objects. The Access object hierarchy starts with the Application object. From the Application object, you can obtain and manipulate the Screen object, the active form, and collections of open forms, reports, and data access pages.

Figure 4.1 visually represents this hierarchy.

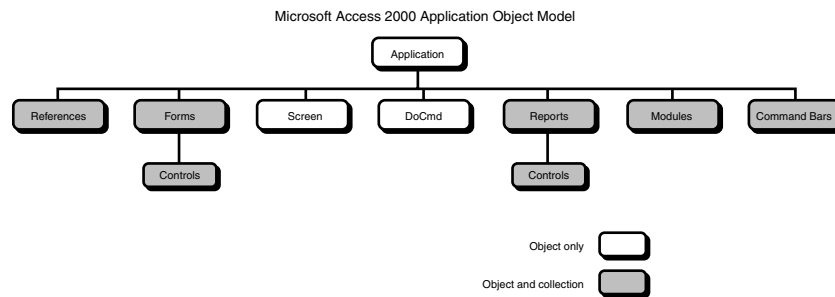


FIGURE 4.1

The Application object provides useful support for your application.

Using the Application Object

The Application object supports properties, methods, and collections for doing the following:

- Turning on and off screen painting for the entire application.
- Setting and retrieving the values of the different options. All the options in the Options dialog are available. Setting and retrieving options are discussed later in the section "Setting and Retrieving Option Values."
- Quitting your application.
- Determining the current active object and whether the object is a table, query, form, report, macro, or module.

- Changing the default menu bar for your database.
- Obtaining an `hWndAccessApp` (Windows handle) for manipulating the application window with the Windows API. `hWnd` is for the Form and Report objects.
- Controlling Access from another application through Automation.
- Calling actions on the `DoCmd` object. The `DoCmd` object contains actions that you can call in Access and your database for manipulating records, calling macros, and much more.
- Accessing descriptions of objects directly through `CurrentData` and `CurrentProject` objects instead of using DAO's Container collection.
- Checking to see if a broken reference is in your project, and examining information about the current references being used.
- Specifying and manipulating printers.
- Obtaining the Forms, Data Access Pages, and Reports collections, which include all the currently loaded forms, data access pages, and reports.

NOTE

In Access 2002, a number of methods and commands that were part of DAO (specifically in the `Workspace` object) have been brought into the `Application` object. Just to name a few are `CompactRepair`, `ConvertAccessProject`, `CreateAccessProject`, and `CreateNewWorkgroupFile`. To get a complete list of the `Application` object properties and methods, look up the `Application` object in the Object Browser.

Turning Screen Painting On and Off

Access supports two ways to control screen repainting. You can turn screen painting on and off globally for the entire application, or individually per form or per subform.

To control the application's painting, you can use the `Echo` method on either the `Application` object or the `DoCmd` object. Both methods have the same result.

TIP

Use the `Application` object rather than `DoCmd` wherever possible for more logical code. The `DoCmd` object has been around since version 2 of Access and exists mainly to mirror the macro commands. The `Application` object method follows the object model paradigm more closely.

The syntax for the Echo method is

```
Application.Echo echo_on, statusbartext
```

The Boolean *echo_on* parameter indicates whether painting is on (true) or off (false). The optional *statusbartext* parameter specifies the text to display on the status bar until the next call of the Echo method.

CAUTION

Turning off screen repainting of your application causes no repaints to occur until you turn screen repainting back on. This is true even if you stop your program. Therefore, when debugging, be careful when you use the Echo method. If you turn it off without turning it back on, Access will never repaint the screen.

To minimize the difficulty of debugging when you use the Echo method, create a macro that's assigned to a key for turning screen painting back on. To create this macro, follow these steps:

1. Go to the Macro page in the database window.
2. Create a new macro.
3. In the Action column, select Echo.
4. On the View menu, if Macro Names doesn't have a check mark next to it, select it. An additional column with the name of the macro appears in the macro window.
5. The Macro Name column specifies which key combination causes the macro to run. You can assign this macro to the F4 key by typing **{F4}**.
6. From the File menu, save this macro as AutoKeys. The macro should look like the macro in Figure 4.2.
7. Close the Macro window. Now, whenever you press F4, Echo is automatically set to true.

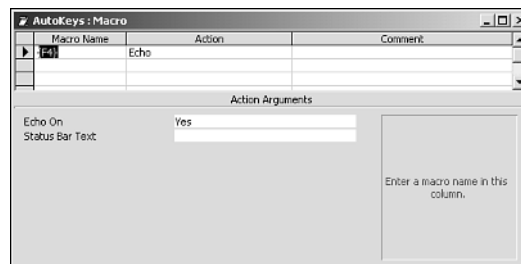


FIGURE 4.2

Using AutoKeys is one way to create your own set of function-key commands.

If you don't want to create this macro with the preceding steps, you can still take advantage of the macro by importing it into your database. The AutoKeys macro is in the Chap04.mdb database, in the \Examples\Chap04 folder on this book's Web page at www.sampublishing.com. The AutoKeys macro name is recognized by Access as a special type of macro. Whenever a key is pressed, Access checks this macro and executes your custom actions.

Setting and Retrieving Option Values

By using the `GetOption` and `SetOption` methods, you can modify most options available in the Access 2002 Options and Startup dialog.

NOTE

Any changes you make to the options are persistent with Access. If you want the options to take effect only for your application, store the original option values in variables and restore them when you quit your application. If you don't restore the user's default settings, all other applications will use the customized settings if they don't specify their own. This changes the user's environment outside your application and might confuse the user.

To see how to switch option values, look at the code in Listing 4.3, which retrieves the current value of the `ShowAnimations` property and asks users whether they want to change this value.

LISTING 4.3 Chap04.mdb: Modifying Options Programmatically

```
Sub ToggleAnimations()  
    Dim varShowAnimations As Variant  
    varShowAnimations = Application.GetOption("Show Animations")  
  
    If MsgBox("Showing Animations is currently " & IIf(varShowAnimations, _  
        "On", "Off") & ". Do you want to switch this setting?", _  
        vbYesNo, "Option Example") = vbYes Then  
        '-- Toggle the option  
        Application.SetOption "Show Animations", Not varShowAnimations  
    End If  
End Sub
```

Listing 4.3 specifies the option name as a string. You can obtain the option name by looking at the Option dialog (from the Tools menu choose Options). Figure 4.3 shows the Datasheet page, with the Show Animations check box.

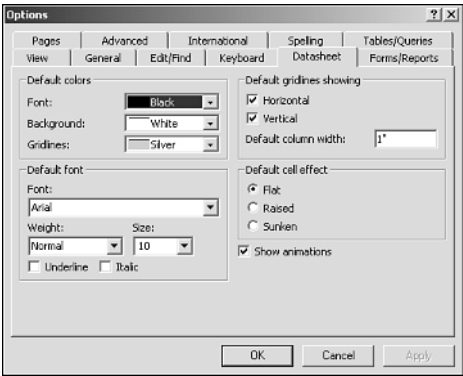


FIGURE 4.3
All options can be set through code as well as through the user interface.

Whenever you retrieve an option value into a variable, use a variant data type. The `GetOption` method always returns the option’s setting as a variant. Using a variant doesn’t prevent you from using Boolean or other operations on the variable. In Listing 4.3, the `Not` operator was used on the current setting of the option to cause it to switch by using the `SetOption` method.

Quitting Your Application

The `Quit` method is a quick and simple way to exit your application (equivalent to choosing Exit from the File menu). `Quit` takes an optional parameter that specifies how to handle any unsaved objects. The following values are available for the optional parameter:

Argument	Description
acPrompt	Prompts whether to save each unsaved object
acSaveYes	Automatically saves all objects without prompting the user (the default when no option is specified)
acExit	Causes any unsaved objects to be discarded without warning

Manipulating the Active Object

Through the `Screen` object on the `Application` object, you can determine the currently active form, report, data access page, or control, as well as the control that last received focus.

Although there are other properties in addition to those listed here, these properties return their respective objects:

<i>Property</i>	<i>Object Returned</i>
ActiveForm	Form
ActiveDataAccessPage	Data access page
ActiveReport	Report
ActiveControl	Control
PreviousControl	Control

To assign these properties to a variable, you must use the Set statement. For example,

```
Dim frmCurrent as Form
Set frmCurrent = Screen.ActiveForm
```

NOTE

The reference to the Application object is omitted when accessing the Screen object. When writing code within Access, you don't need to reference the Application object because it's implicitly assumed. When controlling Access from another application, the Application object must be explicitly referenced.

You can use the PreviousControl property to write code that causes a control to reject the focus. For example, when you click a command button, the button is clicked but is never selected. This can be easily done by putting the code in Listing 4.4 on a button's OnClick event. (This code can be found in the frmUsingPreviousControl form.)

LISTING 4.4 Chap04.mdb: Using the PreviousControl Property

```
Private Sub cmdButton_Click()
    Dim objParent As Object
    ' Error checking to validate there is a previous control to go back to.
    On Error GoTo err_Exit
    Set objParent = Screen.PreviousControl

    ' Find the form that the previous control resides on.
    Do Until TypeOf objParent Is Form
        Set objParent = objParent.Parent
    Loop
```


LISTING 4.4 Continued

```
' Activate the form
objParent.SetFocus
' Activate the control
Screen.PreviousControl.SetFocus

err_Exit:

End Sub
```

Using Your Application's hWndAccessApp

The hWnd (Window handle) allows you to call Windows APIs. Calling Windows APIs can be very dangerous, with unpredictable results, if not done properly. Using the hWnd is discussed in Chapter 15, “Extending the Power of Access with API Calls.”

Controlling Access Through Automation

Access can act as an Automation server as well as an Automation client. It can control other applications that support Automation and also can be controlled by other applications. For example, you can write an application in Excel that manipulates Access through the Access programming model by using the Application object. Chapter 13, “Driving Office Applications with Automation,” explains how to use Automation to control Access from other applications, as well as how to use Access to control other applications.

Commanding the DoCmd Object

The DoCmd object, which provides access to all actions in Access, consists only of methods. These methods provide easy ways to perform complex but often-used tasks. For example, DoCmd has methods for moving records, copying and deleting objects, opening and closing reports and forms, and printing and exporting queries. For more information about the DoCmd object, see Chapter 1, “Macros Are for Weenies; Code Is Cool!”

CurrentData and CurrentProject Objects

The CurrentData and CurrentProject objects contain several properties and collections that you might find useful in your applications.

Before Access 2000, you had to use a combination of the DAO Containers collection and the Access Forms and Reports collections to obtain information on database objects. Now, collections off the CurrentData and CurrentProject objects support ADPs and simplify the database object programming model.

CurrentData has the following object collections:

<i>Object Collection</i>	<i>Description</i>
AllTables	Collection of all tables in the database; applies to ADPs and MDBs
AllQueries	Collection of all queries in the database; applies only to MDBs
AllViews	Collection of all views in the database; applies only to ADPs
AllStoredProcedures	Collection of all stored procedures in the database; applies only to ADPs
AllDatabaseDiagrams	Collection of all database diagrams in the database; applies only to ADPs
AllFunctions	Collection of all Access objects in CurrentData or CodeData

CurrentProject contains the following object collections, which all apply to ADPs and MDBs:

<i>Object Collection</i>	<i>Description</i>
AllForms	Collection of all forms in the database
AllReports	Collection of all reports in the database
AllMacros	Collection of all macros in the database
AllDataAccessPages	Collection of all Data Access Pages in the database
AllModules	Collection of all code modules in the database

These collections contain objects of type `AccessObject`, which has the following read-only properties:

<i>Property</i>	<i>Description</i>
Name	The object's name as it appears in the database window.
FullName	Full path for a data access page link.
IsLoaded	Indicates whether the object is currently open in the Access user interface.
Type	Returns a number from the <code>acObjectType</code> enum.
Properties	A collection of application-defined properties for the object. This applies only to the collections belonging to CurrentProject—namely, AllForms, AllReports, AllMacros, AllDataAccessPages, and AllModules. (More on this later.)

The function in Listing 4.5 uses these collections to list all currently open and saved objects in the VBE Immediate window.

LISTING 4.5 Chap04.mdb: Accessing All Objects in an MDB or ADP by Using CurrentProject

```
Sub ListOpenObjects()
    Dim objAcObj As AccessObject
    Dim objCurProj As CurrentProject
    Dim objCurData As CurrentData
    Set objCurProj = Application.CurrentProject
    Set objCurData = Application.CurrentData

    '-- Tables
    Debug.Print "Tables:"
    For Each objAcObj In objCurData.AllTables
        If objAcObj.IsLoaded Then
            Debug.Print " " & objAcObj.Name
        End If
    Next objAcObj

    '-- Some of the interesting collections in CurrentData depend _
    '-- on project type (MDB or ADP)
    If objCurProj.ProjectType = acMDB Then
        '-- Queries
        Debug.Print "Queries:"
        For Each objAcObj In objCurData.AllQueries
            If objAcObj.IsLoaded Then
                Debug.Print " " & objAcObj.Name
            End If
        Next objAcObj
    Else '-- acADP
        '-- Stored Procedures
        Debug.Print "Stored Procedures:"
        For Each objAcObj In objCurData.AllStoredProcedures
            If objAcObj.IsLoaded Then
                Debug.Print " " & objAcObj.Name
            End If
        Next objAcObj
        '-- Views
        Debug.Print "Views:"
        For Each objAcObj In objCurData.AllViews
            If objAcObj.IsLoaded Then
                Debug.Print " " & objAcObj.Name
            End If
        Next objAcObj
    End If
End Sub
```

LISTING 4.5 Continued

```

Next objAcObj
'-- Database Diagrams
Debug.Print "Database Diagrams:"
For Each objAcObj In objCurData.AllDatabaseDiagrams
    If objAcObj.IsLoaded Then
        Debug.Print " " & objAcObj.Name
    End If
Next objAcObj
End If

'-- Forms
Debug.Print "Forms:"
For Each objAcObj In objCurProj.AllForms
    If objAcObj.IsLoaded Then
        Debug.Print " " & objAcObj.Name
    End If
Next objAcObj
'-- Reports
Debug.Print "Reports:"
For Each objAcObj In objCurProj.AllReports
    If objAcObj.IsLoaded Then
        Debug.Print " " & objAcObj.Name
    End If
Next objAcObj
'-- Data Access Pages
Debug.Print "Data Access Pages:"
For Each objAcObj In objCurProj.AllDataAccessPages
    If objAcObj.IsLoaded Then
        Debug.Print " " & objAcObj.Name
        '-- Also print out the full path to the .HTM file
        Debug.Print " " & objAcObj.FullName
    End If
Next objAcObj
'-- Macros
Debug.Print "Macros:"
For Each objAcObj In objCurProj.AllMacros
    If objAcObj.IsLoaded Then
        Debug.Print " " & objAcObj.Name
    End If
Next objAcObj
'-- Modules
Debug.Print "Modules:"
For Each objAcObj In objCurProj.AllModules

```

LISTING 4.5 Continued

```
If objAcObj.IsLoaded Then
    Debug.Print " " & objAcObj.Name
End If
Next objAcObj

End Sub
```

Working with CurrentProject Properties

How many times have you needed your application's name and path while coding?

CurrentProject has several useful properties, many of them dealing with connections, that primarily interest developers using ADO. All developers will enjoy the following:

<i>Property</i>	<i>Description</i>
FullName	Full path to your database, including the filename.
Name	Name of your database with no path information.
Path	Path to your database without the filename.
ProjectType	Indicates whether the current project is an ADP or MDB. This should be of primary interest to wizard, add-in, and library developers. It can take on the values acADP, acMDB, or acNull.
Properties	A collection of application-defined properties for the database. See the next section for more information.

NOTE

The Access Application object also supports CodeData and CodeProject, objects which refer to the database currently running code (not necessarily the "current database"). Their relationship to CurrentData and CurrentProject is analogous to the relationship between CurrentDb and CodeDb in Access 95 and 97. These should be of primary interest to developers building add-ins and libraries.

Adding Custom Properties to Access Objects

As stated in the previous sections, the *AllObjects* collections and CurrentProject itself support user-defined properties via the AccessObjectProperties collections. The ability to easily add custom properties to all project objects is a great feature.

NOTE

The ability to easily add custom properties applies only to the collections belonging to `CurrentProject`—namely, `AllForms`, `AllReports`, `AllMacros`, `AllDataAccessPages`, and `AllModules`. Unfortunately, it does *not* apply to `CurrentData`'s collections (`AllTables`, `AllQueries`, `AllViews`, `AllStoredProcedures`, and `AllDatabaseDiagrams`).

For an example of how to take advantage of adding custom properties, do the following:

1. Open the VBE from the Tools menu by choosing the macro and then Visual Basic Editor.
2. Open the VBE Immediate window from the View menu or by pressing Ctrl+G.
3. Execute the following statement in the Immediate window:

```
CurrentProject.Properties.Add "MyCustomProperty", _  
    "MyCustomPropertyValue"
```

You just added a custom property to your database. To verify that it worked, close and reopen your database, and then execute the following statement in the Immediate window:

```
? CurrentProject.Properties("MyCustomProperty")
```

It prints `MyCustomPropertyValue`. To remove this property, simply execute

```
CurrentProject.Properties.Remove "MyCustomProperty"
```

It's a good idea to use naming conventions for your Access objects, such as the widely used Leszynski Naming Conventions used in this book. However, you generally don't want to use these names in your application's interface because they aren't end-user friendly. Custom properties can help you solve this dilemma, as shown in Listing 4.6.

LISTING 4.6 Chap04.mdb: Showing the Friendly Names of Objects

```
Sub SetFriendlyName(strUnfriendlyName As String, _  
    strFriendlyName As String, lObjType As AcObjectType)  
  
    Dim objProps As AccessObjectProperties  
    On Error GoTo SetFriendlyName_Err  
  
    '-- Set a reference to the right collection  
    Select Case lObjType  
        Case acForm  
            Set objProps = CurrentProject.AllForms(strUnfriendlyName).Properties  
        Case acReport  
            Set objProps = CurrentProject.AllReports(strUnfriendlyName).Properties
```

LISTING 4.6 Continued

```

    Case acMacro
        Set objProps = CurrentProject.AllMacros(strUnfriendlyName).Properties
    Case acModule
        Set objProps = CurrentProject.AllModules(strUnfriendlyName).Properties
    Case acDataAccessPage
        Set objProps = _
            CurrentProject.AllDataAccessPages(strUnfriendlyName).Properties
    Case Else
        MsgBox "Unsupported object type"
        Exit Sub
    End Select

    '-- Set the property. If it already exists, this will simply
    '-- overwrite the current value.
    objProps.Add "FriendlyName", strFriendlyName

    Exit Sub

SetFriendlyName_Err:
    If Err = 2467 Then
        '-- Object doesn't exist
        MsgBox "Object " & strUnfriendlyName & " doesn't exist."
    Else
        MsgBox "Unexpected error: " & Err.Description
    End If

End Sub

```

Now every time you create an object whose name you want to expose in the user interface, run the SetFriendlyName routine from the Immediate window to add a FriendlyName property:

SetFriendlyName "frmProducts", "Products Form", acForm

When you want to use the friendly name, all you have to do is obtain the property, as in Listing 4.7, which returns a semicolon-delimited list of forms. You can use this function to fill a list box that allows users to select one of your forms to print or open.

LISTING 4.7 Chap04.mdb: Adding a Custom New Property to an Object

```

Function strGetListOfForms() As String
    Dim objFrm As AccessObject
    For Each objFrm In CurrentProject.AllForms
        On Error Resume Next
        strGetListOfForms = strGetListOfForms & _
            objFrm.Properties("FriendlyName") & ";"
    Next objFrm

```

LISTING 4.7 Continued

```
If Err = 2455 Then
    '// Form didn't have a FriendlyName set, use the regular name
    strGetListOfForms = strGetListOfForms & objFrm.Name & ";"
End If
Next objFrm
End Function
```

These collections simplify writing code that uses information related to database objects. The object properties collections open up new possibilities for developers interested in extending database objects.

Looking at the References Collection

Access gives you the capability to examine which references are set for the current application.

New as of Access 2002 is the capability to look at the `BrokenReference` property off the `Application` object. If it's set to `True`, a reference is broken in the References collection, and you can then list them with the code in Listing 4.8.

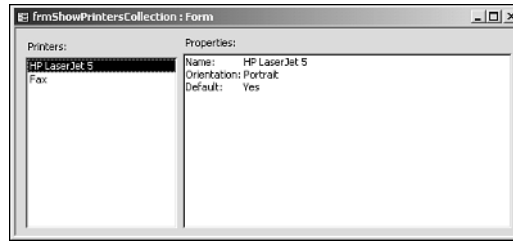
LISTING 4.8 Chap04.mdb: Displaying an Application's References

```
Sub DisplayReferences()
    Dim intCurrRef As Integer
    If Application.BrokenReference = True Then
        For intCurrRef = 1 To References.Count
            Debug.Print "    Name: " & References(intCurrRef).Name
            Debug.Print "Full Path: " & References(intCurrRef).FullPath
            Debug.Print "Broken: " & References(intCurrRef).IsBroken
        Next intCurrRef
    End If
End Sub
```

Specifying and Manipulating Printers

You now can specify and manipulate printer properties with the new `Printers` collection and `Printer` object available in Access 2002 within the `Application` object. This feature has been available in Visual Basic for a few versions.

The `Printers` collection contains the printers currently installed on your system. Each individual `Printer` object is a representation of printers available. Besides being contained in the `Printers` collection, the application, form, and report objects each have a `Printer` property, which points to a printer object. The form in Figure 4.4 takes advantage of the `Printers` collection and `Printer` object off the `Application` object.

**FIGURE 4.4**

With the Printers collection and Printer object, you now have almost total control over how your reports print.

This form, `frmShowPrintersCollection`, contains two controls: `lstPrinters`, a list box, and `txtProperties`, a text box. In the form's `Open` event, the `lstPrinters` list box is populated from the `Printers` collection (see Listing 4.9).

LISTING 4.9 Chap04.mdb: Loading the Available Printers into a List Box

```
Private Sub Form_Open(Cancel As Integer)
    Dim prnCurr As Printer
    For Each prnCurr In Application.Printers
        lstPrinters.AddItem prnCurr.DeviceName
    Next prnCurr
End Sub
```

That's right, you now have an `AddItem` method on list boxes and combo boxes.

After loading the list box with the `DeviceNames` of each printer, the next work comes when a printer is selected. Then the text box is loaded with the displayed properties (see Listing 4.10).

LISTING 4.10 Chap04.mdb: Displaying Properties of the Chosen Printer

```
Private Sub lstPrinters_AfterUpdate()
    Dim strTemp As String
    Dim prnCurr As Printer
    Set prnCurr = Application.Printers(lstPrinters.ListIndex)

    '-- First, the name of the printer (or device)
    strTemp = "Name: " & prnCurr.DeviceName
    '-- Display the orientation
    strTemp = strTemp & vbCrLf & "Orientation: "

    If prnCurr.Orientation = acPRORLandscape Then
        strTemp = strTemp & "Landscape"
```

LISTING 4.10 Continued

```
Else
    strTemp = strTemp & "Portrait"
End If

'-- Next, see if the name of the chosen printer matches
' the applications printer.
strTemp = strTemp & vbCrLf & "Default:      "
If prnCurr.DeviceName = Application.Printer.DeviceName Then
    strTemp = strTemp & "Yes"
Else
    strTemp = strTemp & "No"
End If

'-- Now assign it to the text box.
Me.txtProperties = strTemp

End Sub
```

For additional properties, look up the `Printer` object in the Object Browser. Hopefully, you can see some of the possibilities for more advanced uses for the `Printers` collection and `Printer` object.

Working with the Forms, Reports, and Data Access Pages Collections

This area of the Access object model shouldn't be foreign to you. In Chapter 2, you wrote code that referenced controls on a form. Whenever a form is open, the form is contained in the `Forms` collection. The `Reports` collection works fundamentally the same way as the `Forms` collection.

The `Forms`, `Reports`, and `Data Access Pages` collections consist of all open forms, reports, and Data Access Pages, allowing you to determine which objects are open. These collections don't provide a way to list all the objects of these types in your application, however. For that, look at the collections under the `CurrentProject` object discussed earlier in the section "CurrentData and CurrentProject Objects."

Each form and report object has a `Controls` collection (discussed in Chapter 2). Here is where the similarities end with the `Data Access Page` object. They do not have control collections.

NOTE

In the following sections, although only the Form object is referred to, all these techniques also apply to the Report object.

Using the Me Keyword

When writing code within a form's module, use the `Me` keyword to reference the form and its contents. `Me` returns a reference to the object the code lives behind. Therefore, accessing the caption of the form when writing code behind the form can be done as follows:

```
Me.Caption = "My Form"
```

All code written behind the form should reference that form with the `Me` keyword. You can also optionally omit the `Me` keyword because it's implicitly inferred when writing code behind a form. Never reference the form in the form's code through the Forms collection. Doing so makes it difficult to evolve your application to support multiple-form instances. (Multiple-form instances are discussed later in the section "Supporting Multiple-Form Instances.")

Explicitly Referencing a Form

If you're writing code in a code module outside the form, you need to explicitly reference the form through the Forms collection. You've already seen this done in Chapter 1. For example, to set the form's caption, you can write the following code:

```
Forms!Form1.Caption = "My Form"
```

Because you're referencing the form through the Forms collection, you must make sure that the form is loaded. If the form isn't loaded, a runtime error is generated. Therefore, you should always write an error handler to open the form if it's not open now. This could be done as shown in Listing 4.11.

LISTING 4.11 Chap04.mdb: Setting Up Proper Error Handling for Opening Forms

```
Private Sub cmdMyButton_Click()  
    On Error GoTo Err_cmdMyButton_Click  
    Forms!frmExample.Caption = "MyForm"  
  
Exit_cmdMyButton_Click:  
    Exit Sub  
  
Err_cmdMyButton_Click:  
    If Err = 2450 Then      ' Form not open
```

LISTING 4.11 Continued

```
DoCmd.OpenForm "frmExample"  
Resume  
Else  
    MsgBox "Error: " & Err.Description  
    Resume Exit_cmdMyButton_Click  
End If  
  
End Sub
```

NOTE

A loaded form doesn't necessarily need to be visible. Forms can be loaded but still be hidden or invisible in Access.

Another useful function is to check whether the form is open before operating on it. The following code looks in the Forms collection and checks whether a form is open with the specified name:

```
Function ap_FormIsOpen(strFormName As String) As Boolean  
    ap_FormIsOpen = _  
        Application.CurrentProject.AllForms(strFormName).IsLoaded  
End Function
```

Accessing forms in this manner works fine as long as you aren't creating multiple copies of a form to be displayed at the same time. Creating multiple forms is discussed in more detail later in the section "Programming Multiple Copies of the Same Form." The following section, meanwhile, shows a safer method for accessing forms—one that doesn't depend on a form being open.

Referencing the Form's Default Instance

Access provides a method for setting properties on the form's default class. The default class refers to the definition of the form as saved in your database. That means you can set properties on the form and report without actually opening the form. These property settings aren't saved with the form and have a limited life.

To change the caption of the default instance, refer to the form by using `Form_FormName`. Therefore, to set the caption of `Form1`, use

```
Form_Form1.Caption = "My Form"
```

Now, if the form is already displayed or you subsequently display the form, the new caption is reflected. But as soon as the form is closed, the new caption and any other properties set this way are forgotten.

Setting properties on the default class is very useful. You no longer need to ensure that the form is open before assigning values to the property.

NOTE

If the form is *lightweight* (meaning that it has no code behind it), this method won't work. Lightweight forms don't include modules and, therefore, don't perform all the tasks of those that do. Primarily, they leave out the tasks that you would create with code behind the form. Of course, because these forms don't have modules, they don't incur the overhead of VBA and therefore can generally perform faster.

Forms used in multiple-instance references must have modules. You also might have to explicitly set the `HasModule` property of the form to `true` to specify that the form does in fact have a class module. For more information about class modules, see Chapter 2.

Programming Multiple Copies of the Same Form

One of Access's most powerful features is its capability to display multiple instances of forms. An instance refers to a single copy that's open. You can display the same form onscreen multiple times. In this example, you explore how to reuse the `frmEmployee` form to display employees and their supervisors (see Figure 4.5).

Programming multiple instances of forms requires you to understand a few new programming concepts. Imagine that you open two `frmCustomer` forms. From each `frmCustomer` form, you open an order-entry form. On the `frmOrder` form, you create a button that returns you to the `frmCustomer` form. Usually when developing in Access, you would simply use

```
DoCmd.OpenForm "frmCustomers"
```

When you create an application that supports multiple instances, however, you can have multiple customer forms open. In that case, what instance of the form does that call go to? Therein lies the root of all the difficulties you'll experience. In the following sections, you learn to manage multiple-form instances—first with a single variable, and then with custom Forms collections.

FIGURE 4.5

With multiple instances, you can use the same form for various purposes, all open at once.

NOTE

This section is meant as a introduction to managing multiple instances of forms with collections. For a more advanced example, see Chapter 16.

Supporting Multiple-Form Instances

Earlier, the section “Defining a New Collection” introduced you to the New keyword, which is used to create a new instance of an object. Because a form is just another object, the New keyword is also used to create new instances of forms.

By declaring new instances of a form, you can create, manipulate, and display a single form multiple times. Each instance of the form can be worked on individually. This means that any changes made to an instance of the form affect only that instance.

An example of multiple-form instances is available in the Chap04.mdb database in the \Examples\Chap04 folder on the book’s Web page at www.sampublishing.com. This example uses the frmEmployees form:

1. Open the frmEmployees form from the Forms tab in the Database window.
2. Locate an employee who has a supervisor.
3. Click the ... button that follows the supervisor’s name. A new frmEmployees form appears with the supervisor’s information, and the window’s caption changes. (This form is actually just another instance of the frmEmployees form.)

4. If this supervisor has a supervisor, click the ... button again to view his supervisor in another window. If no supervisor is present, typing in a new supervisor opens a new frmEmployees form to enter the new supervisor's information.
5. Close the original frmEmployees form (identified by the caption Employees; all supervisors have a new caption). All the frmEmployees forms automatically close.

Examining the frmEmployees Form's Code

To examine the code behind the form, open the frmEmployees form in Design mode and then choose Code from the View menu. As you examine the code behind the frmEmployees form, notice that all the references to the frmEmployees form itself are done with the Me keyword. Using the Me keyword is very important when using form instancing. If you were to write your code to reference the form directly through the form's collection, your code would fail to run properly when displaying instances.

Here's the declarations section of the code behind the frmEmployees form:

```
Option Compare Database
Option Explicit
Private frmChild As Form_frmEmployees
```

The frmChild variable creates the next instance of the frmEmployees form. Listing 4.12 shows the code related to multiple instances behind the frmEmployees form. Whenever the ... button (btnViewSuper) is clicked, a new instance of the frmEmployees form is created and stored in the frmChild variable; the caption is set; the appropriate employee is located; and the form is displayed.

LISTING 4.12 Chap04.mdb: Opening Multiple Instances of Forms

```
Private Sub btnViewSuper_Click()
    On Error GoTo Err_btnViewSuper_Click

    If Not IsNull(Me!Supervisor) Then
        Set frmChild = New Form_frmEmployees
        With frmChild
            .Filter = "EmployeeID = " & Me!Supervisor
            .FilterOn = True
            .Caption = "Supervisor for " & Trim(Me!LastName) & _
                ", " & Trim(Me!FirstName)
            .Visible = True
        End With
    End If

Exit_btnViewSuper_Click:
Exit Sub
```

LISTING 4.12 Continued

```
Err_btnViewSuper_Click:
    MsgBox Err.Description
    Resume Exit_btnViewSuper_Click

End Sub

Private Sub Form_Unload(Cancel As Integer)
    Set frmChild = Nothing
End Sub
```

All references to the new instance of the form should be done by using the `frmChild` variable. The form is never referenced by using the Forms collection. Using the Forms collection would require special coding techniques so that you can identify between the two employee forms in the collection. If you want to do this, you can use the `Tag` property or your own custom property to identify the form.

When you assign a new instance of the form to a variable, the new form is loaded into memory. This causes the form's `OnLoad` event to be fired. However, the form isn't yet visible. To make a new instance of a form visible, you must set the form's `Visible` property to `true`. Listing 4.7 earlier in this chapter does this by assigning `true` to the `Visible` property.

When the new instance of the `frmEmployees` form is created, it's added to the Forms collection. For example, if you opened the form for the employee and his supervisor, the Forms collection would contain two references to the `frmEmployees` form—one reference to each instance.

Closing the `frmEmployees` Forms Automatically

When you ran the preceding example, closing the main `frmEmployees` form closed all the other forms. If you examine the code, notice that only a single line of code was written to close the forms. This was done by setting the `frmChild` variable to nothing. How did this close the forms?

Form instances stay loaded for as long as the variable referencing the form stays in scope. When the variable goes out of scope, so does the form. The variable can go out of scope by setting the variable to nothing, or when the form containing the variable is closed.

This actually works much like the way Access works. When you're running Access, the Database window is within the scope of Access. All forms, table views, and other windows are scoped by the database. Therefore, when you close your database, all windows related to the database also close. When you close Access, all windows within Access close.

Therefore, to write code where the instances can live independent of the scope of the form creating them, you must use a public variable in a code module. The best method for handling this is to create or use a public collection. For example, to rewrite the employee example so that the supervisor forms don't automatically disappear, follow these steps:

1. Create a new code module and type the following code line to create a public collection:

```
Public colFormList as New Collection
```

2. Save the code module as modWindowManager.
3. Open the frmEmployees form in Design view and open the code editor.
4. Remove the declaration `Private frmChild as Form_frmEmployees`.
5. Add the declaration **Dim frmChild as Form_frmEmployees** to the btnViewSuper_Click subroutine. Immediately following the `frmChild.Visible` line, add the code **colFormList.Add frmChild**.

Now when you run the frmEmployees form and open supervisor windows, the form is added to colFormList. Closing a form no longer closes any of the other frmEmployees forms because you're storing the reference to the forms in a variable outside the form's scope. The scope of public variables in modules is for the life of the application. Therefore, now the forms won't close until the user or code explicitly closes them, or the user quits the application. You then have to clean up memory by setting the variables to Nothing; otherwise, memory will continue to be used.

Summary

Understanding the collections used in Access is part of the key to understanding Access development. By understanding the structures, methods, and properties of the collections and objects, you can accomplish pretty much anything necessary having to do with programming the user interface.

The collections mentioned in this chapter are used throughout the rest of the book, so you see plenty of examples that put them into practice. The next logical steps are the following chapters:

- Chapter 7, "Handling Your Errors in Access with VBA," shows you the basics of error handling, something that needs to be used in most of your code.
- Chapter 16, "Extending Your VBA Library Power with Class Modules and Collections," shows you additional examples of collections and working with multiple instances of forms.

Introducing ActiveX Data Objects

CHAPTER

5

IN THIS CHAPTER

- Looking at ADO's Object Models 110
- Referencing the Type Libraries 113
- Opening a Connection to a Database 115
- Creating a Recordset 116
- Working with Queries 123
- Working with Tables 127

ActiveX Data Objects (ADO) is Microsoft's standard data-access method for most Office, Web, and Visual Basic environments. Products that ship ADO in the box include Microsoft Office 200x, Visual Studio 6.0 (including all languages in the box), and SQL Server 7.0/2000.

NOTE

The next version of ADO under development is ADO.NET, which will be included in Visual Studio.NET, the next version of the Microsoft tools. At the time of this writing, the .NET tools were in beta testing.

Data Access Objects (DAO), the previous standard, was used throughout Office and Visual Basic but wasn't designed to be used with Web environments such as Visual InterDev and other data servers. There are no plans to update DAO beyond DAO 3.6, which includes support for Unicode and some bug fixes but doesn't have any other new features beyond version 3.5.

NOTE

Although DAO won't be revised anymore, Jet, which is Access's native database engine, is alive and well. In fact, Microsoft has taken great pains to try to create parity between what developers had for managing databases with code in DAO and Jet, and that which is available in ADO and Jet. A number of new features can be found for Jet in ADO. If you can't find a command that you need in ADO, check out Access's Application object, which has been enhanced in Access 2002 to incorporate a number of methods performed in DAO.

Looking at ADO's Object Models

That's right, *object models*, plural. Unlike DAO, which consists of one object model, ADO has three separate object models, which work together to give you the objects and collections necessary to work with your data. The three latest object models consist of

- *ActiveX Data Objects 2.5 (ADODB)* let you create and work with recordsets, as well as perform error handling.
- *ADO Extensions 2.5 for DDL and Security (ADOX)* is the data definition language, allowing you to work with and modify the database schema. This object model also includes security objects.
- The *Jet and Replication Objects 2.5 (JRO)* model enables you to work with the Jet engine and replication.

Although these separate object models will be explained as such, you will also use them cohesively. For instance, to modify the table's structure, you need to get to the `Tables` collection located off the `Catalog` object in the ADOX library; however, this `Catalog` will have its `ActiveConnection` property set to the `Connection` object (ADODB). More on this is discussed later in the section "Working with Tables."

The ActiveX Data Objects 2.5 (ADODB) Object Model

When you use ADO or recordsets, most of the work is done in the ADODB module. This object model (see Figure 6.1) consists of the following:

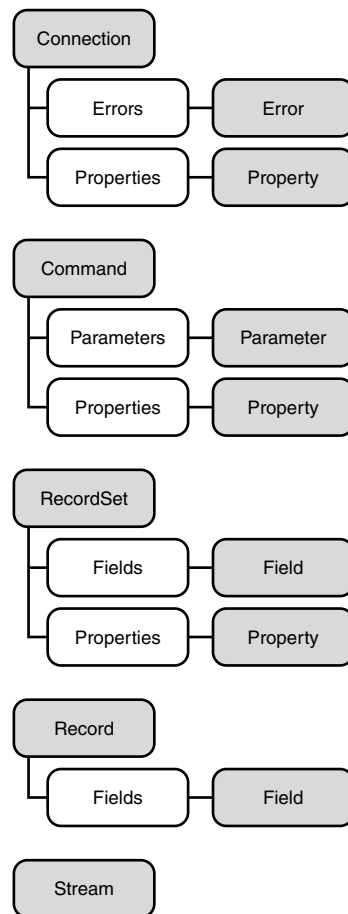
- The `Connection` object, equivalent to the `Database` object in DAO, is where most of your work with ADO begins. The objects and collections mentioned after this all come from the `Connection` object.
- The `Errors` collection/`Error` object, identical to the DAO `Errors` collection and `Error` object, allows developers to manage error handling.
- The `Command` object allows you to run a query against a database and return records in a `Recordset` object, to manipulate a database's structure, or to execute a bulk operation. A collection of parameters is used with the `Command` object.
- Like with the DAO `Recordset` object, you can open ADO `Recordset` objects as read-only or dynamic. Each ADO `Recordset` object also has a `Fields` collection.
- The `Stream` object allows you to read in special tree-structured hierarchies such as e-mail messages or file systems. You can even point the object to an URL, provided the system has set it up consistently. This is another way ADO allows you to read outside data such as from other applications or over the Internet, whereas you couldn't do this with DAO easily.

The ADO Extensions 2.5 for DDL and Security (ADOX) Object Model

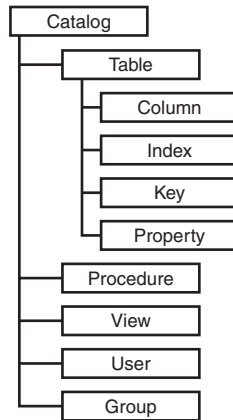
Whenever you create new objects in the database, you use the ADOX object model for saving the objects and maintaining security information. This object model (see Figure 5.2) consists of the following:

- The `Catalog` object is the container for the `Tables`, `Procedures`, `Views`, `Users`, and `Groups` listed here.
- As with the DAO `TableDefs` collection, the ADOX `Tables` collection and `Table` object are the structures of the existing tables in the database. The `Table` object consists of columns, keys, indexes, and properties. Row-returning, non-parameterized queries can be found here.

- The Procedures collection and Procedure object are stored procedures.
- The Views collection and View object are virtual tables that consist of tables and other views. You can create views by using the Command object, found in the ADODB object model.
- The Users collection and User object, like in DAO, are users in the workgroup.
- The Groups collection Group object, like in DAO, are groups within the workgroup.

**FIGURE 5.1**

The object model for ActiveX Data Objects 2.5.

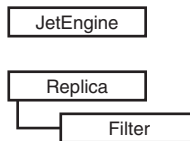
**FIGURE 5.2**

The object model for ADO Extensions 2.5 for DDL and Security.

Jet and Replication Objects 2.5 (JRO) Object Model

The JRO object model (see Figure 5.3) consists of the following:

- Analogous to dbEngine in DAO, the JetEngine object allows you to use the CompactDatabase and RefreshCache methods. For more information on the CompactDatabase method, see Chapter 25, “Startup Checking System Routines Using ADO.”
- Within the Replicas collection and Replica object are all the objects and commands necessary to perform replication programmatically. To see these objects used, check out Chapter 22, “Welcome to the World of Database Replication.”

**FIGURE 5.3**

The object model for Jet and Replication Objects 2.5.

Referencing the Type Libraries

Before seeing how to use these collections and objects, you need to know how to reference the object model libraries for use. The object models are found in type libraries in the file system.

A *type library* contains the information about the object models such as the collections, properties, and methods. Referencing the type libraries isn't too hard a task, but is still very important.

The first type library, ActiveX Data Objects 2.5, is referenced by default for new databases. The others have to be added as needed. To reference the other two object models, choose References from the VBE's Tools menu. You then have to find the type libraries for the object models and select them. The type library names are as follows:

- Microsoft ActiveX Data Objects 2.5 Library (already referenced)
- Microsoft ADO Ext. 2.5 for DDL and Security Library
- Microsoft Jet and Replication Objects 2.5 Library

You can see these libraries, as well as the Microsoft DAO 3.6 Object Library, in Figure 5.4.

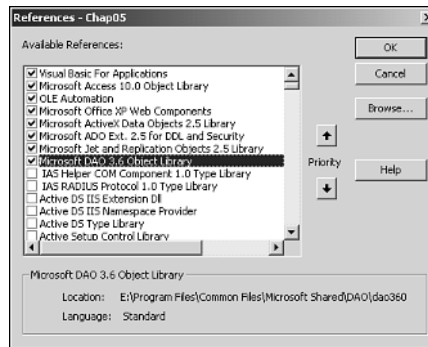


FIGURE 5.4

Registering the object model type libraries.

NOTE

You can see in Figure 5.4 that the DAO library has been moved, using the up and down arrows on the References dialog, to follow the ADO libraries. VBA uses the objects specified in code from the library that occurs first. For example, ADO and DAO both have a Recordset object. When you declare a variable of type Recordset without specifying the library, like this,

```
Dim rstTest as Recordset
```

VBA will figure the Recordset object to be DAO if the DAO library is first.

TIP

You can use both ADO and DAO libraries in the same application without worrying about the order of libraries in the references. This is great when you're converting applications from DAO to ADO. To use all libraries and not worry about the reference order, prefix your objects with the library they come with. Here are some examples, with the specific library types:

```
Dim rstTestDAO as DAO.Recordset
Dim rstTestADO as ADODB.Recordset
Dim catTest as ADOX.Recordset
Dim jeCurr as JRO.JetEngine
```

Although not all objects are shared between the libraries, it's a good idea to specify the library while getting used to the objects.

Opening a Connection to a Database

You first need to open a connection to a database. Creating a reference to a database is pretty straightforward in ADO.

Connecting to the Current Database

Connecting to the current database (or *project*, as is popular in Access) is the easiest. To do so, use the `CurrentProject` object, which is an `Application` object property. The `CurrentProject` object has a `Connection` property, which you assign to a `Connection` object. You can see this in the following code, which prints the `ConnectionString` property to the Immediate window:

```
Sub DisplayLocalConnection()
    Dim cnnLocal As ADODB.Connection
    Set cnnLocal = CurrentProject.Connection
    Debug.Print cnnLocal.ConnectionString
End Sub
```

Although connecting to the current database is pretty easy, connecting to another database takes a little more work.

NOTE

All routines shown in this chapter can be found in `Chap05.mdb` on this book's Web page at www.sampublishing.com.

Connecting to Another Database

The main work in creating a connection to another database comes from having to create the connection string. This code performs this task:

```
Sub DisplayAnotherConnection()  
    Dim cnnNet As New ADODB.Connection  
    cnnNet.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data  
        ➔Source=C:\Books\PwrPrg2002\AppCD\Examples\VideoDat.mdb"  
    Debug.Print cnnNet.ConnectionString  
    cnnNet.Close  
End Sub
```

In this procedure, the connection string passed to the Connection object's Open method consists of the Provider and Data Source. The Provider in this example is Microsoft.Jet.OLEDB.4.0 (Jet).

NOTE

The examples in this chapter use Jet mainly as the service provider. Chapter 24, "Developing SQL Server Projects Using ADPs," shows many examples of working with SQL Server and the connection string.

You can also set the provider ahead of time by setting the Provider property. The following example also shows how to create a connection while specifying the system database and using a user ID and password:

```
Sub DisplayProviderAndSecuredDB()  
    Dim cnnNet As New ADODB.Connection  
    cnnNet.Provider = "Microsoft.Jet.OLEDB.4.0"  
    cnnNet.Properties("Jet OLEDB:System database") = _  
        "C:\Books\PwrPrg2002\AppCD\Examples\Chap05\MySystem.mdw"  
    cnnNet.Open _  
        "Data Source=C:\Books\PwrPrg2002\AppCD\Examples\VideoDat.mdb;  
        ➔UserId=Admin;Password=MyPW"  
    Debug.Print cnnNet.ConnectionString  
End Sub
```

Creating a Recordset

Now it's time to see how to use ADO to perform one of the most common tasks—creating a recordset variable to open a recordset.

Opening a Simple Recordset

To open a recordset, you first create a connection object off the current project, and then specify a new ADO recordset object:

```
Sub OpenRecordsetWithGetStringDisplayExample()
    Dim cnnLocal As New ADODB.Connection
    Dim rstCurr As New ADODB.Recordset
    Set cnnLocal = CurrentProject.Connection
    rstCurr.Open "Select * from tblMovieTitles where ReleaseDate = _
        #02/01/01#", cnnLocal, adOpenStatic, adOpenPessimistic
    Debug.Print rstCurr.GetString
    rstCurr.Close
End Sub
```

Although this code is very similar to DAO, in ADO you use the Recordset object's Open method rather than an OpenRecordset method.

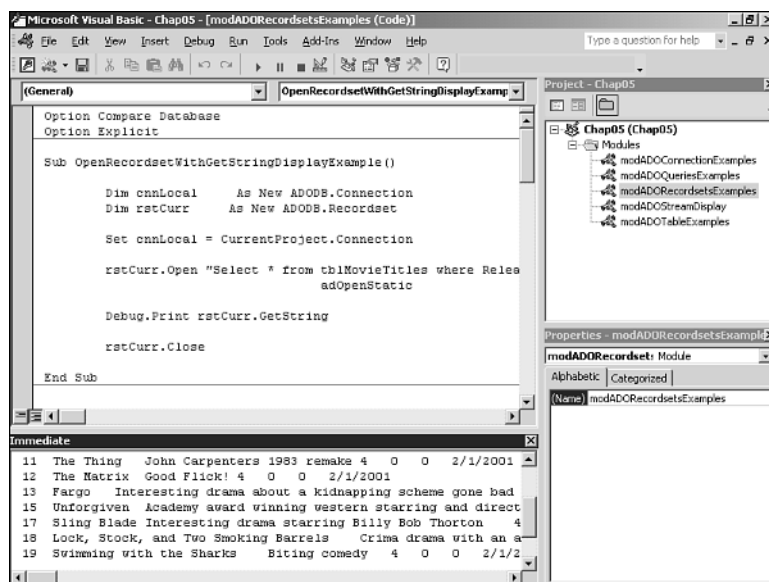
NOTE

The GetString method allows you to put the full recordset into a Variant-type variable. In this case, the results are printed to the Immediate window (see Figure 5.5). In production, this ADO feature is useful usually for serializing the data for network or DCOM transfer. For example, if you tried to do the venerable While Not rs.EOF rs.MoveNext type of loop and pass the data cross-process inside the loop, performance would be hideous. On the other hand, sending a string all at once is relatively fast.

The recordset was opened with an adOpenStatic cursor type and adLockPessimistic locking. (Locking isn't really necessary because the recordset is static, but it allows me to introduce the following tables.) Check out the property choices you have in Tables 5.1 and 5.2.

TABLE 5.1 ADO Recordset Properties Used with Jet

<i>Cursor Type</i>	<i>Description</i>
adOpenKeyset	Recordset is fully updateable, but you don't see others' additions and deletions.
adOpenStatic	Same as a DAO snapshot—recordset is read-only.
adOpenForwardOnly	Same as Static type recordset, except you can only go forward. This gives the best performance, especially if you need to go through the records quickly.

**FIGURE 5.5**

Use the `GetString` method to print a full recordset to the Immediate window.

NOTE

You can create a table-type recordset by combining `adOpenKeyset` with the `adCmdTableDirect` option. This isn't as useful, however, because there's no `Seek` method.

TABLE 5.2 ADO Recordset Locking Options

<i>Locking Method</i>	<i>Description</i>
<code>adLockOptimistic</code>	Use optimistic locking where the recordsets are locked only when the record is saved.
<code>adLockPessimistic</code>	Use pessimistic locking where the recordset is locked when it's first edited.
<code>apLockReadOnly</code>	Lock the entire recordset from other people's use.

TIP

In code, I tend to use pessimistic locking (adLockPessimistic) because that way I make sure that no one will slip under my edits. By using code, the edits and updates are so quick that it doesn't hurt others' performance noticeably.

Looping Through and Editing Recordsets

The example in Listing 5.1 shows a number of features of working with recordsets.

LISTING 5.1 Chap05.mdb: Opening, Looping Through, and Editing a Recordset

```
Sub OpenRecordsetWithEditingExample()  
    Dim cnnLocal As New ADODB.Connection  
    Dim rstCurr As New ADODB.Recordset  
    Dim fldCurr As ADODB.Field  
  
    Set cnnLocal = CurrentProject.Connection  
    rstCurr.Open _  
        "Select * from tblMovieTitles where ReleaseDate = #02/01/01#", _  
        cnnLocal, adOpenKeyset, adLockPessimistic  
    With rstCurr  
        Do Until .EOF  
            '-- Print each of the fields  
            For Each fldCurr In .Fields  
                Debug.Print fldCurr.Value  
            Next  
            '-- Updating the release date; look ma, no Edit or Update!  
            !ReleaseDate.Value = #2/1/2001#  
            Debug.Print  
            .MoveNext  
        Loop  
    End With  
    rstCurr.Close  
End Sub
```

The actual structure of moving through the recordset with `Do Until...Loop` and the use of the `MoveNext` method is the same as in DAO. As with DAO, you can use `MoveLast`, `MoveFirst`, `MoveNext`, and `MovePrevious`.

NOTE

When using an `adForwardOnly` cursor type, you can't use some of the `Move` type methods. This makes sense because you can't `MovePrevious` in a forward-only recordset. Although you can trick ADO by using the recordset's `Cachesize` property, that's not recommended because the `execute` command will be performed again, and you could load memory with the cached records and slow things down worse.

You can see from the line of code that reads

```
!ReleaseDate.Value = #2/1/2001#
```

that you can edit records without using the `Edit` method. You will also update the record by moving off the record—unlike DAO, which causes the information to be lost unless the `Update` method is called. This is convenient, but be careful—you normally still want to call the `Update` method to be consistent.

To cancel updating the record, use the `CancelUpdate` method, off the `Recordset` object.

The `Delete` method works the same as DAO and the `AddNew` method, except that you don't need the `Update` method.

Creating Persistent Recordsets

You might be wondering why anyone would want to create persistent recordsets when he has Make-Table queries and temporary tables. This feature (taking a recordset and creating a file outside Access with it) is handy because if you use an .ADP for your application, you can have temporary tables. No data is stored in the Access Database Projects, only a connection to the back. You would use temporary tables and have them in the front end for these reasons:

- You need to use intermediary data for reports and analysis, where queries just don't do the job.
- You don't have to worry about other users overwriting your data.

The way the issue has been solved is to use persistent recordsets.

To create a persistent recordset, use the `Save` method off the ADO `Recordset` object, passing it the name and path of the file to create. When you want to load the recordset back into memory, use the `Open` method, again passing the filename of the saved recordset. Listing 5.2 shows an example of this.

LISTING 5.2 Chap05.mdb: Saving and Loading a Recordset to/from a File

```

Sub PersistingARecordset()
    Dim cnnNet      As New ADODB.Connection
    Dim rstCurr     As New ADODB.Recordset

    '-- Opening the connection. You will want to change the path
    '-- for your system.
    cnnNet.Provider = "Microsoft.Jet.OLEDB.4.0"
    cnnNet.Open CurrentProject.Path & "\Chap05BE.mdb"
    '-- Open forward only and readonly since we are just saving it to disk.
    rstCurr.Open _
        "Select * from tblMovieTitles where ReleaseDate = #02/01/01#", _
        cnnNet, adOpenStatic

    Debug.Print rstCurr.GetString
    '-- Delete the old copy, if there is one; if you don't,
    '-- you will get an error.
    On Error Resume Next
    Kill CurrentProject.Path & "\TitlesForDate.rst"
    '-- Creating the persistent recordset in the applications directory
    On Error GoTo 0
    rstCurr.Save CurrentProject.Path & "\TitlesForDate.rst", adPersistADTG
    Set cnnNet = Nothing
    Set rstCurr = Nothing
    ' Open the persisted recordset
    rstCurr.Open CurrentProject.Path & "\TitlesForDate.rst", _
        Options:=adCmdFile
    Debug.Print rstCurr.GetString
    rstCurr.Close
End Sub

```

TIP

You can use this feature to help send an update file to another location. That way, the other location can read the update file's records, after assigning them to a recordset variable with the Open method, and process the data as needed.

The Save method also supports XML, which can be used with IE5x. Working with XML and ADO is discussed further in Chapter 6, "Using XML with Access 2002."

NOTE

Although the recordset file has an .rst extension in this example, ADO doesn't care what extension is used.

Using the RecordCount, BOF, and EOF Properties

When attempting to see whether any records in a recordset are using DAO, you could look at the recordset's `RecordCount` property. If the `RecordCount` isn't zero, records were in the recordset. It has been long understood that the `RecordCount` property wasn't going to necessarily be accurate without first performing the `MoveLast` method off the recordset.

However, with ADO you can no longer use `RecordCount` as straightforwardly as DAO. To see whether your recordset has any records in it, use the `BOF` and `EOF` properties. When used as follows on a newly opened recordset, you get `True` or `False` as to whether there are any records:

```
If rstCurr.BOF And rstCurr.EOF Then
```

For more information on using the `RecordCount` property accurately, look it up in Help.

Checking to See What Operations a Recordset Will Support

A number of operations won't be supported by some providers for the data you will work with. The `Supports` method off the `Recordset` object helps with this. This method accepts one argument, which tells the method which feature you're checking to see whether the recordset supports it. The following feature constants can be used:

<code>adAddNew</code>	<code>adHoldRecords</code>	<code>adSeek</code>
<code>adApproxPosition</code>	<code>adIndex</code>	<code>adUpdate</code>
<code>adBookmark</code>	<code>adMovePrevious</code>	<code>adUpdateBatch</code>
<code>adDelete</code>	<code>adResync</code>	

TIP

With the `Supports` method, if `adApproxPosition` or `adBookmark` returns `True`, the `RecordCount` property returns the correct number of records, even without using the `MoveLast` method on the recordset.

Cloning Recordsets

One task that I've occasionally performed in DAO is opening a recordset off another, so that I can move around in the new one without affecting the original. Although you can do this in DAO by using the `OpenRecordset` command off the original recordset, both DAO and ADO have a faster way.

The faster way to create another recordset off the original is to use the `Clone` method. To use this method in ADO, you would use something similar to this:

```
Set rstClone = rstOriginal.Clone
```

You can also specify one parameter that affects the locking method of the new recordset. The choices are `adLockUnspecified` (default) or `adLockReadOnly`.

TIP

One nice thing about using the `Clone` method is that the bookmarks in both recordsets are identical. This means that you can set a bookmark, move around in the other recordset, and then set either recordset's bookmark to the others.

Storing Bookmarks

Bookmarks allow you to mark where you are in a recordset, and then return to that location. In DAO you can store a bookmark to a string, move around in the recordset, and then set the `Bookmark` property to the saved string. You would then be repositioned to the bookmarked record. You can read more of this in Chapter 16, "Extending Your VBA Library Power with Class Modules and Collections."

In ADO, Microsoft gives you more functionality in the bookmarks. By using the `CompareBookmarks` method off a recordset, you can tell which bookmark occurs later in the recordset.

Working with Queries

After working with recordsets, the next important task that will probably be necessary is to be able to manipulate queries from within VBA using ADO. To get started, see how to create a straightforward query and then open a recordset off it.

Creating a New Query

When creating and working with queries, use the ADOX Command object's CommandText property off the Catalog object, as follows. The command is then added to the Procedures collection.

```
Sub CreateASimpleQuery()  
    Dim catCurr      As New ADOX.Catalog  
    Dim cmdCurr      As New ADODB.Command  
    catCurr.ActiveConnection = CurrentProject.Connection  
    cmdCurr.CommandText = "Select * FROM tblMovieTitles"  
    catCurr.Procedures.Append "qryMovieTitles", cmdCurr  
    Set catCurr = Nothing  
End Sub
```

NOTE

As with DAO objects, you can use the Set syntax to set a reference to catCurr.ActiveConnection. I've found that most ADO examples use the method displayed in the example just displayed, as do other languages.

Creating a Parameterized Query

When creating a parameterized query, you add the parameters to the SQL string just as you would with DAO. Check it out in Listing 5.3.

LISTING 5.3 Chap05.mdb: Creating and Storing a Parameterized Query

```
Sub CreateAQueryWithParams()  
    Dim catCurr      As New ADOX.Catalog  
    Dim cmdCurr      As New ADODB.Command  
    catCurr.ActiveConnection = CurrentProject.Connection  
    cmdCurr.CommandText = "Parameters [currDate] Date;" & _  
        "Select * from tblMovieTitles where ReleaseDate = [currDate]"  
    catCurr.Procedures.Append "qryMovieTitlesForReleaseDate", cmdCurr  
    Set catCurr = Nothing  
End Sub
```

Opening a Recordset Off a Parameterized Query

As with DAO, you can open recordsets off parameterized queries. To do so, you work with the Parameters collection, off the Command object. You can see this in Listing 5.4.

LISTING 5.4 Chap05.mdb: Supplying a Parameter and Opening a Recordset

```
Sub ExecuteAQueryWithParams()  
    Dim catCurr      As New ADOX.Catalog  
    Dim cmdCurr      As New ADODB.Command  
    Dim rstCurr      As New ADODB.Recordset  
  
    catCurr.ActiveConnection = CurrentProject.Connection  
    Set cmdCurr = _  
        catCurr.Procedures("qryMovieTitlesForReleaseDate").Command  
    cmdCurr.Parameters("[currDate]") = #2/1/2001#  
    rstCurr.Open cmdCurr, , adOpenForwardOnly, adLockReadOnly, _  
        adCmdStoredProc  
    Debug.Print rstCurr.GetString  
    rstCurr.Close  
End Sub
```

NOTE

The code in Listing 5.4 uses the query that was created using the code in Listing 5.3, so be sure to execute that code first.

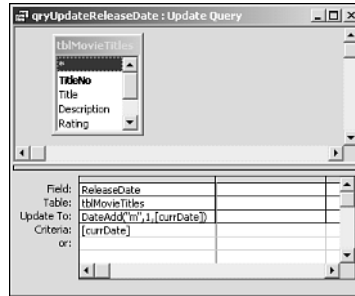
As mentioned, you can use the Parameters collection off the Command object to specify the criteria. The following section describes another way to specify parameters, in which a parameter is specified as an argument for the Execute method off the Command object.

Executing Bulk Queries

Executing bulk queries isn't too tough using ADO, as you can see in Listing 5.5. Here, the qryUpdateReleaseDate query is opened, and the currDate parameter is updated. A shot of qryUpdateReleaseDate follows in Figure 5.6.

LISTING 5.5 Chap05.mdb: Updating the Release Date

```
Sub ExecuteABulkUpdateQuery()  
    Dim catCurr      As New ADOX.Catalog  
    Dim cmdCurr      As New ADODB.Command  
    Dim rstCurr      As New ADODB.Recordset  
    Dim intAffected  As Integer  
    catCurr.ActiveConnection = CurrentProject.Connection  
    Set cmdCurr = catCurr.Procedures("qryUpdateReleaseDate").Command  
    cmdCurr.Execute intAffected, Array(#2/1/2001#), adExecuteNoRecords  
    Debug.Print "Records Affected = " & intAffected  
End Sub
```

**FIGURE 5.6**

Adding a month to the release date passed to the query called qryUpdateReleaseDate.

You can't see the parameter updated by name as in the previous example because it's passed in the `Array()` passed in the `Execute` method. If there were more than one parameter, you would simply add a comma and the second parameter in the `Array()` call.

Finally, notice the `intAffected` variable, which the `Execute` method updates because it's passed by reference by default.

Modifying an Existing Query

To modify a query that already exists, reference the query in the Procedures collection:

```
Sub ModifyingAnExistingQuery()
    Dim catCurr      As New ADOX.Catalog
    Dim cmdCurr      As New ADODB.Command

    catCurr.ActiveConnection = CurrentProject.Connection
    Set cmdCurr = _
        catCurr.Procedures("qryMovieTitlesForReleaseDate").Command
    cmdCurr.CommandText = "Parameters [currDate] Date;" & _
        "Select * from tblMovieTitles where ReleaseDate = [currDate]"
        ➔Order By Title"
    Set catCurr.Procedures("qryMovieTitlesForReleaseDate").Command = cmdCurr
    Set catCurr = Nothing
End Sub
```

End Sub

Deleting a Query

To delete an existing query through code, use the `Delete` method off the Procedures collection:

```
Sub DeleteAQuery()
```

```

Dim catCurr      As New ADOX.Catalog
catCurr.ActiveConnection = CurrentProject.Connection
catCurr.Procedures.Delete "qryMovieTitlesForReleaseDate"
End Sub

```

Working with Tables

Working with tables in ADO is similar to working with them in DAO. Rather than work with the TableDefs collection, you work with the Tables collection. The Tables collection is located off the Catalog object. One difference between ADO's Tables collection and DAO's TableDefs collection is that more than just standard table and link table objects are stored in it. Table 5.4 lists the object types in the ADO Tables collection.

TABLE 5.4 Table Type Objects Stored in the Tables Collection

<i>Object Type</i>	<i>Description</i>
Access Table	Access system table
Link	Linked table from a non-ODBC data source
Pass-Through	Linked table from an ODBC data source
System Table	Jet system table
Table	Table
View	Row returning, non-parameterized query

You can use a couple of methods to see what objects are in the Tables collection for a specific type. You can use the traditional DAO method by using the Tables collection and looking at the Table object's Type property (see Listing 5.6).

LISTING 5.6 Chap05.mdb: Looking at the Table Type Object Through the Tables Collection

```

Sub ListTablesUsingTableCollection(strTypeToList As String)
    Dim catCurr      As New ADOX.Catalog
    Dim tblCurr      As ADOX.Table
    catCurr.ActiveConnection = CurrentProject.Connection
    For Each tblCurr In catCurr.Tables
        If tblCurr.Type = strTypeToList Then
            Debug.Print tblCurr.Name
        End If
    Next
End Sub

```

The other available method is `OpenSchema` off the `Connection` object (see Listing 5.7). This method is actually faster because you don't have to create the individual `Table` variables.

LISTING 5.7 Chap05.mdb: Using `OpenSchema` to See the `Tables` Collection's Object Types

```
Sub ListTablesUsingSchema(strTypeToList As String)
    Dim rstSchema As ADODB.Recordset
    Set rstSchema = CurrentProject.Connection.OpenSchema(adSchemaTables)
    Do Until rstSchema.EOF
        If rstSchema.Fields("TABLE_TYPE") = strTypeToList _
            Or strTypeToList = "ALL" Then
            Debug.Print rstSchema.Fields("TABLE_NAME")
        End If
        rstSchema.MoveNext
    Loop
End Sub
```

Creating a New Table with Fields and Indexes

Creating tables in ADO is very much like creating them in DAO, where you append objects to the various necessary collections. Unlike DAO, you don't have to use the `CreateObject()` methods. Another difference is the use of the `Keys` collection, which varies from the `Indexes` collection in that it specifies a table's primary and foreign keys. Listing 5.8 shows the creation of a table, along with fields and the primary key field.

LISTING 5.8 Chap05.mdb: Creating a Table with Fields and Keys

```
Sub CreateTableWithIndexes()
    Dim catCurr As New ADOX.Catalog
    Dim tblNew As New ADOX.Table
    Dim keyNew As New ADOX.Key
    catCurr.ActiveConnection = CurrentProject.Connection

    '-- Create a new Table object.
    With tblNew
        .Name = "MyNewTable"
        '-- Add new columns
        '-- Add an autoincrement field
        .Columns.Append "MyKeyField", adInteger
        .Columns("MyKeyField").ParentCatalog = catCurr
        .Columns("MyKeyField").Properties("AutoIncrement") = True
        '-- Add a character field
        .Columns.Append "MyCharField", adWChar
        '-- Add a Memo field
```

LISTING 5.8 Continued

```

.Columns.Append "MyMemoField", adLongVarChar
.Columns("MyMemoField").Attributes = adColNullable
'-- Add a Primary Key field
keyNew.Name = "PrimaryKey"
keyNew.Columns.Append "MyKeyField"
.Keys.Append keyNew, adKeyPrimary
End With

'-- Add the new table to the database.
catCurr.Tables.Append tblNew
Set catCurr = Nothing

```

End Sub

Look at Listing 5.8 step by step:

1. Set the catalog's connection, and then specify the name of the new table. As with DAO tables, the table won't actually be added to the Tables collection until the Append method is specified later.

```
catCurr.ActiveConnection = CurrentProject.Connection
```

```

'-- Create a new Table object.
With tblNew
    .Name = "MyNewTable"

```

2. Add the columns to the new table:

```

' - Add an autoincrement field
.Columns.Append "MyKeyField", adInteger
.Columns("MyKeyField").ParentCatalog = catCurr
.Columns("MyKeyField").Properties("AutoIncrement") = True
'-- Add a character field
.Columns.Append "MyCharField", adVarChar
'-- Add a Memo field
.Columns.Append "MyMemoField", adLongVarChar
.Columns("MyMemoField").Attributes = adColNullable

```

In this case, three different data types are added to the table: Long Integer (adInteger for ADO), Character, and Memo. Table 5.5 lists the various data types possible.

TABLE 5.5 Field Data Types Used in ADO, with Their DAO Equivalent

<i>ADO Data Type</i>	<i>DAO Data Type</i>
adBinary	dbBinary
adBoolean	dbBoolean

TABLE 5.5 Continued

<i>ADO Data Type</i>	<i>DAO Data Type</i>
adUnsignedTinyInt	dbByte
adCurrency	dbCurrency
adDate	dbDate
adNumeric	dbDecimal
adDouble	dbDouble
adGUID	dbGUID
adSmallInt	dbInteger
adInteger	dbLong
adLongVarBinary	dbLongBinary
adLongVarChar	dbMemo
adSingle	dbSingle
adWChar	dbText

3. Add the primary key to the table:

```
'-- Add a Primary Key field
keyNew.Name = "PrimaryKey"
keyNew.Columns.Append "MyKeyField"
.Keys.Append keyNew, adKeyPrimary
```

4. Append the new table to the Tables collection:

```
'-- Add the new table to the database.
catCurr.Tables.Append tblNew
```

Modifying an Existing Table by Adding an Index

To modify the table, simply refer to it in the Tables collection. To add a new index, create it and add it to the Indexes collection.

```
Sub ModifyTableAndAddIndex()
    Dim catCurr As New ADOX.Catalog
    Dim idxNew As New ADOX.Index
    catCurr.ActiveConnection = CurrentProject.Connection
    With catCurr.Tables("MyNewTable")
        idxNew.Name = "MyCharField"
        idxNew.Columns.Append "MyCharField"
        .Indexes.Append idxNew
    End With
End Sub
```

Summary

As of this writing, a great reference for switching from DAO to ADO is a Microsoft white paper titled “Migrating from DAO to ADO.” You can find this paper at www.sampublishing.com (enter this book’s ISBN in the Search field).

You can find more ADO examples throughout the rest of book, particularly in these chapters:

- Chapter 22, “Welcome to the World of Database Replication,” takes advantage of the replication features available only through ADO.
- Chapter 24, “Developing SQL Server Projects Using ADPs,” shows how to use ADO from an Access database project to work with stored procedures.
- Chapter 25, “Startup Checking System Routines Using ADO,” shows you how to use ADO to manage your applications on startup.
- Chapter 26, “Creating Maintenance Routines,” discusses some maintenance routines created by using ADO.

IN THIS CHAPTER

- **Getting to Know XML 134**
- **Working with XML with the Access User Interface 139**
- **Coding with XML and VBA in Access 2002 142**
- **Taking Advantage of the Other Office Applications' XML Support 144**

Well, here we go with another buzz word: *XML*. How can such a short acronym, and even a smaller command set, cause such a flurry? As the Internet comes of age, some developers are hesitant to jump in and learn every new Web language that comes along. Well, let me tell you, XML is not one of those.

Microsoft, like most other Internet proponents, is making a point to fully support a standard version of XML in its Office products, as well as its various development languages. Access 2002 allows you to create XML documents from Access reports or Data Access Pages that can be used directly on the Web or by other applications. You can also use the XML Object Model to manipulate XML documents with VBA. Data can also be imported and exported to XML documents.

NOTE

As mentioned, XML also has been introduced into Access with Data Access Pages (DAPs). In Access 2002, Microsoft enhanced DAPs in many ways, such as the use of XML for underlying data, and the ability to take DAPs offline while using XML for the data storage. For a complete discussion on DAPs, see Chapter 12, “Working with Data Access Pages.”

Before showing how to use XML in Access 2002, let’s take a closer look at what XML is.

Getting to Know XML

XML, short for *Extensible Markup Language*, was originally created to use on the Web. *HTML* (*Hypertext Markup Language*) was created to display and control Web pages, and display data on those Web pages. However, HTML doesn’t say how the data should be handled; XML does that. XML, as with HTML, is simply text set up in such a way that it can be read by not only programs, but also humans.

Looking at XML’s History

XML was started by the *World Wide Web Consortium (W3C)*, which first met in 1996. XML is made up of standards in a format such that any language using that format can read the XML data. This is very powerful, and goes way beyond just uses for the Internet. For example, a doctor’s office has created an administrative system. My client, the doctor, wants me to use data from the system to create custom reports in Access. If the system dumps data out in XML format, with Access 2002, I could read the file by either importing it into a table, or read it directly from the file by using data access pages.

Now you could say, “Well, you could do this by simply exporting the data to an ASCII text file.” But the difference is that you can also send that file to Excel, and even the Web, and they would be able to read it without changing anything in the file. By taking some of the XML features mentioned in this chapter, they could even view the data while someone is working (updating) the information. You can see an example of this in Figure 6.1, where the data displayed in three different products were all from the same XML file, Customers.XML, found on this book’s Web page at www.sampublishing.com.

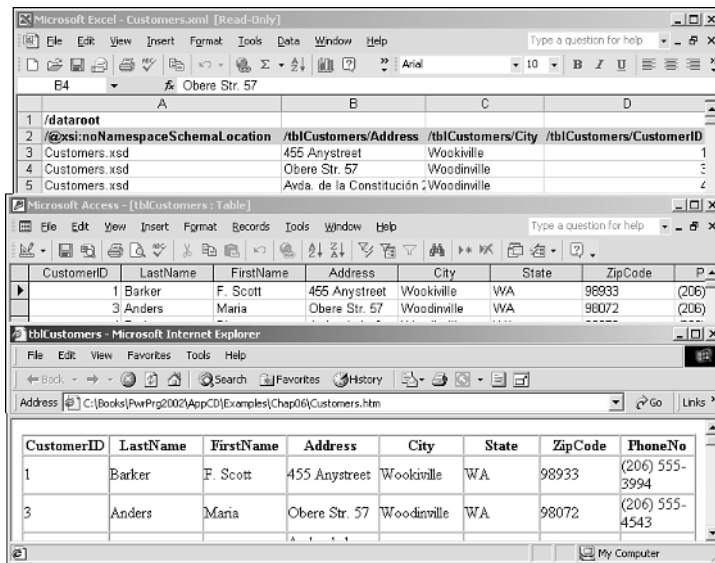


FIGURE 6.1

XML enables various applications to share data in ways like never before possible.

Examining the Files That Make Up XML Documents

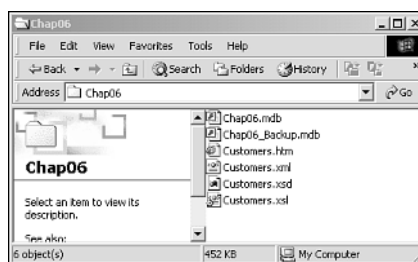
After you export data to XML format from Access (described in a later section), you will be able to point either to an HTML file created, which will let you display the file in a Web browser, or directly to an *.XML file.

In actuality, unlike other file formats such as ASCII text files, the XML format uses a number of files to structure itself. Other structures (files) can be used to set up how the data is presented. Table 6.1 shows a few file types used to define XML data.

TABLE 6.1 File Types Associated with XML Documents

Extension	Description
*.XML	The XML data document. This is a static snapshot of data itself.
*.XSD	The schema file. This schema was based off the persisted table or query, and is in the W3C XSD standard.
*.XSL	Presentation document. Specifies how the data in the *.XML is to be displayed.
*.HTM	Final Package. This ties the *.XML (data) and *.XSL (presentation) together to be used on the Web.

These files, also referred to as *documents*, can all be found on this book's Web page at www.sampublishing.com. You can see the files and their icons in Figure 6.2.

**FIGURE 6.2**

These files allow data to be used by various applications and over the Web in a standard way.

I mentioned at the beginning of the section that the XML document is structured so that humans as well as computers can read it. As an example, Listing 6.1 shows some of the document called Customers.XML.

LISTING 6.1 Customers.XML: File Containing the Data Itself

```
<?xml version="1.0" encoding="UTF-8" ?>
- <dataroot xmlns:od="urn:schemas-microsoft-com:officedata"
  xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="Customers.xsd">
- <tblCustomers>
  <CustomerID>1</CustomerID>
  <LastName>Barker</LastName>
  <FirstName>F. Scott</FirstName>
  <Address>455 Anystreet</Address>
  <City>Wookiville</City>
  <State>WA</State>
```

LISTING 6.1 Continued

```

    <ZipCode>98933</ZipCode>
    <PhoneNo>(206) 555-3994</PhoneNo>
  </tblCustomers>
- <tblCustomers>
  <CustomerID>3</CustomerID>
  <LastName>Anders</LastName>
- <FirstName>
- <![CDATA[
Maria
  ]]>
  </FirstName>

```

Remember, the XML document describes the data itself. You can see that the schema file, Customers.xsd, is referenced toward the top of the file. This file sets up how the data itself is structured. If this file isn't included, the systems need to figure out how to handle the data. This way, data can be interpreted more concisely, and how the originator of the data intended. Listing 6.2 shows a part of Customers.xsd.

LISTING 6.2 Customers.xsd: Schema File Used With Customers.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
xmlns:od="urn:schemas-microsoft-com:officedata">
<xsd:element name="dataroot">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element refer="tblCustomers"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
<xsd:element name="tblCustomers">
  <xsd:complexType>
    <xsd:all>
      <xsd:element name="CustomerID" od:jettype="autonumber"
        od:sqltype="int" od:autounique="yes" od:nonnullable="yes"
        od:adotype="3">
        <xsd:simpleType>
          <xsd:restriction base="xsd:integer"/>
        </xsd:simpleType>
      </xsd:element>

```

As you can see from this schema file, you can specify data types as well as other properties about the schema of the data being used.

The presentation document, Customers.xml, describes how the data will be displayed in the browser. You can see the source for this file in Figure 6.3.

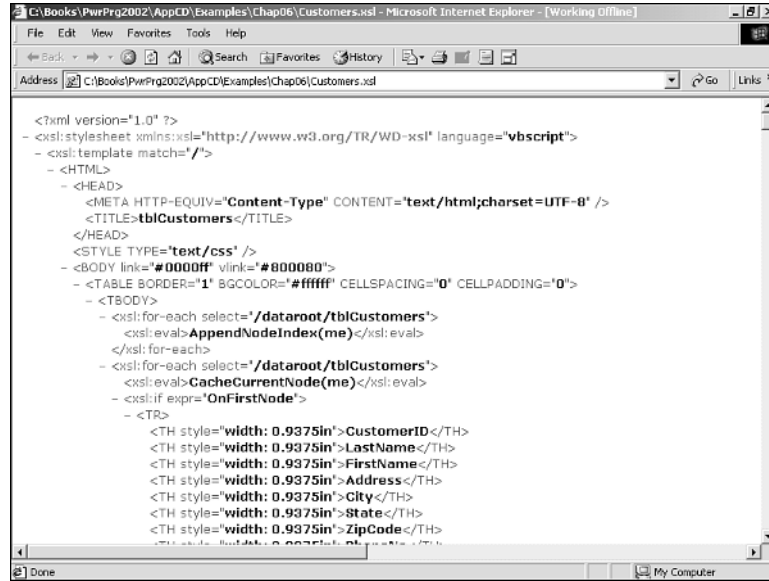


FIGURE 6.3

*Notice that even border colors for tables are specified by using the *.xsl file.*

The final file, Customers.htm, pulls them all together. This document is specified in your Web page. Listing 6.3 shows the entire contents of the file.

LISTING 6.3 Customers.HTM: The VBScript Used Directly in Web Sites

```
<HTML xmlns:signature="urn:schemas-microsoft-com:office:access">
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=UTF-8" />
</HEAD>

<SCRIPT event=onload for=window>
  objData = new ActiveXObject("MSXML.DOMDocument");
  objData.async = false;
  objData.load("Customers.xml");
  if (objData.parseError.errorCode != 0)
    alert(objData.parseError.reason);
```

LISTING 6.3 Continued

```
objStyle = new ActiveXObject("MSXML.DOMDocument");
objStyle.async = false;
objStyle.load("Customers.xsl");
if (objStyle.parseError.errorCode != 0)
    alert(objStyle.parseError.reason);

document.open("text/html","replace");
document.write(objData.transformNode(objStyle));
</SCRIPT>

</HTML>
```

Although I would love to get into details of XML development, whole books have been written for this topic. In fact, for further XML instruction, check out Que Publishing's *XML By Example*, by Benoit Marchal. My goal for this chapter is to give you an overview of XML, and how Access has been enhanced to work with XML documents.

Working with XML with the Access User Interface

Microsoft has added quite a few features to Access 2002 that have to do with XML. The first two that immediately come to mind are the exporting and importing of XML documents through the Access user interface (menus).

Exporting from Access to XML

Exporting a table or any Access object to XML is similar to exporting to other file formats, with a few exceptions. To export a table:

1. Highlight the table and choose Export from the File menu.
2. In the Export Table dialog, choose XML Documents (*.xml) for the Save as Type, and click Export.

In the Export XML dialog, you can choose the information you want to include with the data. You can choose to include one or all of the following:

- Data (XML) is usually a good thing to include. In some cases, however, you'll want to export just the schema and hold off on the data.
- Choosing Schema generates a schema that provides additional information for applications using the data exported.
- Choosing Presentation (XSL) provides a nice format based on a template you provide. The default for this is an HTML 4.0 sample XSL.

At this point, you could simply click OK, and Access will create the specified files. However, you can get even more detailed by clicking the Advanced button. If you do so, you will be able to have more control over how Access creates your XML documents, as shown in Figure 6.4.

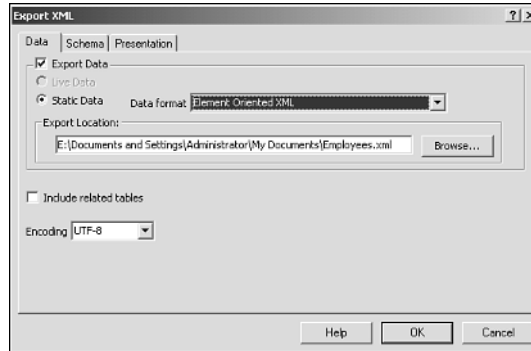


FIGURE 6.4

By clicking Advanced, you now have more control over the three choices provided on the Export XML dialog: Data, Schema, and Presentation.

On the Data page, you can choose to create live connections (reports only and when you are using an ADP with SQL Server back end) or static. If you are exporting as static data, you can specify where you want the *.xml document to be stored. Finally, you can specify how you want the data converted (Encoding), where the default is UTF-8.

Clicking the Schema tab reveals the page in Figure 6.5.

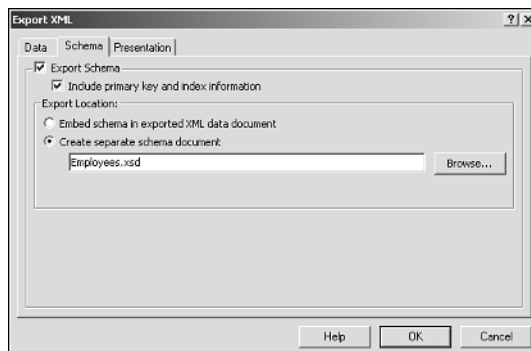


FIGURE 6.5

You can specify on this page what information you want included in the schema, as well as whether to include the schema in the data.

TIP

For better performance, you will want to keep the schema separate from the data, so choose Create Separate Schema Document.

6

After finishing with this page, click the Presentation tab (see Figure 6.6).

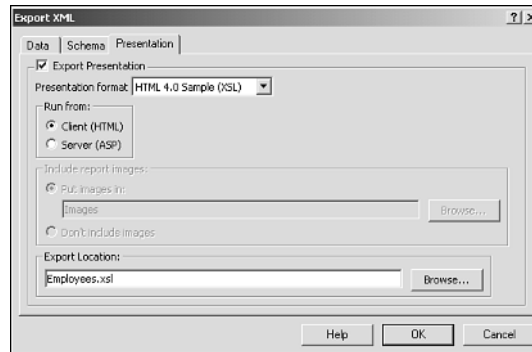


FIGURE 6.6

Use this page to specify how you want data presented.

If you choose to specify the presentation information, you will be creating the *.xsl document. Some choices on this page are where you want the presentation to take place: On the client machine with an HTML page, or on the server, where an ASP page will be created. You can also specify where you want the *.xsl file placed, and if and where you want images included.

After you choose your options, you can click OK to create your files. You then just have to point to the created *.HTM file and have the code do the rest.

Differences in Exporting Between Access User Interface and ADO

The main difference between exporting from Access (through the user interface) and exporting from ADO (discussed later in the section “Coding with XML and VBA in Access 2002,”) is that

- ADO creates attribute-centric XML. This results in a more finely specified XML file that’s defined to a greater level.
- Access will export element-centric XML, resulting in a more loosely defined XML file that might be read by more applications, but the application will need to fill in some of the blanks as to how the data is to be handled.

Importing an XML Document

Importing XML documents into Access takes fewer steps than it does to export. To import XML data into Access, follow these steps while in the database window:

1. Choose Get External Data from the File menu.
2. Choose Import. The Import dialog appears.
3. Select XML Documents (*.xml, *.xsd) from the Files of Type drop-down list. You can then locate the XML document or schema that you want to import.
4. When you have find the document, highlight it and click OK.

You will then be taken to the XML Import dialog. This is a simple dialog, and when you click the Options button, you will see a dialog similar to Figure 6.7.

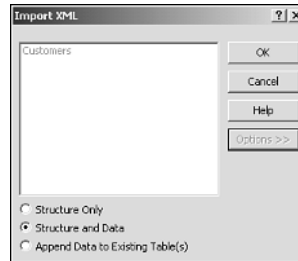


FIGURE 6.7

This dialog allows you to import an XML schema, import XML data and schema, or append XML data to existing data in Access tables.

A few issues concern the kind of XML documents that you can import into Access:

- Access can import only one document at a time.
- The data must be in a format that Access recognizes, either in a native format or through the use of a schema.
- The entire file has to be imported. You can't choose a subset of the XML document.

Coding with XML and VBA in Access 2002

Using code to export and import XML involves two main commands. For exporting, there is the ExportXML method off the Application object. The syntax for the command is as follows:

```
ExportXML(ObjectType As AcExportXMLObjectType, DataSource As String, _
    [DataTarget As String], [DataTransform As String], _
    [SchemaTarget As String], [SchemaFormat As AcExportXMLSchemaFormat _
    = acSchemaNone], [SchemaTransform As String], _
    [PresentationTarget As String], [ImageTarget As String], _
    [LiveReportSource As String], [Encoding As AcExportXMLEncoding _
    = acUTF8], [OtherFlags As Long])
```

You can follow the parameters used for the ExportXML method by looking at the Export XML dialog shown earlier in Figures 6.4, 6.5, and 6.6.

For importing XML, use the ImportXML method off the Application object:

```
ImportXML(ObjectType As AcImportXMLObjectType, DataSource As String, _
    [DataTransform As String], [SchemaFile As String], _
    [SchemaTransform As String], [OtherFlags As Long])
```

For an example of these two commands in action, look at the following simple snippet:

```
Sub ShowXMLImportExport()
    Application.ImportXML CurrentProject.Path & "\XMLProducts.xml", _
        acStructureAndData

    DoCmd.RunSQL _
        "UPDATE XMLProducts SET XMLProducts.UnitPrice = [UnitPrice]*1.05;"

    Application.ExportXML acExportTable, "XMLProducts", CurrentProject.Path & _
        "\XMLProducts.xml", CurrentProject.Path & "\XMLProducts.xsd"

    DoCmd.DeleteObject acTable, "XMLProducts"
End Sub
```

The ShowXMLImportExport routine, in the modXMLImportExportExample module, can be found in Chap6.mdb on this book's Web page at www.sampublishing.com. The routine does the following:

1. It imports the XMLProducts.xml file (found with the Chap6.mdb database).
2. It runs an update query to add 5% to the data in the UnitPrice field.
3. The routine exports the file back to XMLProducts.xml and creates a schema file named XMLProducts.xsd.
4. It deletes the XMLProducts table that was created by importing the XML data.

This is just one way to deal with XML data from Access. Another way is to use other Office applications to work with XML data.

Taking Advantage of the Other Office Applications' XML Support

While Access got, by far, the most XML features thrown in, some of the other Microsoft Office products got some as well. Excel and Word both have XML features that are worth mentioning—Excel in particular.

Introducing the Excel XML Spreadsheet Schema (XML SS)

The main feature added to Excel is the capability to create an XML spreadsheet using the Save As command. Some of the capabilities this gives developers are as follows:

- Materials that need to be read over the Web real time. This means that users can still update the data, while people accessing the Web pages see the changes immediately.
- Users continue to work on an *.XML file from within Excel, just as they would a regular spreadsheet file. They can create a spreadsheet model in Excel (including all the formatting, formulas, data validation, and so on), and then save this model as XML SS, without losing any of the richness.
- Microsoft Spreadsheet Control can display in a Web page.

NOTE

XML SS is valid, well-formed XML. What this means is that the information within an XML spreadsheet can be used with various XML enabled applications, while also being unlocked.

To create an XML spreadsheet from within Excel, choose Save As from the File menu, and then select XML Spreadsheet for the type of file to save as. After clicking Save, you see a dialog that will inform you that some objects won't be saved (see Figure 6.8).

After you create the XML Spreadsheet document, you can use it from a Data Access Page using Office's Spreadsheet Web component, seeing updates immediately when the spreadsheet is modified.

**FIGURE 6.8**

You can see the XML Spreadsheet file type behind the dialog informing you that VB code and Shapes won't be exported.

Summary

The XML enhancements to Access are immense in Access 2002. They continue to increase Access's power, and make it all that much more of a viable enterprise tool. You can learn many things when working with XML documents. Hopefully, this chapter has at the very least increased your understanding of XML, and how you can use it with Access.

For more information on using XML with your application, see the following chapters:

- Chapter 12, "Working with Data Access Pages," gives an overview of using Data Access Pages and the types of files created by them.
- Chapter 19, "Using Access with the Internet," discusses more of the features that enable you to use Access with the Internet.

Handling Your Errors in Access with VBA

CHAPTER

7

IN THIS CHAPTER

- **Examining Access's Runtime Error Handling 148**
- **Working with the Err and Error Objects 153**
- **Working with the ADO Errors Collection 157**
- **Creating User-Defined Errors 161**
- **Using Custom Error Logs to Track Errors 163**
- **Creating a Centralized Error-Handling Routine 171**
- **A Last Look at Error-Handling Issues 176**

Although it's not the most glamorous subject for a book, runtime error handling is one of the most important subjects when it comes to creating robust applications. Access has several commands that help you use error handling to trap all those bugs that either slip through when you're debugging an application, or come about through no fault of yours but because of bad data or user error. Access's error handling is very straightforward—when you get the hang of it. It's up to you, the developer, to take advantage of its power.

When an error occurs, Access simply displays a description of the error by default. The retail version of Access gives you the dialog shown in Figure 7.1. In this dialog, choosing Debug causes the VBE to appear; choosing Continue ignores the error; choosing End stops execution. If you are using an application distributed with the runtime product, you're given the option to retry or cancel, and then are dumped to Windows.

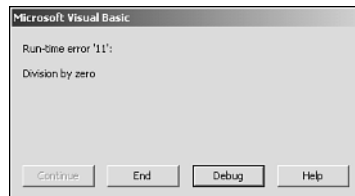


FIGURE 7.1

After you get this error message, Access's default error handler lets you debug the routine.

You can use error handling in many ways to your benefit. For example, with error handling, you can account for situations that can't be debugged at development time. These are just a few situations that can occur during runtime and can be handled by using Access's error-handling commands:

- The user forgets to place the disk in the floppy drive, causing an error. You can have the system prompt the user to place the disk in the drive and then try again.
- The user deletes some tables from the database through code, but one table is already deleted. This normally causes an error. In some cases you might not care, so you can have the error handler continue with the next line in the code.
- When someone moves the back end (the database on the server) to another location on the network, you can test for it when starting up the application. To do so, you need to turn off error handling and trap the error yourself. If a particular error has occurred, you can call a routine to relink the tables.

Examining Access's Runtime Error Handling

When working with Access's error handling, you can use these commands: `On Error`, `Exit`, and `Resume`.

Using the On Error Command

On Error lets you specify exactly where you want Access to proceed to when an error occurs. The following variations of the On Error command are used to direct program execution.

The On Error Goto Statement

To control error handling, you must take over for the default handler by issuing an On Error Goto statement:

```
On Error Goto LineNumber|LineLabel
```

The On Error Goto statement has two options:

- *LineNumber* can be a number placed in your routine. Although VBA lets you use line numbers to specify lines to go to, doing so is not good practice. Goto tends to lead to unreadable code, unless it's used with the On Error statement.
- *LineLabel* is a label signifying the start of a section of a routine, equivalent to a label used in a DOS batch file.

The code in Listing 7.1 shows a very simple error handler. You can find the SimpleError subroutine in the modErrorHandler module in the VideoApp(ADO).mdb database, which you can find on this book's Web page at www.sampublishing.com.

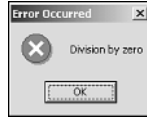
LISTING 7.1 VideoApp(ADO).mdb: Creating a Simple Error Handler

```
Sub SimpleError(intNumerator as Integer, intDenominator as Integer)
    On Error GoTo SimpleError_Error
    Debug.Print intNumerator / intDenominator
    Exit Sub
SimpleError_Error:
    MsgBox Err.Description, vbCritical, "Error Occurred"
    Exit Sub
End Sub
```

SimpleError_Error is the line label used for the error handler in Listing 7.1. Rather than give you the dialog shown in Figure 7.1 as in the default error handler, this routine displays an error message and then exits the subroutine. Figure 7.2 shows the message box generated by the error handler in the SimpleError routine.

NOTE

The Err object in the line `MsgBox Err.Description, vbCritical, "Error Occurred"` is discussed later in the section "Working with the Err and Error Objects."

**FIGURE 7.2**

After receiving this simple message, the user is exited directly from the routine with the error.

The On Error Resume Next Statement

Use `On Error Resume Next` when the program needs to continue with just the next line of code if an error occurs. Suppose that you're trying to delete a table that has already been deleted. The following is the code for this example:

```
Sub InlineResumeNextExample()
'-- This command tells Access to ignore errors and resume on the next line.
  On Error Resume Next
  DoCmd.DeleteObject acTable, "NoTable"
End Sub
```

The On Error GoTo 0 Statement

Use `On Error GoTo 0` when you want to have Access either return to its default error handler, or refer to an error handler that was enabled in a routine above the current routine (also called *nesting* an error). More information on nesting error handlers is discussed later in the section “Nesting Error Handlers.”

Using the Exit Sub|Function Command

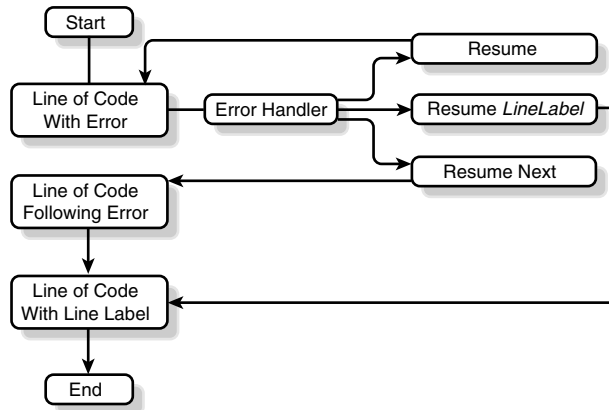
Depending on the type of routine that's executing, you can use `Exit Sub|Function` to exit directly from the error handler. The `SimpleError` routine in Listing 7.1 uses the `Exit Sub` command to leave the routine after the error is displayed.

Using Resume, Resume Next, and Resume LineLabel

The `Resume` statements are used within the error handler to continue with the routine after the error is handled. You can use three alternatives: `Resume`, `Resume Next`, and `Resume LineLabel`. Figure 7.3 shows how these options work.

The Resume Statement

`Resume` by itself retries the line on which the error occurred. This statement is used when the error can be fixed and then retried. An example of using the `Resume` statement is when the user tries to copy a file to a floppy in drive A when no disk is in the drive. By using the `Resume` statement, you can prompt the user to insert the disk and then retry.

**FIGURE 7.3**

Resume, Resume Next, and Resume LineLabel give you flexibility in handling errors.

Listing 7.2 shows the use of `Resume` to prompt users to re-enter a value for the denominator. Figure 7.4 shows the input box displayed after you get the error.

LISTING 7.2 VideoApp(ADO).mdb: Resuming After an Error

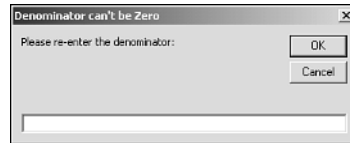
```

Sub ResumeExample(intNumerator As Integer, intDenominator As Long)
    On Error GoTo Error_ResumeExample
    Debug.Print intNumerator / intDenominator

    Exit Sub

Error_ResumeExample:
    If Err.Number = apDivisionByZero Then 'Division by zero error (11)
        intDenominator = InputBox("Please re-enter the denominator:", _
            "Denominator can't be Zero")
        Resume
    Else
        Exit Sub
    End If
End Sub
  
```

If the value entered for the denominator is a different data type, such as text, the routine in Listing 7.2 will halt execution and dump the user into the code module editor. In this case, you need an error handler outside the routine. The `CallResumeExample` routine in Listing 7.3 traps the error that occurs in the `ResumeExample`'s error handler. This is called *nesting* error handlers, which you can read more about in “Nesting Error Handlers” later in this chapter.

**FIGURE 7.4**

With Resume, you can allow users to fix the error and try again.

LISTING 7.3 VideoApp(ADO).mdb: Trapping the Error from ResumeExample

```
Sub CallResumeExample(intNumerator As Integer, intDenominator As Integer)
    On Error GoTo Error_CallResumeExample
    ResumeExample intNumerator, intDenominator

    Exit Sub

Error_CallResumeExample:
    MsgBox Err.Description
    Exit Sub

End Sub
```

The Resume Next Statement

In an error handler, `Resume Next` provides the capability to give a message and handle the error, and then move to the next line of code from where the error occurred. The routine in Listing 7.4 displays a message that the table doesn't exist and then to continue to the next line.

LISTING 7.4 VideoApp(ADO).mdb: Resuming to the Next Line

```
Sub ResumeNextExample()
    On Error GoTo Error_ResumeNextExample
    DoCmd.DeleteObject acTable, "NoTable"

    Exit Sub

Error_ResumeNextExample:
    MsgBox Err.Description, vbCritical, "Delete Table Error"
    Resume Next

End Sub
```

In this case, `Resume Next` doesn't do much for you. But sometimes the capability to skip to the line following the error comes in handy.

The Resume *LineLabel* Statement

Resume *LineLabel* allows you to resume at a specified line label—usually one placed before cleanup and exit statements. This statement is generally used to create one exit for the routine. You can see from the subroutine in Listing 7.5 that in the error handler, Access resumes to the Exit_ResumeLineLabelExample line label.

LISTING 7.5 VideoApp(ADO).mdb: Resuming to a Line Label

```
Sub ResumeLineLabelExample()  
    On Error GoTo Error_ResumeLineLabelExample  
    DoCmd.DeleteObject acTable, "NoTable"  
  
    '-- This label starts the cleanup and exit section  
Exit_ResumeLineLabelExample:  
    Exit Sub  
  
    '-- This label is for the error handler  
Error_ResumeLineLabelExample:  
    MsgBox Err.Description, vbCritical, "Delete Table Error"  
    Resume Exit_ResumeLineLabelExample  
  
End Sub
```

Additional examples are shown later in this chapter. One item that you might have noticed throughout the examples is the use of the Err object.

Working with the Err and Error Objects

The different ways of using the Err object (used by Access) and Error objects (used by ADO), as well as the Error statement, can make them confusing. It's best to examine them in the order that you'll use them.

As of Access 97, The Err object contains information about the most recent error that has occurred. The Err object takes the place of the Err statement and Error\$ in versions before Access 95. Like other Access objects, the Err object contains properties and methods. It has two methods, Clear and Raise, and the properties listed in Table 7.1.

TABLE 7.1 Properties Used with the Err Object

<i>Property</i>	<i>Description</i>
Description	Contains the description of the error if a description exists.
HelpContext	Stores the context ID for the VB help file.

TABLE 7.1 Continued

<i>Property</i>	<i>Description</i>
HelpFile	Stores the path and filename of the VB help file.
LastDLLError	Returns the last error in a 32-bit DLL call. For more information on <i>dynamic link libraries (DLLs)</i> , see Chapter 15, “Extending the Power of Access with API Calls.”
Number	Contains the number of the error that has been set in the Error object. This is the default property for the Err object.
Source	Reflects the system in which this error occurred. Most Access errors set the Source property to MSAccess. When Automation is used, if another application (such as Excel) caused the error, the Source property is set to that application.

The Err Object’s Clear Method

The Clear method allows you to clear the last error that’s now in the Err object. With Clear, you can retry an action in a loop that might cause an error. After the error occurs, you might want to start at the top of the loop and recheck the error. So, at the bottom of the loop, you perform an Err.Clear to set the Err.Number object back to 0.

NOTE

Performing an Exit Sub|Function to exit a routine, performing a Resume statement, or using another On Error statement automatically clears the Err object.

The Err Object’s Raise Method

Raise allows you to generate a runtime error on purpose. This is useful when debugging an application or creating a custom error. Properties for Raise include the same properties used for the Err object itself: Description, HelpContext, HelpFile, Number, and Source.

NOTE

For compatibility purposes, you can still use the Error statement, but the Err.Raise method is the preferred method in VBA.

Listing 7.6 shows the `Raise` method used to test the centralized error handler discussed later in the section “Creating a Centralized Error-Handling Routine.” You can find this code on the `frmTestErrors` form in the `VideoApp(ADO).mdb` database, which is found on this book’s Web page at www.sampublishing.com. The `cmdTestRaiseError_Click()` procedure is attached to the `cmdTestCustomError` command button’s *OnEvent* event.

LISTING 7.6 VideoApp(ADO).mdb: Raising an Error on Purpose

```
Private Sub cmdTestRaiseError_Click()  
    Dim apRoutineError As String  
    apRoutineError = "cmdTestRaiseError_Click"  
  
    On Error GoTo Error_cmdTestRaiseError_Click  
    Err.Raise 11 ' Division By Zero Error  
    Exit Sub  
  
Error_cmdTestRaiseError_Click:  
    apCurrErrNo = Err.Number  
    apCurrErrMsg = Err.Description  
    ap_ErrorLog apModuleError, apRoutineError, apCurrErrNo, _  
        apCurrErrMsg, CurrentProject.Connection  
    Select Case ap_ErrorHandler(apCurrErrNo, apCurrErrMsg)  
        Case apTryAgain  
            Resume  
        Case apExitRoutine  
            Exit Sub  
        Case apResumeNext  
            Resume Next  
    End Select  
  
End Sub
```

The `Raise` method also generates user-defined errors. Use `Raise` when you want to send errors through standard error handlers with your own information. You can set up Automation errors in this way.

TIP

You can use the `Raise` method to test the error handlers you create. Look for this method in action later in the section “Using Custom Error Logs to Track Errors.”

To use the `Err.Raise` method, you need to supply the error information via the `Err` object's properties mentioned earlier in Table 7.1. The following code shows constants used for a new error, located in the declarations section of the `frmTestErrors` form:

```
Const apErrNoDatabaseNotFound = 32000
Const apErrMsgDatabaseNotFound = "No Database found in this Directory"
```

Next, this routine asks users to supply a folder where the database can be found. If the database isn't found in this folder, the routine uses the new error associated with `apErrNoDatabaseNotFound`. The code in Listing 7.7 is also found on the `frmTestErrors` form, attached to the `OnClick` event of the `cmdTestUserDefinedError` command button. Figure 7.5 shows the result of running this routine.

LISTING 7.7 VideoApp(ADO).mdb: Using Custom Error Messages

```
Private Sub cmdTestUserDefinedError_Click()
    Dim strDirToLook As String
    Dim strMDBFound As String
    On Error GoTo Error_cmdTestUserDefinedError_Click
    '-- Looking for the first MDB found
    strDirToLook = InputBox("Please enter a directory:")
    strMDBFound = Dir(strDirToLook & "\*.mdb")
    '-- if no mdb found call the standard error handler with our error
    If Len(strMDBFound) = 0 Then
        Err.Raise apErrNoDatabaseNotFound, strDirToLook, _
            apErrMsgDatabaseNotFound
    Else
        MsgBox strMDBFound
    End If
    '-- Other code if desired
    Exit Sub

Error_cmdTestUserDefinedError_Click:
    MsgBox Err.Description, vbCritical, Err.Source
    Exit Sub
End Sub
```

TIP

You can use another set of functions for creating custom error messages: `CVErrors()` and `IsError()`. For more information on these functions, see "Creating User-Defined Errors" later in this chapter.

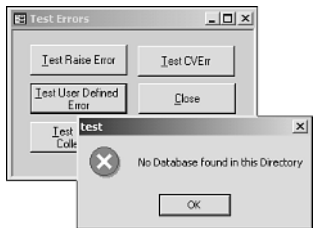


FIGURE 7.5
Creating your own error codes is useful for customizing your application error reporting.

That’s it for the properties and methods of the Err object. There are also collections for errors that occur while working with Data Access Objects (DAO) and ActiveX Data Objects (ADO).

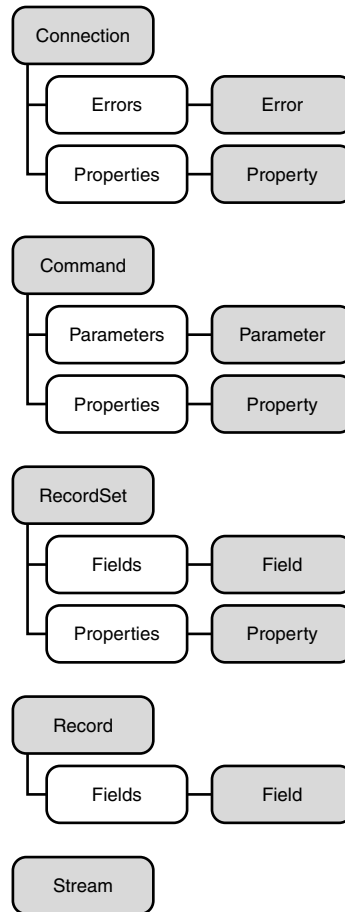
Working with the ADO Errors Collection

The Errors collection stores the most recent set of multiple errors that can occur with features such as ODBC and ADO. In the ADO hierarchy, the Errors collection hangs off the Connection object (see Figure 7.6).

The Errors collection contains one method, Refresh, and one property, Count, which is the number of the Error objects the collection contains. The Errors collection is made up of Error objects, which have some of the same properties as the Err object (see Table 7.2). Error objects have no methods.

TABLE 7.2 The Properties of Error Objects

Property	Description
Description	Contains the description of the error that occurred, if a description exists.
HelpContext	Stores the context ID for the VB help file.
HelpFile	Stores the path and filename of the VB help file.
NativeError	Retrieves the database-specific error information for a particular Error object (SQL Server only).
Number	Contains the number of the error that’s set in the Error object. This is the default property for the Err object.
Source	Reflects the system in which this error occurred. Most Access errors set the Source property to MSAccess. When Automation is used, if another application (such as Excel) caused the error, the Source property is set to that application.
SQLState	Shows a five-character error code that the provider returns when an error occurs during the processing of a SQL statement (SQL Server only).

**FIGURE 7.6**

The Errors collection displays multiple errors as they occur in ADO.

Like other Access collections, the Errors collection's index base starts with `Error(0)` and goes to `Error(Error.Count - 1)`.

NOTE

Because the Errors collection deals only with ADO errors, you need to check the `Error(0)` member to see whether it matches the error that shows up in the `Err` object. If it does, you can print all the errors that make up the main error.

On the frmTestErrors form is another button, cmdTestErrorCollection, which intentionally causes an ADO error by looking for a table object that doesn't exist in the Northwind SQL Server back end. The code that causes this error is on the OnClick event in Listing 7.8.

LISTING 7.8 VideoApp(ADO).mdb: Looking for Multiple Errors

```
Private Sub cmdTestErrorCollection_Click()
    Dim cnn As New ADODB.Connection
    Dim rstTest As New ADODB.Recordset

    On Error GoTo Error_cmdTestErrorCollection_Click
    Dim apRoutineError As String
    apRoutineError = "cmdTestErrorCollection_Click"
    cnn.Open "DSN=Northwind;UID=sa;PWD=;"
    rstTest.Open "tblNotExist", cnn, adOpenDynamic, adLockOptimistic, adCmdTable
    Exit Sub

Error_cmdTestErrorCollection_Click:
    apCurrErrNo = Err.Number
    apCurrErrMsg = Err.Description
    ap_ErrorLog apModuleError, apRoutineError, apCurrErrNo, apCurrErrMsg, cnn
    Select Case ap_ErrorHandler(apCurrErrNo, apCurrErrMsg)
        Case apTryAgain
            Resume
        Case apExitRoutine
            Exit Sub
        Case apResumeNext
            Resume Next
    End Select
End Sub
```

The ap_ErrorLog routine checks this error to see whether it's an ADO error. This routine is listed fully in Listing 7.14 later in this chapter, but Listing 7.9 shows the section that handles examining the Errors collection and creating a string of all ADO errors that have occurred.

Listing 7.9 also references a variable named errCurrent. This variable is declared an ADO Error type variable on the line that reads `Dim errCurrent As ADODB.Error`. (This line is shown later, in Listing 7.14, along with the complete code listing.)

LISTING 7.9 VideoApp(ADO).mdb: Logging an Error

```
With rstErrors
    .AddNew
        !ErrModule = strModule
        !ErrFunction = strRoutine
```

LISTING 7.9 Continued

```

!ErrNumber = lngErrorNo
!ErrMsg = strErrorMsg
!ErrForm = strFormName
!ErrControl = strControlName
!UserName = ap_GetUserName()
!ComputerName = ap_GetComputerName()
!Occured = Now
'-- Test for ADO Errors
If cnnCurr.Errors.Count > 0 Then
    '-- Make sure the current error in the Errors
    '-- Collection matches the current error
    If cnnCurr.Errors(0).Number = lngErrorNo Then
        '-- Concatenate the errors into a string
        strTempErrs = ""
        '-- Loop through the Errors collection
        For Each errCurrent In cnnCurr.Errors
            strTempErrs = strTempErrs & Str(errCurrent.Number) & _
                " - " & errCurrent.Description & vbCrLf
        Next errCurrent
        '-- Now put it in the table
        rstErrors!ADOErrors = strTempErrs
    End If
End If
.Update
rstErrors.Close
End With

```

This routine uses the Err object to record most of the information for the main error, starting with the following:

```

'-- Test for ADO Errors
If cnnCurr.Errors.Count > 0 Then

```

These events take place:

1. The Errors collection is refreshed.
2. The routine checks to see whether an Error object is in the ADO Errors collection.
3. It sees whether the first error in the collection (Errors(0)) matches the error found in the Err object. If it does, the routine cycles through all the errors in the Errors collection and concatenates them in a string.
4. The routine places the string in the ADOErrors table field.

The table in Figure 7.7 shows strings in the ADOErrors field for the error that's indeed an ADO error.

ErrNumber	ErrMessage	ADOErrors
11	Division by zero	
11	Division by zero	
-2147217965	[Microsoft][ODBC SQL Server Driver][SQL Server]Invalid object name 'tblNotExist'.	-2147217965 - [Microsoft][ODBC SQL Server Driver][SQL Server]Invalid object name 'tblNotExist'. -2147217965 - [Microsoft][ODBC SQL Server Driver][SQL Server]The cursor was not declared.

FIGURE 7.7
When an ADO error occurs, the ADO Errors collection is parsed and placed in the ADOErrors field.

Creating User-Defined Errors

By using the `CVErr()` and `IsError()` functions and the `Application.AccessError` method, you can create your own functions that can return an error value if an error situation occurs in the function. Here's a short description of each function and method that can be used:

- The `CVErr()` function converts a value to an Error data type, assigning it to a Variant variable.
- The `IsError()` function checks to see whether a Variant is an Error data type.
- The `Application.AccessError` method provides an error description without the error itself.

For example, when you divide two numbers, either the function receives a text value instead of the number needed, or a zero is sent. Listing 7.10 shows the `cmdTestCVErr_Click` routine that calls the `ap_ShowErrorCVErr` subroutine, which passes back either the answer or an error number. Both `cmdTestCVErr_Click` and `ap_ShowErrorCVErr` are located on the `frmTestErrors` form.

LISTING 7.10 VideoApp(ADO).mdb: Controlling Returned Errors

```
Private Sub cmdTestCVErr_Click()
    Dim varRetVal As Variant

    varRetVal = ap_ShowErrorCVErr(InputBox("Please enter the Numerator:", _
        "Numerator"), InputBox("Please enter the Denominator:", "Denominator"))

    If IsError(varRetVal) Then
        '-- User defined number
    End If
End Sub
```

LISTING 7.10 Continued

```

    If CInt(varReturnVal) = 2001 Then
        MsgBox "You should have used numbers!"
    '-- Number of Divide by Zero
    ElseIf CInt(varReturnVal) = 11 Then
        MsgBox Application.AccessError(11)
    End If
Else
    MsgBox "The answer is: " & varReturnVal
End If

End Sub

```

Consider a few items from Listing 7.10:

- You'll want to type the return variable as a `Variant` and convert it to `Integer` to compare the number to error numbers.
- Note the use of the `IsError()` function to test for an `Error` type value.
- You'll want to declare the function as `Variant` as well because it can contain either the answer (in this case, a `double`) or an `Error` value.

Listing 7.11 shows the `ap_ShowErrorCVer()` function, which passes back either an error number for various errors, or the answer itself if everything is okay.

LISTING 7.11 VideoApp(ADO).mdb: Passing Back Errors

```

Function ap_ShowErrorCVer(varNumerator As Variant, varDenominator As _
    Variant) As Variant

    '-- Check and make sure correct data type
    If Not IsNumeric(varNumerator) Or Not IsNumeric(varDenominator) Then
        ap_ShowErrorCVer = CVErr(2001)
    '-- Make sure no zero for denominator
    ElseIf CDbl(varDenominator) = 0 Then
        ap_ShowErrorCVer = CVErr(11)
    '-- If ok, perform action
    Else
        ap_ShowErrorCVer = CDbl(varNumerator) / CDbl(varDenominator)
    End If

End Function

```

From Listing 7.11, you can see that

- If the entered data isn't numeric, a custom error number is assigned to the return value.
- If the denominator is 0, an error is sent back.
- If all is OK, the answer is sent.

You can see from these examples that the `CVErr()` and `IsError()` functions give you that much more control over your application.

Using Custom Error Logs to Track Errors

Users have a hard time remembering what they were doing when an error occurred. By creating an error log, you can track errors as they happen and program to deal with them the next time around.

Before creating an error log, you need to set up a few constants and variables. Both the constants and the error log routine, `ap_ErrorLog`, are found in the `modErrorHandling` module. Listing 7.12 shows the declarations section of this module.

LISTING 7.12 VideoApp(ADO).mdb: Error Log Routine Declarations

```
Option Compare Database
Option Explicit

'-- Constants for how to handle the error.
Public Const apTryAgain = 1
Public Const apExitApplication = 2
Public Const apExitRoutine = 3
Public Const apResumeNext = 4
Public Const apMultiUser = 5

'-- Number of times to retry locks
Public Const apMaxMURetries = 2

'-- Variables to store error information
Public apCurrErrNo As Long
Public apCurrErrMsg As String

'-- Each module needs to have this for the Error Log
Const apModuleError = "ModErrorHandling"

Const apDivisionByZero = 11
''ModErrorHandling
```

These major items are declared:

- The constants used to specify how to handle the error. These are used in the `ap_ErrorHandler` routine (discussed in the next section).
- The number of times to retry locks, used in the `ap_ErrorTryMUAgain()` function.
- The variables to store error information (seen in each error handler). The following code, discussed in the next section, shows the variables in use:

```
Case apTryAgain
    Resume
Case apExitRoutine
    Exit Sub
Case apResumeNext
    Resume Next
```

- The module constant that's passed to the error log. This line declares the constant:

```
Const apModuleError = "modErrorHandling"
```

An Example Error Handler Calling the Error Log

Listing 7.13 shows the , you can track errors as they happen and program to dealcode for the `cmdTestErrorCollection` command button's `OnEvent` event, located on the `frmTestErrors` form.

LISTING 7.13 VideoApp(ADO).mdb: Calling the Error Log Routine

```
Private Sub cmdTestErrorCollection_Click()
    Dim cnn As New ADODB.Connection
    Dim rstTest As New ADODB.Recordset
    On Error GoTo Error_cmdTestErrorCollection_Click
    Dim apRoutineError As String

    apRoutineError = "cmdTestErrorCollection_Click"
    cnn.Open "DSN=Northwind;UID=sa;PWD=;"
    rstTest.Open "tblNotExist", cnn, adOpenDynamic, adLockOptimistic, adCmdTable
    Exit Sub

Error_cmdTestErrorCollection_Click:
    apCurrErrNo = Err.Number
    apCurrErrMsg = Err.Description
    ap_ErrorLog apModuleError, apRoutineError, apCurrErrNo, apCurrErrMsg, cnn
    Select Case ap_ErrorHandler(apCurrErrNo, apCurrErrMsg)
        Case apTryAgain
            Resume
        Case apExitRoutine
            Exit Sub
```

LISTING 7.13 Continued

```
Case apResumeNext
Resume Next
End Select

End Sub
```

The first items in this routine are the lines

```
Dim apRoutineError As String
apRoutineError = "cmdTestErrorCollection_Click"
```

These lines store the name of the current routine in the `apRoutineError` variable, which is passed to the `ap_ErrorLog` error log routine. Before this routine is called, the `Err` object information is stored in variables for future use. This occurs in the next few lines of code, with the `ap_ErrorLog` routine called next:

```
apCurrErrNo = Err.Number
apCurrErrMsg = Err.Description
ap_ErrorLog apModuleError, apRoutineError, apCurrErrNo, apCurrErrMsg, cnn
```

The Actual Error Log Code

Listing 7.14 shows the code for `ap_ErrorLog`, the routine that records the errors. This routine allows you to track the following information for errors: , you can track errors as they happen and program to deal

- The module in which the error occurred. It's useful if you can jump right to the offending module, or perhaps sort the error log by module and then by routine to organize the debugging workload.
- The routine where the error occurred. Again, this helps pinpoint the problem.
- The error number, as retrieved from the `Err` object.

TIP

Storing the error information in variables as soon as you get into an error handler is a good idea. If another error occurs after the first error but before the first error can be handled, you might handle the wrong error.

- The error message as retrieved from the `Err` object.
- The active form at the time the error occurred.

- The active control at the time the error occurred.
- The current user's name, according to Windows.
- The current user's computer name, according to Windows.

NOTE

Two routines are used—one for the computer name and one for the username. These routines, `ap_GetUserName()` and `ap_GetComputerName()`, use Windows API calls and are described in detail in Chapter 15.

- The current date and time when the error occurred, using the `Now` system variable.
- All the Jet errors that occurred, if any.

LISTING 7.14 VideoApp(ADO).mdb: Logging an Error

```
Sub ap_ErrorLog(strModule, strRoutine, lngErrorNo As Long, _
    strErrorMsg As String, Optional cnnCurr As ADODB.Connection)
    Dim cnnError As New ADODB.Connection
    Dim rstErrors As New ADODB.Recordset
    Dim errCurrent As ADODB.Error
    Dim strFormName As String
    Dim strControlName As String
    Dim strTempErrs As String

    '-- Store the name of the active form and control
    strFormName = ""
    strControlName = ""
    On Error Resume Next
    strFormName = Screen.ActiveForm.Name
    strControlName = Screen.ActiveControl.Name
    Err = 0
    '-- Attempt to open the errorlog table on the backend
    cnnError.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & _
        ap_GetDatabaseProp("LastBackEndPath") & ap_GetDatabaseProp("BackEndName")
    '-- If there is a problem, open the local errorlog table
    If Err Then Set cnnError = CurrentProject.Connection
    On Error GoTo Error_ap_ErrorLog
    '-- Log the information
    rstErrors.LockType = adLockOptimistic
    rstErrors.Open "tblErrorLog", cnnError
    With rstErrors
```

LISTING 7.14 Continued

```

.AddNew
!ErrModule = strModule
!ErrFunction = strRoutine
!ErrNumber = lngErrorNo
!ErrMessage = strErrorMsg
!ErrForm = strFormName
!ErrControl = strControlName
!UserName = ap_GetUserName()
!ComputerName = ap_GetComputerName()
!Occured = Now
'-- Test for ADO Errors
If cnnCurr.Errors.Count > 0 Then
'-- Make sure the current error in the Errors
'-- Collection matches the current error
If cnnCurr.Errors(0).Number = lngErrorNo Then
'-- Concatenate the errors into a string
strTempErrs = ""
'-- Loop through the Errors collection
For Each errCurrent In cnnCurr.Errors
strTempErrs = strTempErrs & Str(errCurrent.Number) & _
" - " & errCurrent.Description & vbCrLf
Next errCurrent
'-- Now put it in the table
rstErrors!ADODErrors = strTempErrs
End If
End If
.Update
rstErrors.Close
End With

Exit_ap_ErrorLog:
Exit Sub

Error_ap_ErrorLog:
'-- centralized error handler
apCurrErrNo = Err.Number
apCurrErrMsg = Err.Description
Select Case ap_ErrorHandler(apCurrErrNo, apCurrErrMsg)
Case apTryAgain
Resume
Case apExitApplication
Application.Quit
Case apExitRoutine
Resume Exit_ap_ErrorLog

```

LISTING 7.14 Continued

```
Case apResumeNext
Resume Next
End Select

End Sub
```

You’ve already examined parts of this code. Notice the use of `Screen.ActiveForm` and `Screen.ActiveControl` to get the current form and control information for the error log.

Another interesting feature of this routine is that it tries to log the error to the back end; if that’s not possible, it logs the error to the front end (the local application). For more information on front and back ends, see Chapter 21, “Handling Multiuser Situations.”

Logging to the Back End First, or Front End if Necessary

This code allows for the possibility that the back end is causing the error:

```
'-- Attempt to open the errorlog table on the backend
cnnError.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & _
    ap_GetDatabaseProp("LastBackEndPath") & ap_GetDatabaseProp("BackEndName")
'-- If there is a problem, open the local errorlog table
If Err Then Set cnnError = CurrentProject.Connection
```

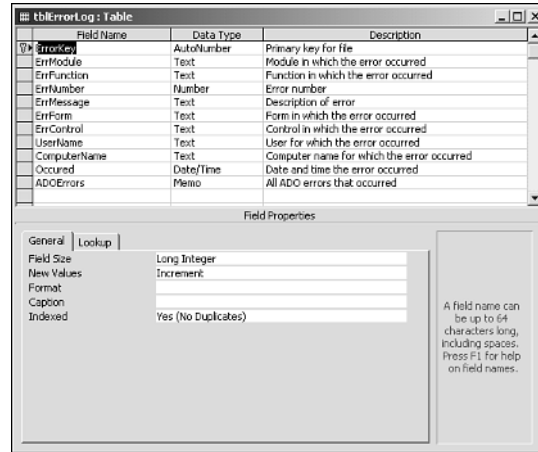
Following an `On Error Resume Next` statement, the first line tries to open a connection to the back end by using the path and filename stored in the custom database properties `LastBackEndPath` and `BackEndName`. If an error occurs, the ADO connection variable `cnnError` is set to the current project’s connection.

The back end and front end have the same table, named `tblErrorLog`. Figure 7.8 shows the Design view of `tblErrorLog`.

Updating the Back End with Any Errors That Occur

Constantly looking at all individual workstations to see whether any errors have occurred would be a pain. To remedy this situation, you can place the `ap_ErrorCheckLocal` routine in the system startup routine. Listing 7.15 shows the code for `ap_ErrorCheckLocal`, which does the following:

- Checks to see whether any errors are in the local `tblErrorLog` table
- If there are any errors, copies the errors out to the back end
- Deletes the errors that were in the front end



Field Name	Data Type	Description
ErrKey	AutoNumber	Primary key for file
ErrModule	Text	Module in which the error occurred
ErrFunction	Text	Function in which the error occurred
ErrNumber	Number	Error number
ErrMessage	Text	Description of error
ErrForm	Text	Form in which the error occurred
ErrControl	Text	Control in which the error occurred
UserName	Text	User for which the error occurred
ComputerName	Text	Computer name for which the error occurred
Occured	Date/Time	Date and time the error occurred
ADOErrors	Memo	All ADO errors that occurred

Field Properties

General | Lookup

Field Size: Long Integer
 New Values: Increment
 Format:
 Caption:
 Indexed: Yes (No Duplicates)

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

FIGURE 7.8

The structure of the `tblErrorLog` table is identical in the front-end and back-end databases.

LISTING 7.15 VideoApp(ADO).mdb: Checking the Local Error Log

```
Sub ap_ErrorCheckLocal()
    Dim rsLocalErrs As New ADODB.Recordset
    Dim intAreErrors As Integer
    '-- Variable for recording the name of the routine
    Dim apRoutineError As String
    apRoutineError = "ap_ErrorCheckLocal"
    On Error GoTo Error_ap_ErrorCheckLocal
    '-- Open the Local version of the Error Log and test for records
    rsLocalErrs.Open "tblErrorLog", CurrentProject.Connection
    intAreErrors = rsLocalErrs.RecordCount
    rsLocalErrs.Close

    '-- If error entries exist copy them to the backend
    ' and delete local version
    If intAreErrors Then
        DoCmd.Echo True, "Copying Local Errors into backend..."
        CurrentProject.Connection.Execute "INSERT INTO tblErrorLog IN '" & _
            ap_GetDatabaseProp("LastBackEndPath") & _
            ap_GetDatabaseProp("BackEndName") & _
            "' SELECT ErrModule, ErrFunction, ErrNumber, ErrMessage, " & _
            "ErrForm, ErrControl, UserName, ComputerName, Occured, " & _
            "ADOErrors FROM tblErrorLog;"
    End If
End Sub
```

LISTING 7.15 Continued

```

        CurrentProject.Connection.Execute "DELETE * FROM tblErrorLog"
        DoCmd.Echo True
    End If

    Exit Sub

Error_ap_ErrorCheckLocal:
'-- Store Error information into variables for future use.
apCurrErrNo = Err.Number
apCurrErrMsg = Err.Description
ap_ErrorLog apModuleError, apRoutineError, apCurrErrNo, apCurrErrMsg, _
    CurrentProject.Connection
Select Case ap_ErrorHandler(apCurrErrNo, apCurrErrMsg)
    Case apTryAgain
        Resume
    Case apExitRoutine
        Exit Sub
    Case apResumeNext
        Resume Next
End Select

End Sub

```

This routine rounds out the error-logging routines. Again, you can create reports to examine errors and fix them. Figure 7.9 shows such a report.

Module	Function	Err No.	ErrorMessage	Form	Control
Form_TestErrors	cmdTestErrorColla	-2147217865	Microsoft[ODBC SQL Server Driver][SQL Server]Invalid object name 'tblNoExist'.	frmTestErrors	cmdTestErrorCollection
	cmdTestRaiseErro	11	Division by zero	frmTestErrors	cmdTestRaiseError
	cmdTestRaiseErro	11	Division by zero	frmTestErrors	cmdTestRaiseError

Page: 14 | 1

FIGURE 7.9

You can create various reports to track errors so that you can fix them.

Creating a Centralized Error-Handling Routine

In addition to logging errors, it's useful to come up with a standard way to handle errors. A centralized error handler is an ever-changing animal. As you come across new errors, you'll want to handle them consistently.

In the meantime, you'll also want to handle most errors without having to create custom error handlers for each. A centralized error handler is just the ticket. With a centralized error handler, you can have Access handle various errors the same way in all your routines, just by programming for them in one location.

NOTE

Programming your routines is just the start of what it takes to come up with a good, robust, centralized error handler. You need to customize this handler for your individual applications. The routines will help you, as the developer, get on the right track for developing your own handler that's right for your applications.

To start, it's a good idea to look again at the declarations section of the `modErrorHandler` module (from Listing 7.12), where the error-handling routines reside. The following statements are pertinent:

```
'-- Constants for how to handle the error.  
Public Const apTryAgain = 1  
Public Const apExitApplication = 2  
Public Const apExitRoutine = 3  
Public Const apResumeNext = 4  
Public Const apMultiUser = 5
```

These constants correspond to values assigned to possible errors. Table 7.3 shows the five constants and describes their purposes.

TABLE 7.3 Error Values

<i>Error Number</i>	<i>Meaning</i>
1	Retry; performs a Resume immediately
2	Terminate App; immediately closes the application
3	Terminate Function; like Terminate App but simply exits the executing function

TABLE 7.3 Continued

Error Number	Meaning
4	Resume Next; continues with the next line of code
5	Multiuser; treats the error as a multiuser error, calling a special function named <code>ap_ErrorTryMUAgain()</code> , which displays a message while it retries the command

These errors are stored in the table `tblErrorInfo`, whose structure you can see in Figure 7.10. Figure 7.11 shows some of the errors stored in `tblErrorInfo`, including the `HowToHandle` field.

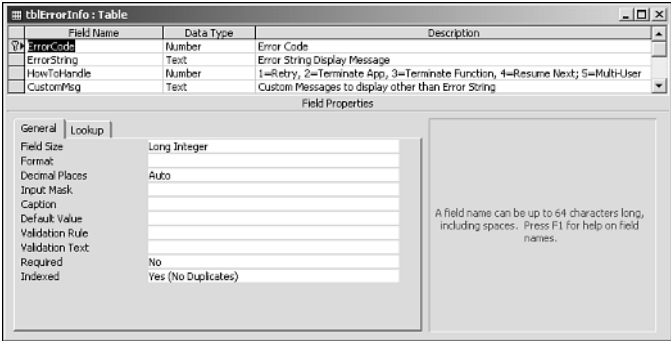


FIGURE 7.10

This structure for the `tblErrorInfo` table allows for custom error messages.

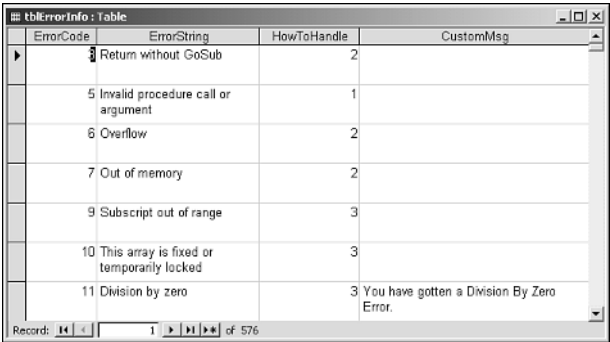


FIGURE 7.11

You can let Access know how to handle errors automatically.

In the code for `ap_ErrorCheckLocal` (refer to Listing 7.15), you saw a sample of how to call the centralized error handler, `ap_ErrorHandler`. The code in Listing 7.16 performs the actual call to the code that examines the error.

LISTING 7.16 VideoApp(ADO).mdb: Calling the Centralized Error Handler

```
Error_ap_ErrorCheckLocal:
'-- Store Error information into variables for future use.
apCurrErrNo = Err.Number
apCurrErrMsg = Err.Description
ap_ErrorLog apModuleError, apRoutineError, apCurrErrNo, apCurrErrMsg, _
    CurrentProject.Connection
Select Case ap_ErrorHandler(apCurrErrNo, apCurrErrMsg)
    Case apTryAgain
        Resume
    Case apExitRoutine
        Exit Sub
    Case apResumeNext
        Resume Next
End Select
```

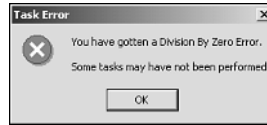
As you can see from Listing 7.16, the `ap_ErrorHandler()` function is called with the number and description of the offending error passed as arguments. You can also see that some of the constants declared to handle the error are used to respond correctly, depending on the error.

NOTE

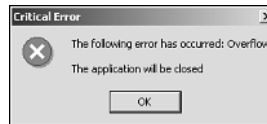
The constants `apQuitApplication` and `apMultiUser` weren't included in Listing 7.16 because both are used in `ap_ErrorHandler` itself.

Listing 7.17 shows the code for the `ap_ErrorHandler` function. But before looking at the code, note some of its features:

- The error is looked up in the `tblErrorInfo` table.
- If a custom message exists in the `CustomMsg` field or a message is displayed, it uses the custom message (see Figure 7.12 for an example).
- If the error's `HowToHandle` field equals `apExitApplication`, the user sees a message saying which error has occurred and that the application will be closed (see Figure 7.13).

**FIGURE 7.12**

Display any custom message you want for an error.

**FIGURE 7.13**

Let users know that the application will close.

- If the `apExitRoutine` constant is needed, a message box appears to tell users that the application will be closed (refer to Figure 7.13).
- If a multiuser error has occurred, special multiuser functions are called to handle the retrying of locks and such. Chapter 21 covers multiuser errors in detail and how to handle them.

LISTING 7.17 VideoApp(ADO).mdb: The Centralized Error Routine Itself

```
Function ap_ErrorHandler lngCurrErrNo, strOrigErrMsg) As Integer
    Dim catCurr As New ADOX.Catalog
    Dim cmdCurrErr As New ADODB.Command
    Dim rstCurrErr As New ADODB.Recordset
    Dim intHowTohandle As Integer
    Dim strErrMsg As String

    On Error GoTo Error_ap_ErrorHandler
    lngCurrErrNo = CLng(lngCurrErrNo)
    '-- Get the entry in ErrorInfo for the current error occurring.
    catCurr.ActiveConnection = CurrentProject.Connection
    Set cmdCurrErr = catCurr.Procedures("qryCurrError").Command
    cmdCurrErr.Parameters("CurrError") = lngCurrErrNo
    rstCurrErr.Open cmdCurrErr, , adOpenForwardOnly, _
        adLockReadOnly, adCmdStoredProc
    If Not (rstCurrErr.EOF And rstCurrErr.EOF) Then
        '-- If a custom message has been created, use it.
        strErrMsg = IIf(IsNull(rstCurrErr!CustomMsg), _
            "The following error has occurred: " & strOrigErrMsg, _
            rstCurrErr!CustomMsg)
```

LISTING 7.17 Continued

```

'-- Store how to handle the error to a variable
intHowToHandle = rstCurrErr!HowToHandle
'-- Check to see how this error is to be handled
Select Case intHowToHandle
    Case apExitApplication
        '-- if quit the application, give a message saying so, then quit.
        strErrorMsg = strErrorMsg & vbCrLf & vbCrLf & _
            "The application will be closed"
        Beep
        MsgBox strErrorMsg, vbCritical, "Critical Error"
        Application.Quit
    Case apExitRoutine
        '-- If exiting the routine, give a message saying
        '-- the task has been stopped.
        strErrorMsg = strErrorMsg & vbCrLf & vbCrLf & _
            "Some tasks may have not been performed"
        Beep
        MsgBox strErrorMsg, vbCritical, "Task Error"
    Case apMultiUser
        '-- For multi-user errors try a couple of times, then check with user.
        If ap_ErrorTryMUAgain(strOrigErrorMsg) Then
            '-- If user wants, try again
            intHowToHandle = apTryAgain
        Else
            '-- If struck out, exit the routine
            intHowToHandle = apExitRoutine
        End If
    Case Else
        '-- If exiting the routine, give a message saying
        '-- the task has been stopped.
        strErrorMsg = strErrorMsg & vbCrLf & vbCrLf & _
            "Some tasks may have not been performed. " & _
            "Note this error has not been categorized."
        Beep
        MsgBox strErrorMsg, vbCritical, "Task Error"
        '-- If struck out, exit the routine
        intHowToHandle = apExitRoutine
    End Select
Else
    Beep
    MsgBox strOrigErrorMsg
    '-- if unknown error, then exit the function
    intHowToHandle = apExitRoutine
End If

```

LISTING 7.17 Continued

```
Exit_ap_ErrorHandler:
    '-- Pass back how to handle the error
    ap_ErrorHandler = intHowTohandle
    Exit Function

Error_ap_ErrorHandler:
    '-- Create a simple error handler for this routine
    MsgBox "An Error occurred in the error handler", vbCritical, _
        "Error Handler Problem"
    intHowTohandle = apExitRoutine
    Resume Exit_ap_ErrorHandler

End Function
```

Again, this centralized error-handling routine is a great starting point for you to create a custom and complete error handler of your own.

A Last Look at Error-Handling Issues

The following sections cover some of the issues to watch for when creating error handlers for your routine. Watching for issues such as environment changes and knowing how to handle transactions correctly can save you from many hassles later.

Watching for Environment Changes

When setting Echo off, HourGlass on, and SetWarnings off in your routines, be sure to toggle them back to the way you want them in your error handler. These *environment settings* affect what happens in the application's environment. This is a good reason for using a label to exit your routine and placing these commands in the code to toggle the settings as necessary. Listing 7.18 shows an example.

LISTING 7.18 VideoApp(ADO).mdb: Resetting the Environment Settings

```
Sub ShowErrorCleanup()
    On Error GoTo Error_ShowErrorCleanup
    DoCmd.Hourglass True
    DoCmd.Echo False, "Displaying Message"
    DoCmd.SetWarnings False
    DoCmd.DeleteObject acTable, "NoTable"

    '-- This label starts the cleanup and exit section
Exit_ShowErrorCleanup:
```

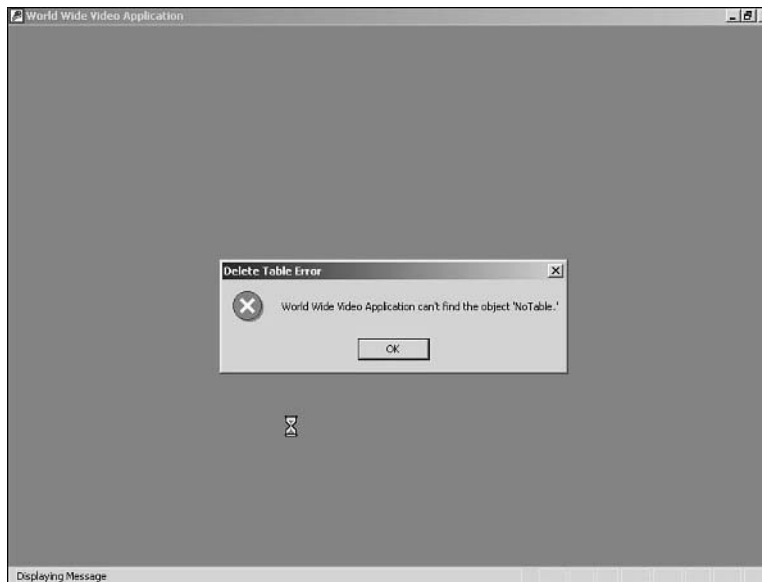
LISTING 7.18 Continued

```
DoCmd.Hourglass False
DoCmd.Echo True
DoCmd.SetWarnings True
Exit Sub

'-- This label is for the error handler
Error_ShowErrorCleanUp:
    MsgBox Err.Description, vbCritical, "Delete Table Error"
    Resume Exit_ShowErrorCleanUp

End Sub
```

The problem with this way of switching back the environment variables is that they aren't changed until after the error message box appears. This can confuse users when the cursor is still shown as an hourglass and not a pointer to the message box. Figure 7.14 shows this problem.

**FIGURE 7.14**

When the hourglass cursor is still shown after an error occurs, users might become confused.

To solve the problem, repeat the commands in the error handler and then just exit the routine, so you don't perform the commands twice. Listing 7.19 shows this method.

LISTING 7.19 VideoApp(ADO).mdb: Preventing the Resetting of the Environment Changes

```
Sub ShowErrorCleanup2()  
    On Error GoTo Error_ShowErrorCleanup2  
    DoCmd.Hourglass True  
    DoCmd.Echo False, "Displaying Message"  
    DoCmd.SetWarnings False  
    DoCmd.DeleteObject acTable, "NoTable"  
    DoCmd.Hourglass False  
    DoCmd.Echo True  
    DoCmd.SetWarnings True  
  
    Exit Sub  
  
'-- This label is for the error handler  
Error_ShowErrorCleanup2:  
    DoCmd.Hourglass False  
    DoCmd.Echo True  
    DoCmd.SetWarnings True  
    MsgBox Err.Description, vbCritical, "Delete Table Error"  
    Exit Sub  
  
End Sub
```

Notice the cleanup commands in the error handler itself. The Exit label is unnecessary here.

TIP

As shown earlier in Listing 7.18 with the Resume statement, using routine exiting code with labels is good programming practice, although it's not required.

Using Your Error Handler to Roll Back Transactions

You can do transaction processing by using VBA. With transaction processing, you can wrap multiple tasks into one transaction. If one task fails, the whole transaction can be rolled back, and all the tables involved can be restored to where they were before any tasks were begun. This is where error handlers come in. You'll most likely use the error handler to roll back transactions.

Listing 7.20 shows that even though the error occurred after the deletion of records, the transaction is rolled back and the records are still in the table. The `Err.Raise` method is used intentionally to cause an error.

LISTING 7.20 VideoApp(ADO).mdb: Rolling Back the Transaction on an Error

```
Sub ShowErrorRollback()  
    Dim cnnLocal As New ADODB.Connection  
    On Error GoTo Error_ShowRollback  
    Set cnnLocal = CurrentProject.Connection  
    cnnLocal.BeginTrans  
    cnnLocal.Execute "Delete * From RollbackExample"  
    Err.Raise 11  
    cnnLocal.CommitTrans  
  
    Exit Sub  
  
'-- This label is for the error handler  
Error_ShowRollback:  
    MsgBox Err.Description, vbCritical, _  
        "Delete Records Error, Transaction Rolled Back"  
    cnnLocal.RollbackTrans  
    Exit Sub  
  
End Sub
```

CAUTION

You must have a `RollBackTrans` or `CommitTrans` in your code! If you don't complete the transaction, using one command or the other keeps the recordset open—and locked—until you quit the application (even if you closed the recordset or have the recordset variable declared as `Private`).

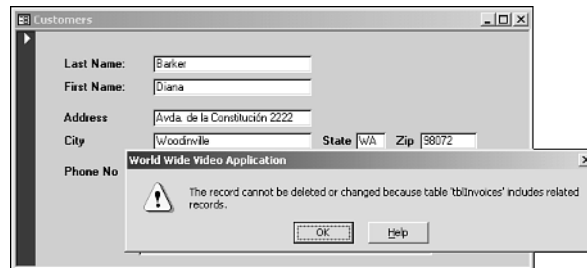
Transaction processing is straightforward to use, and very powerful; however, you need to have error handling involved to make it robust as well.

Using a Form's On Error Event

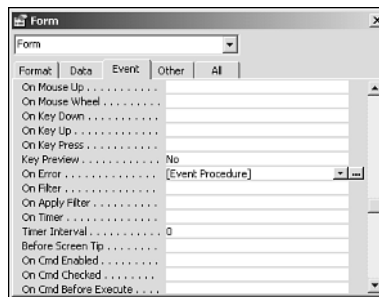
In VBA, you can use error handlers to trap errors. But what about when users are in a form and try to delete a customer that has invoices? If you have set up referential integrity correctly, the user sees a dialog similar to Figure 7.15.

Users sometimes need a more understandable error message than what Access and Jet provide. Access gives you a method for rewording or handling errors on forms and reports with the Form/Report error handler.

Figure 7.16 shows the property sheet for the `frmCustomers` form in `VideoApp(ADO).mdb`. You can see the `On Error` event listed here.

**FIGURE 7.15**

The default error message appears when users try to delete records with referential integrity set.

**FIGURE 7.16**

The On Error event is available on forms and reports.

By using the On Error event, you can catch and change the referential integrity error shown in Figure 7.15. Listing 7.21 shows the code used to perform this task, assigned to On Error.

LISTING 7.21 VideoApp(ADO).mdb: Handling Form Errors

```
Private Sub Form_Error(DataErr As Integer, Response As Integer)
    If DataErr = 3200 Then
        ' - Can't delete because of Referential Integrity
        Beep
        MsgBox "Sorry, but customers with invoices can't be deleted."
        Response = acDataErrContinue
    End If
End Sub
```

The On Error event uses two parameters:

- DataErr is the error number.
- Response tells Access how to continue with the error. acDataErrContinue doesn't display an error and continues handling the error, whereas acDataErrDisplay displays the error.

NOTE

Although you can display new messages and clean up errors, you can't cancel an error that occurred with the Form/Report error handler.

The code used in Listing 7.21 creates the new message box, shown in Figure 7.17.

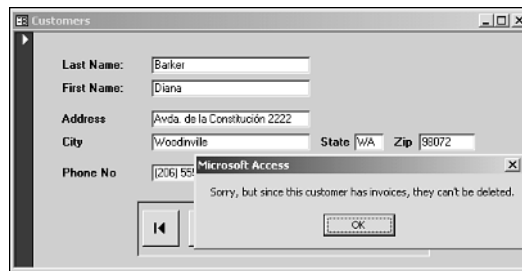


FIGURE 7.17

This message is a little easier for users to understand.

TIP

Have the Form/Report error handler look up the error inside the tblErrorInfo table to use the custom messages found there. When you're trying to give detailed custom messages, looking up generic errors for a whole application can be difficult.

Nesting Error Handlers

You can nest error handlers in such a way that you can just have a main handler per each chain of routines that's called. Assume that you have two routines—Routine A calls Routine B. Here are three scenarios:

- Routine A has an error handler; Routine B doesn't. If an error occurs in Routine B, it uses Routine A's error handler.
- Routine A has an error handler; so does Routine B. If an error occurs in Routine B, its own error handler takes precedence.
- If neither has an error handler, Access's default error handling takes over.

Looking at Some New Options for Error Handling

Some VBA options affect how code breaks when errors occur. You can find these options on the General page of the Options dialog (in the VBE, choose Options from the Tools menu). In the middle right side of the page are three options in an Error Trapping section (see Figure 7.18):

- Break on All Errors breaks on all errors, whether or not custom error handling is invoked. This is good when you're debugging your application, but not when you distribute it.
- Break in Class Module breaks only in a class module (discussed in Chapter 2, "Coding in Access 2002 with VBA"), unless you've placed error handling in them.
- Break on Unhandled Errors means that errors not handled by you with the `On Error` statement break whether or not they're in a class module or public module.

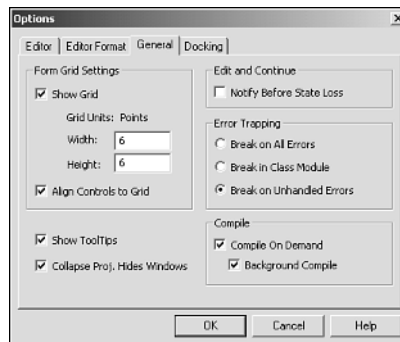


FIGURE 7.18

Use these options to tell Access how you want errors handled during runtime.

Summary

Access provides a number of commands for controlling error handling. With very little code, you can have Access log all the errors that occur during runtime. You can also create centralized error-handling routines to anticipate errors and handle them accordingly.

- Chapter 4, “Working with Access Collections and Objects,” details the various collections and objects found within Access.
- Chapter 25, “Startup Checking System Routines Using ADO,” uses various error-handling routines to check the back end at startup.
- Appendix C, “Working with Data Access Objects,” provides a background on DAO.
- Appendix E, “Access 2002 and Jet 4 Errors,” lists the errors for Access 2002 and Jet 4.

You can find both appendixes at www.sampublishing.com; simply type this book’s ISBN in the Search field.

Manipulating and Presenting Data

PART



IN THIS PART

- 8 Using Queries to Get the Most Out of Your Data 187
- 9 Creating Powerful Forms 241
- 10 Expanding the Power of Your Forms with Controls 265
- 11 Creating Powerful Reports 313
- 12 Working with Data Access Pages 355

Using Queries to Get the Most Out of Your Data

CHAPTER

8

IN THIS CHAPTER

- Understanding Where Queries Are Used in Access 188
- Working with Select Queries 193
- Working with Action Queries 199
- Performing Advanced Query Operations 201
- Adding More Power with VBA 205
- Driving Reports and Forms with Queries 217
- Solving Problems with Queries 217
- Examining the Architecture of the Query Resolution Process 227
- Optimizing Queries with Jet 229
- Understanding Optimization Techniques 233
- Using Unconventional Optimization Techniques 236
- Using the Analyzer Wizards 238
- Looking at Access 2002's New Query Features 239

Queries can make you a hero or get you killed. To come out the hero, you need to consider how you use them. For example, you have some code that runs through various recordsets and examines pieces of data that will be used later. Because this process is being performed on a million-record table, it takes 20 minutes to run. But if you come up with an Access query that takes only five minutes to run because of Jet's optimizations, you're then crowned king.

This chapter looks into some things to consider when dealing with queries. To start, it's a good idea to find out just where queries are used with Access. Doing so will help you to know when and where to take advantage of the queries.

NOTE

Although this chapter covers queries, which are used in the standard Access MDB, when using an ADP you will use views, stored procedures, and SQL functions (SQL Server 2000) instead. For more on these objects, see Chapter 23, "Moving Workgroup Applications to Client/Server," and Chapter 24, "Developing SQL Server Projects Using ADPs."

Understanding Where Queries Are Used in Access

Access queries come in three basic groups:

- Select queries, which retrieve information from one or more tables or from other select queries, including totals and crosstab queries
- Action queries, which affect data in tables
- Pass-through queries, which communicate with a SQL back-end database

This chapter covers select and action queries. Table 8.1 defines where you can use these queries in Access.

TABLE 8.1 Where Queries Are Used in Access 2002

<i>Select Queries</i>	<i>Action Queries</i>
Form record source	Macro OpenQuery action
Form filters	DoCmd OpenQuery in VBA
Report record source	Directly from the database container
Source for queries	ADO Connection.Execute
Data Access Pages	
Subform source objects	
Input to other queries	

You can use select queries in most of the same places that you can use a base table. A select query provides a view of the data from one or more tables that varies from the way the data actually appears in the table. The data might be shown alphabetically in the query, even though the underlying table stores the data in the order it was entered into the table.

Using Queries with Form and Report Record Source Properties

A form or report's Record Source property can specify a query as a SQL statement or a saved query. Using each option has advantages and disadvantages. Some developers prefer to use SQL statements rather than saved queries, so you can treat the form or report as a container and copy around the record source without worrying about losing the associated saved query. Figure 8.1 shows the form property sheet with a SQL statement as the source. By using this approach, you can edit the record source without accidentally affecting the source of another form or report.

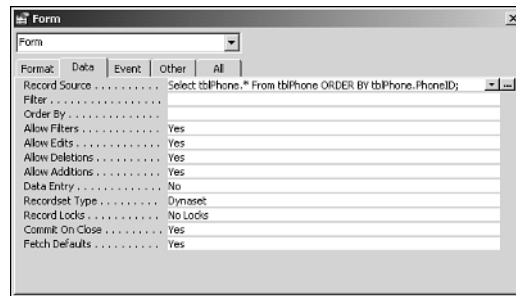


FIGURE 8.1

Using a SQL statement for a record source saves having to keep track of another object (the query).

Figure 8.2 shows the same record source stored as a saved query. If the speed of the form load is critical, store the SELECT statement as a saved query and be careful to follow a naming convention so that the query can easily be associated with the form or report it applies to.

Access compiles a SQL statement before storing it as a saved query, so part of the work required to solve the query is already done. On the other hand, SQL statements in a record source must be checked for syntax and then compiled before the query engine can begin to determine how to resolve the query. See the later section “Examining the Architecture of the Query Resolution Process” for more information on the query resolution process.

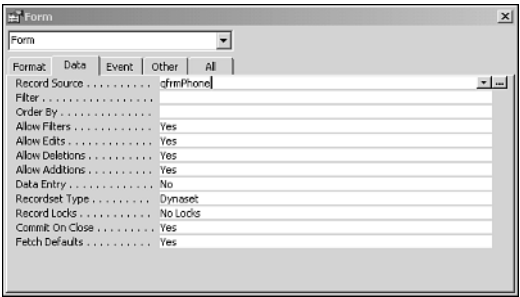


FIGURE 8.2
Use query objects instead of SQL statements when the query is complicated.

Giving Users Access to Queries

When developing end users, you might want to expose some queries directly to them. With Access security, you can allow users to run specific queries directly from the database container and restrict access to other queries that shouldn't be run. Although controlling which queries users have access to with security is quite effective, you might need to provide more information to users before they can run a query.

To show how to expose select and action queries to users, the following query form example, `zsfrmQuery`, allows new queries to be easily exposed to users as the system grows. In this example, the `zstblQuery` table tracks a short name that users can remember and a long description of what the query will do. Figure 8.3 shows the `zstblQuery` table; Figure 8.4 shows the long description that appears when users click a query.

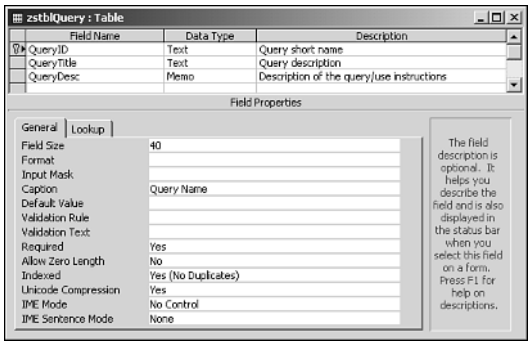
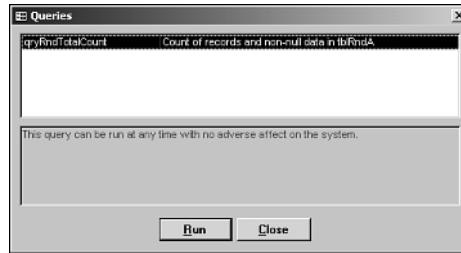


FIGURE 8.3
The `zstblQuery` table stores information on queries and lets users run queries themselves.

**FIGURE 8.4**

Providing users with an interface gives them the power to safely run any of the queries they're allowed to run.

Users are presented with a form (zsfrmQuery) containing a list box that displays the queries from zstblQuery. The `OnClick` event for `lstQuery` shows the description of the query so that users can scroll through each query. They can also read the information that describes the query's purpose and when it should be run. When the table is in place, you can create and expose as many queries as needed to users without giving those users access to the database container.

TIP

To make this application more user-friendly, you can add other information to this table, such as text that will appear in a confirmation prompt when users click the Run button.

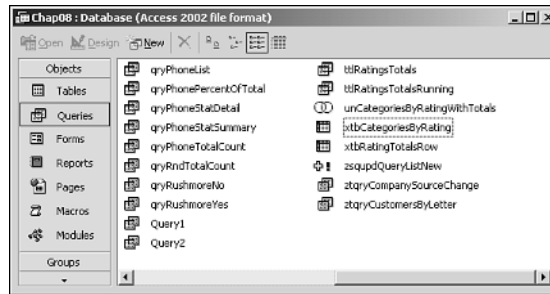
8

USING
QUERIES TO
GET THE MOST OUT
OF YOUR DATA

Using Naming Conventions and Query Documentation

One of the biggest messes developers can create is in the database container's query list. New business opportunities and concerns drive the need for new reporting and data analysis from the data collected in database tables. Developing a naming convention and documenting the purpose of each query help keep the mess under control.

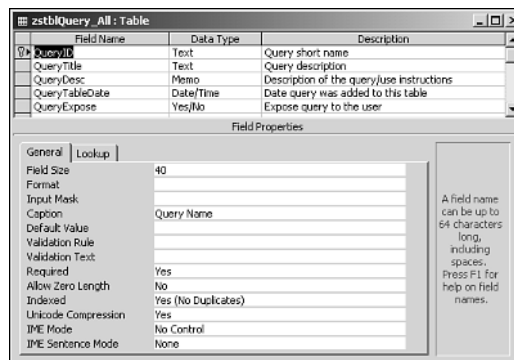
What developer hasn't received a call from a customer, "Hey, how many customers do we have in [wherever] now?" After a couple of clicks and dragging some fields, the answer is onscreen, and after some cleanup the query has the other five columns the customer needed. Tell him the answer and throw the query away, right? Not quite. You save the work in case the customer calls again tomorrow with the same question. Figure 8.5 shows the result of saving this query without using a specific convention for naming queries. Notice that a couple of queries are named simply `Query1` and `Query2`. Two weeks later, you try to remember what `Query2` was for. Was it part of a form or report in the system, or just an *ad hoc* query run for the customer? This example points to the need for naming standards and query documentation.



By not using naming conventions, you create a mess of meaningless names.

Query definitions have a 255-character Description property that you can use to document a query's purpose. Combine this with a good naming convention such as LNC, and you have a good beginning (for LNC level 2 conventions, see Appendix F, "Leszynski Naming Conventions for Microsoft Access" on this book's Web page at www.samspublishing.com). The naming conventions help convey the query's purpose, especially when it's tied to a specific form or report. For example, `qfrmAddr` would be the query used as a record source for `frmAddr`. Although query names can be up to 64 characters long, using that many characters isn't recommended.

Thanks to the number of queries in a database and the infrequent use of some queries, going into a more detailed description might be preferred. This is especially true when multiple summary queries are assembled into a final summary or union query that feeds a report. Figure 8.6 shows a modified version of `zstblQuery`, called `zstblQuery All`, with a new date field added.



Adding the QueryTableDate field to the zstblQuery table allows you to keep track of when the query was added to the table.

Notice the addition of the query table date and the query expose flag. The table date is included to track the date the record was added to the table. Viewing the last date in the existing records provides a good date to reference when looking for changes in `MSysObjects`, a table that Access uses to keep track of information about objects such as queries and reports.

NOTE

The query expose flag is included so that all queries can be stored in one table and documented in one place. With the exposed flag, you can add query description information in the standard `zstblQuery` table and have the query form filter out queries that shouldn't be exposed to users.

The query `zsqupdQueryListNew` updates this table with all new queries in the system. Run the query once or twice a day when developing a project or when the database is closed for the session. Documenting the queries and using standard naming conventions helps reduce the amount of time and energy spent combing through the query list in the database container.

Working with Select Queries

Access developers must learn the query design grid and learn it well. After you learn it, it becomes a valuable tool with which you can build applications quickly and get a jump on the development process.

The query design grid in Query Design view is the fastest way to get a query going, and to see that you're selecting and sorting on the right fields. However, Query Design view also has the SQL view window for accessing the SQL statement directly, as well as the datasheet view for testing.

For select queries, you can update data right in the grid without programming.

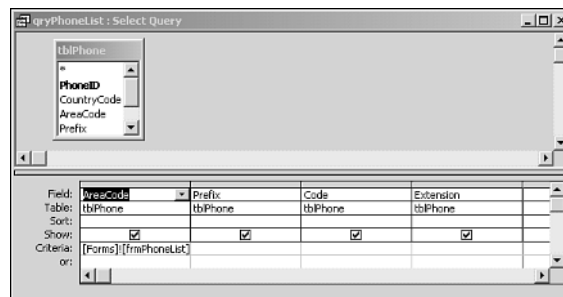
Folks who say, "Real programmers don't use the query design grid. They use SQL," have way too much time on their hands. The query design grid is the fastest and most intuitive way to build queries against complicated data structures. In less than a minute, you can build a complicated query with many tables and many output fields. A few more seconds is time enough to choose Send from the File menu, and off go the results to a customer or manager. Okay, so using Send isn't quite query by example, but it does provide a way to use the information created with the query design grid.

NOTE

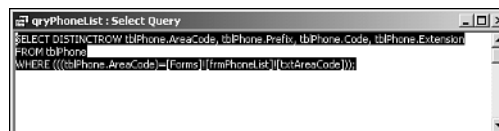
You can do certain things only in SQL view. Union queries and the SELECT DISTINCTROW statement sometimes need to be manually added, as opposed to the general SELECT you get from designing a Query using the design grid. SQL view is also useful if by mistake you renamed your tables and broke the query.

Generating SQL statements in the query design grid prevents misspelled table and field names. It also keeps you from having to deal with entering SQL keywords or parentheses. VBA provides no syntax checking on SQL statements, so the query design grid is a great way to develop SQL statements before moving the statements to VBA.

To get the most out of the query design grid, use it for doing *ad hoc* mass cleanup of data, viewing and changing individual records, and generating VBA code. Most queries are easy to build in the grid. You also can move them to VBA by choosing SQL from the View menu, and then copying the SQL SELECT statement displayed in the window. Figure 8.7 shows a query using the query design grid; Figure 8.8 shows the same query in SQL view.

**FIGURE 8.7**

Using the query design grid is a good way to get used to the Access query design tool.

**FIGURE 8.8**

To move this SQL statement to VBA, copy and paste it into the appropriate routine.

To create the code in VBA, use the code-generating `zsfrmSQLVBA` form (available in `Chap08.mdb` on this book's Web page at www.sampublishing.com). Copy the SQL statement from the SQL view window to the code generator form. Click **Generate**, and you have code that you can paste into VBA (see Figure 8.9).

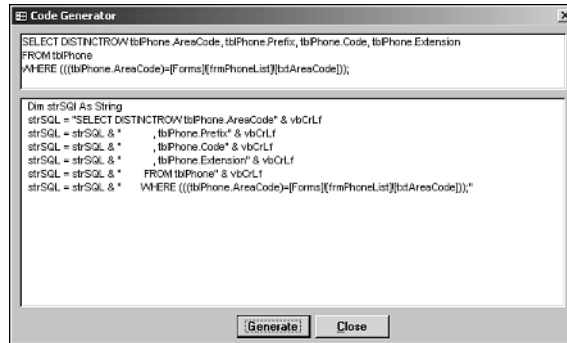


FIGURE 8.9

This form generates the VBA code necessary for a given SELECT statement.

The query code generator lets you build SQL statements that you can paste right into your code. Now you can replace the criteria in the **WHERE** clauses with more complicated values or values derived from formulas. With the generator, you can take advantage of the query design grid for most of your work and then code the rest in VBA.

Joining Tables

Defining the relationships between tables in the query design grid makes the process of building the SQL much faster than coding it from scratch. Left and right joins allow you to select all records from one table and only the records from another table where the joined fields match. An inner join includes only records that exist in both tables used in the join.

Access's online help provides specific information on the capabilities of the different join operations, but a few caveats are worth noting:

- When you create inner joins in which one table has multiple records and another has a single matching record, the single record will be repeated for each record on the multiple side. For example, joining `Customers` to `Invoices` repeats the customer record for each invoice.
- When you can't fill in the join field on one side of the relationship, consider using a left or right join to ensure that the record is still included in the resultset, even though the joined field is null. An example of an outer join would be if you wanted to see all

customers with no invoices. You could do so by setting the criteria of an invoice number equal to Null. Because the customer record is displayed regardless, the criteria would limit the recordset to only those customers with no invoices.

- When joining large numbers of records, consider performance issues. See the performance sections later in this chapter, starting with “Understanding Optimization Techniques.”

Using the Same Table Twice (Self Joins)

Developers run into many situations that require including the same table in a query multiple times. Figure 8.10 shows a scenario in which the phone number table is used for companies and employees. To get company and employee phone numbers, the phone table needs to be joined into the query twice.

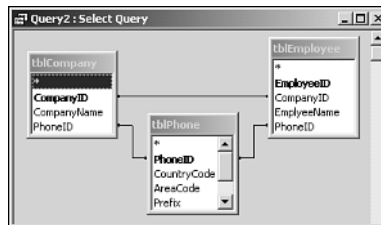


FIGURE 8.10

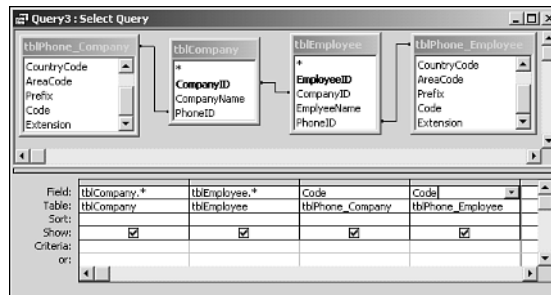
Here's a case of relating a table, tblPhone, to two different tables.

To obtain a list of all companies with employees, you need to proceed as follows by using the Chap08.mdb database (on this book's Web page at www.samspublishing.com):

1. At the Query tab of the database container, click the New button to the right of the query list, and then click OK in the New Query dialog. A new query opens in the query design grid, without using any wizards. The Show Table dialog then appears.
2. Select the **tblCompany** and **tblEmployee** tables and click Add. The relationship is already set, so all companies are shown with the list of their employees.
3. Choose Show Table from the Query menu to add the **tblPhone** table to the grid. The relationship to **tblCompany** and **tblEmployee** will both be added for **tblPhone** automatically.
4. Click the join line from **tblEmployee** to **tblPhone** and then press Delete.
5. If the Table Names line isn't already showing in the grid below, choose Table Names from the View menu to view the Table Names line of the grid.

6. From the Query menu, choose Show Table and add another copy of tblPhone to the grid. The new table appears as tblPhone_1.
7. To ensure that the tables are used properly and documented correctly, choose SQL from the View menu. Next, replace the tblPhone_1 reference in the SQL statement with something more meaningful—for this example, tblPhone_Employee. Do this for all tblPhone_1 references.
8. Change the tblPhone reference so that it includes an AS clause to rename tblPhone as tblPhone_Company. The resulting SQL statement should look like this:


```
SELECT tblEmployee.*, tblCompany.*, tblPhone_Company.*,
tblPhone_Employee.* FROM ((tblCompany INNER JOIN tblPhone
as tblPhone_Company ON tblCompany.PhoneID =
tblPhone_Company.PhoneID) INNER JOIN tblEmployee
ON tblCompany.CompanyID = tblEmployee.CompanyID)
INNER JOIN tblPhone AS tblPhone_Employee
ON tblEmployee.PhoneID = tblPhone_Employee.PhoneID;
```
9. Return to Design view to see much more meaningful names than the _1 designation (see Figure 8.11).

**FIGURE 8.11**

Changing table names can help avoid confusion when you use the same table for different purposes.

Run the query to produce a list of all companies that have phone numbers and also have employees with phone numbers. You can modify the query to remove all the unused fields, or you can save it as is to be used as a basis for a report.

TIP

To modify the preceding query to show all companies and employees, whether or not they have phone numbers, change the joins to the phone tables from inner joins to left outer joins. Double-click the join lines in the top of the query design grid and then choose the appropriate join type in the Join Properties dialog.

Using the Access AutoLookup Feature

Access links records in a query's datasheet view by keeping track of pointers to each table's records. As information changes in the link field on the many side of a one-to-many relationship, the fields from the many side are updated to reflect the new link. This feature is known as *AutoLookup*. The select query in Figure 8.12 shows how the feature works.

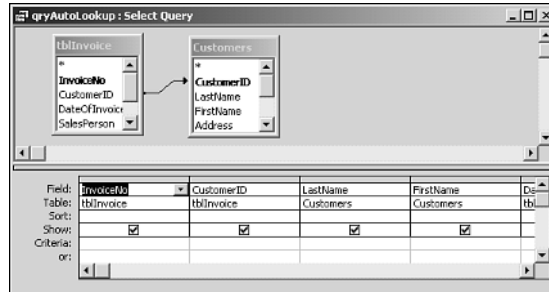


FIGURE 8.12

Notice in this AutoLookup example that the *CustomerID* is from the many side (Invoice).

Open the query in Datasheet view to enter the invoice information into the table. Figure 8.13 shows the datasheet after 20 is entered in the *CustomerID* field and before Tab is pressed to move to the next field. Figure 8.14 shows how the last and first name information is automatically located and displayed after Tab is pressed.

InvoiceNo	CustomerID	LastName	FirstName	DateOfInvoice	SubTotal	Tax
0	12	Lincoln	Elizabeth		300	20
12	12	Lincoln	Elizabeth		199	12.5
14	18	Brown	Elizabeth		233	4
15	19	Ottlieb	Sven		45.51	2.51
16	20				0	0

FIGURE 8.13

Notice how the last and first names look before pressing Tab.

InvoiceNo	CustomerID	LastName	FirstName	DateOfInvoice	SubTotal	Tax
0	12	Lincoln	Elizabeth		300	20
12	12	Lincoln	Elizabeth		199	12.5
14	18	Brown	Elizabeth		233	4
15	19	Ottlieb	Sven		45.51	2.51
16	20	Labruno	Janine		0	0

FIGURE 8.14

With AutoLookup, the last and first names are updated immediately after you press Tab.

Working with Action Queries

Action queries can manipulate large numbers of records very quickly. They also provide the basis for moving data into a database and managing data once it resides in tables. The best way to develop an action query is by converting a standard select query to an action query. This way, you can view the records that the select query will use or affect and then determine whether its criteria is correct before creating the action query.

TIP

By having the query design grid switch to Datasheet view instead of using the Run command, you can also check out what the action query will affect. To do so, click the Datasheet View button on the far left of the Query Designer toolbar.

You can create action queries in the query design grid from the Query menu by choosing an action query type. You can use several types of action queries to insert data and manipulate existing data. Table 8.2 shows the query types and the Access SQL keywords used with them.

TABLE 8.2 Types of Action Queries and Access SQL Keywords

<i>Action Query</i>	<i>SQL Keywords</i>	<i>Purpose</i>
Make Table	SELECT INTO	Creates a new table
Append	INSERT INTO	Appends records to an existing table
Update	UPDATE . . SET	Updates existing records in a table
Delete	DELETE	Deletes records from a table

To familiarize yourself with action queries, start with a select query in the query design grid and explore each action query's capabilities. The following sections discuss each query in Table 8.2 in more detail, as well as include example SQL code.

Make Table Queries (SELECT INTO)

Access allows you to create queries that can create new tables as part of query execution. This way, you can create tables quickly and easily, but you can't create indexes. On calculated fields, Access will guess at the new data type for a created field, which might not match the data type you were expecting. The following is an example of the SQL code for a make table query:

```
SELECT DISTINCTROW Customers.LastName, Customers.FirstName  
INTO tblNewCustomer FROM Customers;
```

Although you can use make table queries to create new tables, consider using them only when you create *ad hoc* tables or need to have new fields automatically created in the output table. To allow for new fields, use the asterisk (*) in the field name. The Make Table query then automatically creates the new table, including any new fields added to the original base table.

TIP

When copying most fields from a table to the query design grid, double-click the table's title bar to select all the table's fields. Click and drag the fields to the query design grid to add all the fields automatically to the grid.

When using Data Definition Language (DDL) queries, you can define the data types as part of a `CREATE TABLE` statement. After creating the table, you can add records by using an append query (discussed in the next section). Unless you need the flexibility of making the new table based on changes in a base table's definition, you should create the table with the DDL query `CREATE TABLE` statement. DDL queries are covered later in the section "DDL Queries."

Append Queries (INSERT INTO)

Append queries add records to an existing table from one or more tables or queries. You can also use append queries to write a single record to a table that's based on information on a form or based on constants entered directly into the query design grid. To create an append query, follow these steps:

1. Select the Query tab in the database container and click the New button.
2. In the New Query dialog, choose OK with the default, Design View, selected.
3. In the Show Table dialog, select the Customers table, click Add, and then click Close.
4. In the query design grid, choose Append from the Query menu.
5. Supply the name of the table to append to—in this case, `tblNewCustomer`.

If the goal is to create new records with all fields in the appended table filled in, go to the Append To line. For each column of the query design grid, select a field from the table to be appended to. Now return to the first column and begin filling in the Field line with the formulas or field references that will produce the correct values for the field. Here's an example of the SQL code for an append query:

```
INSERT INTO tblNewCustomer ( LastName, FirstName )
SELECT DISTINCTROW Customers.LastName, Customers.FirstName
FROM Customers;
```

You can also use append queries to quickly clone information from existing records into other records in the same table. For example, if a series of records contains most of the information you need to create a new set of records, you can create the new records by building an append query. Build a select query with the information to transfer, and then choose Append from the Query menu. The table to append to is the same as the table used in the select query, so choose the table in the Append dialog and click OK. If the fields are copied directly from the query source table to the append table, all the field names will automatically be added to the grid's Append To line.

Update Queries (UPDATE . . SET)

Update queries update existing records with calculated data, constants, or data from other tables. Update queries are useful for setting status codes or calculating field values based on other fields in the table. In the following update query, all records in the Customers table will have the City field set to Woodinville:

```
UPDATE DISTINCTROW Customers SET Customers.City = "Woodinville";
```

Delete Query (DELETE)

Because using delete queries to remove records from a table can be dangerous, make sure that the criteria are correct before executing a delete query. Later, the section “Finding and Deleting Duplicate Records” shows how to use a delete query to delete duplicate records in a table. The following query deletes all the records in the Customers table:

```
DELETE Customers.* FROM Customers;
```

Performing Advanced Query Operations

Select and action queries are quite useful and can answer many problems you face. Often, however, you must summarize a table's records so that a reasonable amount of information shows up on a form or report. By nesting queries atop each other, using sub-SELECT statements, and capitalizing on some of Access's more advanced features, you can really tap the power of queries.

Summary Queries

Summary or *totals* queries allow you to total groups of records and do some statistics on the values. To create a summary query, use the View menu's Totals command from the query design grid.

NOTE

You can combine summary queries with append or make table queries. They can't be used with update, delete, or crosstab options in the same query.

Table 8.3 lists the functions available when a summary query is run. Sum() and Count() solve most of the problems you'll encounter.

TABLE 8.3 Summary Query Functions and Their Purpose

<i>Function</i>	<i>Purpose</i>
Group By	Groups by the selected field. The other functions then use this function to calculate the field(s).
Sum()	Adds the values.
Avg()	Calculates the mathematical average of the values.
Min()	Compares all values and chooses the smallest.
Max()	Compares all values and finds the largest.
Count()	Counts all records where this column isn't null.
StDev()	Calculates the group's statistical standard deviation.
Var()	Calculates the group's statistical variance.
First()	Takes the value from a group's first record.
Last()	Takes the value from a group's last record.
Expression()	Sets the column to be an expression based on constants and group values.
Where()	Ensures that only the criteria section of the query design grid is used for this field.

Figure 8.15 shows a regular select query displaying the subtotal and InvoiceNo for each customer. Figures 8.16 and 8.17 show the Sum() and Count() functions, which don't include records with a null in the field. See the later section "Solving Problems with Queries" for details on nesting summary queries to perform complex query operations.

CustomerID	SubTotal	InvoiceNo
10	300	1
12	199	12
18	233	14
19	45.51	15
20	456	16
0	0	0

FIGURE 8.15

These base records are ready to be summarized by a summary select query.

Field	CustomerID	SubTotal	Invoices Count: InvoiceNo
Table:	tblInvoice	tblInvoice	tblInvoice
Total:	Group By	Sum	Count
Sort:			
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Criteria:			

FIGURE 8.16

The Sum() and Count() functions are shown in a summary query's Design view.

CustomerID	SumOfSubTotal	Invoices Count
10	499	2
18	233	1
19	45.51	1
20	456	1

FIGURE 8.17

Here are the results of a summary query.

Union Queries

A union query combines two tables that have the same structure, with all the records from each table included. One way to use a union query is to add a value (such as <<All Customers>>) to the top of a list of choices, and then include all customers in a report when this choice is made. Chapter 10, “Expanding the Power of Your Forms with Controls,” covers union queries in more detail and provides a complete example.

Nested Queries

When you save queries, you can reference them in many of the same places that you can reference tables, including from the query design grid. If a series of reports will be based on the same general criteria, it sometimes makes sense to build one query with all the criteria in it and use it as the foundation to build other queries.

This technique can pay big dividends during development of reports that will be printed from large amounts of data. Rather than base a summary on a base table, use the Top Values property to base it on a query that returns the first 50 rows. Figure 8.18 shows the select query `qryEmployee50` limiting the data to only 50 records from the `tblEmployee` table.

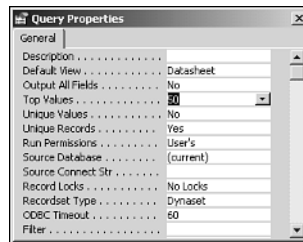


FIGURE 8.18

With the Top Values property, you can choose or enter the Top Values or Top Percentage (by adding a % after the 50).

With `qryEmployee50` as the base query, you can build a new query, `qrptEmployee`, with all the criteria for selecting and sorting the employee data (see Figure 8.19). You can then use the query as input to a report and for testing.

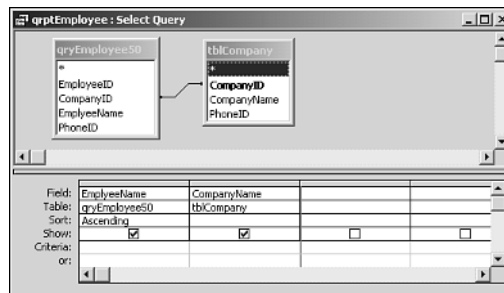


FIGURE 8.19

This query is partly based on the query `qryEmployee50`.

Subqueries

Using a subquery in the criteria for a query allows you to base criteria on the results of another query without creating and saving query. The **Not** operator has a special application for finding a list of records that aren't in another table.

For example, you can list all companies that don't have employees by using **Not In ()**, as shown in the `qryCompaniesWNoEmployees` query in Figure 8.20. The **subselect** limits the query to include only the company records that don't have matching `CompanyIDs` in `tblEmployee`.

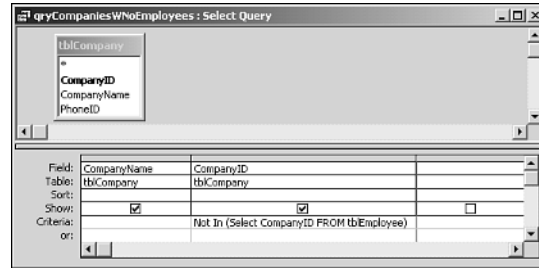


FIGURE 8.20

This query uses a subquery for the `CompanyID` criteria.

DDL Queries

Select and action queries allow you to manipulate records within the database. DDL queries allow you to create objects in the database. Although DDL provides the functionality, you can perform the same operations through ADO. For information on ADO, refer to Chapter 5, "Introducing ActiveX Data Objects."

Adding More Power with VBA

To get even more power from queries, use them from VBA where you can use control-of-flow statements such as `If...Endif` and `Select Case...End Select` to build SQL statements and pass parameters to queries. You can give your users more versatile user interfaces for selecting and viewing their data.

Building Faster Queries

With VBA, you don't need to use a parameter query to have users specify the criteria. Consider a report that requires two or three optional parameters. If the parameters involve multiple tables, the query might be joining tables that are needed only some of the time. To make the query run more quickly, build only the required portion of the `SELECT` statement.

Suppose that users can choose to select all states or just one state. The criteria for `=[State]` or `IsNull()` would need to be evaluated as part of the selection. Building the `SELECT` statement in VBA gives you the option of eliminating the criteria completely if users don't supply a value.

Some criteria can be determined based on information in other tables or information determined in the VBA code. Most often, the need to alter criteria comes from query-by-form requirements.

NOTE

Building queries on-the-fly like this means that they aren't precompiled. This might turn out to be slower than having a more complex query stored as a precompiled query.

Using Query by Form

Query by form (QBF) is one of the most requested features of most any system. Developers are regularly challenged with striking a balance between ease of use and features. Query by form provides users with a "fill-in-the-blank" approach to solving most queries users likely will ask regularly.

The simplest version of QBF is to provide users with a form containing all fields in a table. Users fill in a field; then, when they click the form's search or OK button, the records that match the values the users entered are retrieved and shown in a report or list box. Access provides this functionality by using the filter-by form available to users in the form and table views. Generally, this won't be enough for end users who want a simple screen with yes/no options.

A Simple Query by Form

Figure 8.21 shows a simple query-by-form example. The user regularly wants to run a query for customers based on the first letter of the customer's last name. If a letter isn't specified, all customers are included.

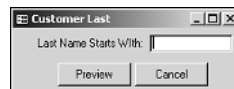


FIGURE 8.21

This form, frmCustomerLast, is an example of query by form.

When the user clicks Preview, the code in Listing 8.1 executes.

LISTING 8.1 Chap08.mdb: Creating and Opening a Query

```
Private Sub cmdPreview_Click()  
    Dim strSQL As String  
    Dim catCurr As New ADOX.Catalog  
    Dim cmdCurr As New ADODB.Command  
    strSQL = "SELECT Customers.* FROM Customers"  
    If Not IsNull(Me![txtLetter]) Then  
        If Me![txtLetter] <> "" Then  
            strSQL = strSQL & " WHERE LastName Like '" & Me![txtLetter] & " *'"  
        End If  
    End If  
    catCurr.ActiveConnection = CurrentProject.Connection  
    Set cmdCurr = catCurr.Views("ztqryCustomersByLetter").Command  
    cmdCurr.CommandText = strSQL  
    Set catCurr.Views("ztqryCustomersByLetter").Command = cmdCurr  
    DoCmd.OpenQuery "ztqryCustomersByLetter"  
End Sub
```

The code creates a SQL string with the basic SELECT statement, and then adds a WHERE clause *only if* users entered a value in the first letter text box. You can extend this example by adding more text boxes for users to fill in. If users need to select two letters, for example, you can add a second text box. The code that builds the WHERE clause needs to be modified to include OR LIKE if users fill in each text box.

A More Complex Query by Form

For a better idea of what you can do with QBF, a more complex example takes multiple fields on a form and creates a record source for a subform (see Figure 8.22). This form, frmQueryByFormExample, can be found in the Chap08.mdb database on this book's Web page at www.sampublishing.com.

NOTE

Break out this example into its own database to make it as simple as possible for your own applications. You'll take the techniques used here, rather than the specific code, because this example is pretty exact in its use.

The screenshot shows a form titled "frmQueryByFormExample : Form". It has three main sections: "Category To Include", "Ratings To Include", and "Beginning Purchase Date".

Category To Include: A list box with the following items: Action/Adventure (selected), Drama, Horror, Comedy, Childrens, Family, and Science Fiction.

Ratings To Include: A group of checkboxes: G (unchecked), PG (unchecked), PG-13 (unchecked), R (checked), and NC-17 (unchecked).

Beginning Purchase Date: A text box with a date picker icon.

Auto Query / No Auto Query: Two radio buttons. "Auto Query" is selected.

Buttons: "Requery" and "Clear".

Query By Form Data: A table showing the results of the query.

Title	Rating	CategoryCode	Description	Purchase Date
Die Hard	R	Action/Adventure	Bruce Willis Stars in this action packed adventure	3/1/2001
Fargo	R	Drama	Interesting drama about a kidnapping scheme gon	6/1/2001
Glen Gary, Glen Ross	R	Drama	Great drama with a big cast	2/1/2001
Good Will Hunting	R	Drama	Mid story	4/1/2001
L.A. Confidential	R	Drama	Crima drama with an all star cast	2/1/2001
Rumble in the Bronx	R	Action/Adventure	Fun action with Jackie Chan	7/1/2001
Schindler's List	R	Drama	Very Intense movie about the Holocaust	3/1/2001

Records: 1 of 12

FIGURE 8.22

Allowing your users to see results based on their choices increases a system's flexibility.

This example is actually pretty complete in that it allows you to

- Select data items and have them automatically requery the subform
- Turn off the auto requery feature, but turn a manual requery button red to notify users if the subform needs to be requered because of changed criteria
- Use a manual Requery button after all criteria are set
- Clear the currently selected criteria with a Clear button

TIP

I included an option for turning off auto requery because with large recordsets, it can become cumbersome to have the system requery every time you pick a new criteria. This way, users can set up all the criteria they want, and then click the Requery button. Users will then feel that the system is pretty quick regardless of the data items, and also feel as though they're in control.

This example doesn't let users combine the different criteria with ANDs and ORs. It ANDs the different pieces of criteria, and ORs the like pieces of criteria. This becomes clearer as you walk through the example.

You can find all the important items in this example on the `frmQueryByFormExample` form. If you open this form in Design view, you'll see the following controls:

- The `lboCategoryToInclude` multiselect list box allows users to include multiple movie categories. This list box is based on the `Categories` table.
- A series of check boxes—`chkRatedG`, `chkRatedPG`, and so on—represent the possible movie ratings.

TIP

You can specify the same type of criteria by using either a multiselect list box or a series of check boxes. These are both shown to give you a more complete example to help you choose between different methods of displaying information. Use check boxes when you need to present a limited and static list; otherwise, I recommend using list boxes because they're easier to maintain.

- The txtBegPurchaseDate text box contains the beginning purchase date the user is querying.
- The optAutoRequery option group enables and disables the Auto Requery feature.
- The cmdRequery command button re-creates the subform record source when you have the Auto Requery feature turned off.
- The cmdClear command button resets the criteria choices and clears the subform.

In the first scenario, you leave the Auto Requery feature turned on. Whenever you click the list box, select the ratings check boxes, or enter a new purchase date, the RequerySubform() sub-routine is called from the control's AfterUpdate event. RequerySubform() is located in the general routines of the form frmQueryByExample. You can see this code in Listing 8.2.

LISTING 8.2 Chap08.mdb: Combining the User's Choices and Updating the Subform's RecordSource Property

```
Private Function RequerySubform()
    Dim strCategorySQL As String
    Dim strRatingsSQL As String
    Dim strPurchaseDateSQL As String
    Dim strWhereSQL As String
    Dim strFullSQL As String

    '-- If AutoRequery is set to True, or the Requery button was pressed, then
    '-- re-create the Where clause for the recordsource of the subform.
    If Me!optAutoRequery Or Screen.ActiveControl.Name = "cmdRequery" Then
        '-- Store all the criteria for the Where statement into variables.
        strCategorySQL = IncludeCategories()
        strRatingsSQL = IncludeRatings()
        strPurchaseDateSQL = IncludePurchaseDate()
        '-- Store the initial Where statement with whatever is from
        '-- the Category criteria.
        strWhereSQL = "Where " & strCategorySQL
        '-- If a rating was passed back, then add it to the Where clause.
        If Len(strRatingsSQL) <> 0 Then
```

LISTING 8.2 Continued

```

    '-- If the Category criteria was already added,
    '-- AND it with the Ratings criteria.
    If strWhereSQL <> "Where " Then
        strWhereSQL = strWhereSQL & " And "
    End If
    strWhereSQL = strWhereSQL & strRatingsSQL
End If
'-- If a Purchase date was passed back, then add it to the Where clause.
If Len(strPurchaseDateSQL) <> 0 Then
    '-- If some of the other criteria was already added,
    '-- AND it with the Purchase date criteria.
    If strWhereSQL <> "Where " Then
        strWhereSQL = strWhereSQL & " And "
    End If
    strWhereSQL = strWhereSQL & strPurchaseDateSQL
End If
'-- If no criteria was chosen, make it so the subform will be blank.
If strWhereSQL = "Where " Then
    strWhereSQL = "Where False"
End If
'-- Create the new SQL String and Store it to the Recordsource.
strFullSQL = "Select * From MovieTitles " & strWhereSQL
Me!subQueryByForm.Form.RecordSource = strFullSQL
'-- Set the requery button to black.
Me!cmdRequery.ForeColor = 0
Else
    '-- Set the requery button to red.
    Me!cmdRequery.ForeColor = 255
End If

```

End Function

The `RequerySubform()` function performs the bulk of the form's tasks, either by doing the work itself or by calling those routines that do. First, the function checks to see whether `optAutoRequery` is set to true or the `Requery` button was clicked:

```
If Me!optAutoRequery Or Screen.ActiveControl.Name = "cmdRequery" Then
```

TIP

This code line is a great example of using an option group in a true/false situation. Setting the first option to -1 (true) and the second option to 0 (false) cuts down coding because you can treat the option group as a logical expression.

Next, `RequerySubform()` calls functions that create the individual criteria segments. These calls are a good way to create the segments because you can then debug each routine. The following code calls the functions:

```
strCategorySQL = IncludeCategories()
strRatingsSQL = IncludeRatings()
strPurchaseDateSQL = IncludePurchaseDate()
```

These functions show how to use different methods to create SQL criteria strings. The `IncludeCategories()` function in Listing 8.3 shows how to create a portion of SQL criteria with the `lboCategoryToInclude` multiselect list box. In Listing 8.4, `IncludeRatings()` uses the check boxes and combines those that have been chosen into one string. The last function, `IncludePurchaseDate()` in Listing 8.5, takes a text box and concatenates it to a string used for the SQL criteria. These routines all pass back a string that's combined later.

NOTE

For more information on using the multiselect list box programmatically, see Chapter 10 or Chapter 11, "Creating Powerful Reports."

LISTING 8.3 Chap08.mdb: Creating a SQL Criteria String with a Multiselect List Box

```
Private Function IncludeCategories() As String
    '-- Create the Categories Where portion of the SQL statement
    Dim varCategory As Variant
    Dim strTemp As String

    '-- for each of the items in the ItemsSelected collection
    For Each varCategory In Me!lboCategoryToInclude.ItemsSelected()
        strTemp = strTemp & "[CategoryCode] = " & _
            Me!lboCategoryToInclude.ItemData(varCategory) & " Or "
    Next
    If Len(strTemp) > 0 Then
        IncludeCategories = "(" & Left$(strTemp, Len(strTemp) - 4) & ")"
    Else
        IncludeCategories = ""
    End If
End Function

End Function
```


LISTING 8.4 Chap08.mdb: Using Check Boxes for a Static List with a Limited Number of Choices

```
Private Function IncludeRatings() As String
    '-- Create the Categories Where portion of the SQL statement
    Dim strTemp As String

    If Me!chkRatedG Then
        strTemp = "[Rating] = 'G'"
    End If

    If Me!chkRatedPG Then
        '-- If a rating before this one was included, OR them.
        If Len(strTemp) <> 0 Then
            strTemp = strTemp & " Or "
        End If
        strTemp = strTemp & "[Rating] = 'PG'"
    End If

    If Me!chkRatedPG13 Then
        '-- If a rating before this one was included, OR them.
        If Len(strTemp) <> 0 Then
            strTemp = strTemp & " Or "
        End If
        strTemp = strTemp & "[Rating] = 'PG-13'"
    End If

    If Me!chkRatedR Then
        '-- If a rating before this one was included, OR them.
        If Len(strTemp) <> 0 Then
            strTemp = strTemp & " Or "
        End If
        strTemp = strTemp & "[Rating] = 'R'"
    End If

    If Me!chkRatedNC17 Then
        '-- If a rating before this one was included, OR them.
        If Len(strTemp) <> 0 Then
            strTemp = strTemp & " Or "
        End If
        strTemp = strTemp & "[Rating] = 'NC-17'"
    End If
```

LISTING 8.4 Continued

```
'-- If at least one rating was chosen, assign the
'-- temp string as the return value.
If Len(strTemp) <> 0 Then
    IncludeRatings = "(" & strTemp & ")"
End If

End Function
```

LISTING 8.5 Chap08.mdb: Using a Text Box with Query by Form

```
Private Function IncludePurchaseDate() As String
    '-- Create the Categories Where portion of the SQL statement
    If Not IsNull(Me!txtBegPurchaseDate) Then
        IncludePurchaseDate = "([DatePurchased] >= " & _
            "Forms!frmQueryByFormExample!txtBegPurchaseDate)"
    End If
End Function
```

After the subroutines are completed, RequerySubform() starts the WHERE clause, with the strCategorySQL string concatenated to it with this line of code:

```
strWhereSQL = "Where " & strCategorySQL
```

You can use the next piece of code both for Ratings and Purchase Date, with minor changes. The code for Ratings is shown here:

```
'-- If a rating was passed back, then add it to the Where clause.
If Len(strRatingsSQL) <> 0 Then
    '-- If the Category criteria was already added,
    '-- AND it with the Ratings criteria.
    If strWhereSQL <> "Where " Then
        strWhereSQL = strWhereSQL & " And "
    End If
    strWhereSQL = strWhereSQL & strRatingsSQL
End If
```

The code is similar for Purchase Date; it checks to see whether there's anything in the current section of the Where statement, strRatingsSQL, and then concatenates an AND statement if something is already added to strWhereSQL. Finally, it adds strRatingsSQL to strWhereSQL.

Next, the code checks to see whether any conditions were created; if not, the code uses Where False so that no records show up. The code then concatenates the Where clause to the rest of the SQL statement and assigns that to the subform's RecordSource property:

PART II

```
If strWhereSQL = "Where " Then
    strWhereSQL = "Where False"
End If
```

```
'-- Create the new SQL String and Store it to the Recordsource.
strFullSQL = "Select * From MovieTitles " & strWhereSQL
Me!subQueryByForm.Form.RecordSource = strFullSQL
```

Finally, the Requery button's color is set, based on whether AutoRequery is set or a manual requery has just been performed. If so, the color is set to black; otherwise, the color is set to red, so users know that they might want to click the Requery button. Here's the code:

```
'-- Set the requery button to black.
Me!cmdRequery.ForeColor = 0
Else
    '-- Set the requery button to red.
    Me!cmdRequery.ForeColor = 255
End If
```

That's it for the RequerySubform() function. Most buttons and controls call RequerySubform(). The only other routine is the event procedure behind the Clear button's OnClick event (see Listing 8.6).

LISTING 8.6 Chap08.mdb: Clearing All Controls and Calling RequerySubform()

```
Private Sub cmdClear_Click()
    Dim varDummy As Variant
    Dim intCurrCat As Integer

    '-- Clear all the criteria
    '-- First, the multi-select list box.
    For intCurrCat = 0 To Me!lboCategoryToInclude.ListCount - 1
        Me!lboCategoryToInclude.Selected(intCurrCat) = False
    Next

    Me!chkRatedG = False
    Me!chkRatedPG = False
    Me!chkRatedPG13 = False
    Me!chkRatedR = False
    Me!chkRatedNC17 = False
    Me!txtBegPurchaseDate = Null

    '-- Re-create the RecordSource for the subform
    varDummy = RequerySubform()

End Sub
```

Creating Temporary Command Objects

In VBA and ADO, a temporary Command object allows you to create a query that won't be included in the Catalog collection. This feature is useful when you have common library routines called throughout an application. A temporary Command isn't given a name and therefore eliminates the need for you to be concerned about finding a unique name for the query throughout the entire application. You can create a temporary Command object only in VBA, and it's deleted automatically when closed. You also can create multiple temporary Command objects within VBA at the same time.

To create a temporary Command object in VBA, create a new Command object without going through the Catalog object. You can set Command's CommandText property to perform a specific action; the Execute method will invoke the action. The Close method closes and deletes the query.

The following code sample shows one possible use of a temporary query by deleting records from a table without saving the query:

```
Dim cmd As New ADODB.Command
'*** Create temp command object
cmd.ActiveConnection = CurrentProject.Connection
cmd.CommandText = "Delete * FROM tblCompanyTest"
cmd.CommandType = adCmdUnknown
cmd.Execute
```

Using the DoCmd Object's RunSQL Method

The DoCmd object in Access allows you to execute SQL statements against the current database. DoCmd is similar to the functionality of a temporary Command object used to execute action queries.

The method used off the DoCmd object is RunSQL. One disadvantage to RunSQL is that it operates only on the current database. Unlike a temporary Command object, which can be created against another database, RunSQL is based only on tables that exist in the current database.

RunSQL prompts users about changes performed unless warnings are turned off. The following code clears the salesperson code from all records in the invoice table. Warnings are turned off and then back on so that users don't have the option of canceling the operation.

```
DoCmd.SetWarnings False
DoCmd.RunSQL "UPDATE tblInvoice SET SalesPerson = NULL"
DoCmd.SetWarnings True
```

RunSQL doesn't allow you to create an object, so you can't use it to create dynasets or snapshots. The DoCmd object was created to help users migrate from Access macros to VBA.

Developers generally avoid using `DoCmd.RunSQL` and instead use temporary queries consistently to execute select and action queries against a database. This avoids the problem of turning warnings off before and then back on after the `RunSQL` method, as well as during error recovery. Using temporary queries also allows the statement to execute on a database other than the current default database.

Issuing Parameter Queries

The parameters passed to parameter queries are handled differently in VBA than they are from the database container or query designer. In the query designer, you can create parameters that reference forms. Running the query retrieves the parameters from the forms as needed. If the query is called from VBA, the parameters must be set from VBA even if the form is open. The form shown in Figure 8.23 has three options for running the query: Preview, Bad Code, and Good Code.

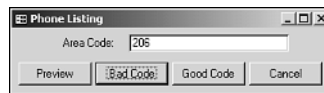


FIGURE 8.23

This form, frmPhoneList, shows how and how not to use parameter queries with VBA.

The Preview button uses `DoCmd.OpenQuery` to open the query from the database container:

```
DoCmd.OpenQuery "qryPhoneList"
```

The Bad Code button runs VBA code that doesn't properly set the parameters before calling the Open method, resulting in a runtime error (see Figure 8.24). The parameters haven't changed and the form is still open, but the form parameters aren't understood under VBA.

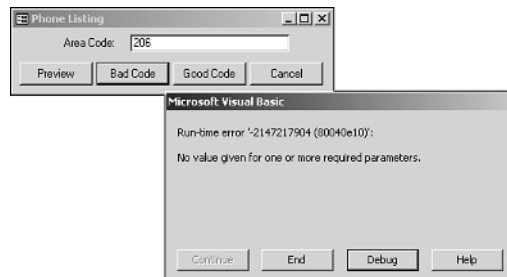


FIGURE 8.24

When dealing with parameterized queries through VBA, not entering the correct amount of parameters results in the No value given for one or more required parameters error.

The following code sets the parameter properly so that the query will execute and return the five records you expect. By setting the parameter this way, you can use parameter queries from code even when they reference forms.

```
cmd.Parameters("[Forms]![frmPhoneList]![txtAreaCode]") = _  
    Forms![frmPhoneList]![txtAreaCode]
```

Driving Reports and Forms with Queries

Reports and forms can use queries or SQL statements as a record source. If the record source will change based on complex criteria, it can be helpful to have the report pointing to a query that contains the SQL statement for the report. VBA code can change the query before opening the form or report rather than change it during the Form OnOpen or Report OnOpen event.

Creating the query ahead of time is advantageous because even before the VBA code is written to generate the real query, you can create a query with the fields that will be used on the final report and begin the report design. This allows you to develop the report and view data from existing tables until the VBA code is written. When the code is written, it simply changes the SQL property of the existing query and opens the report.

Another advantage of using a query to drive reports and forms this way is evident if it becomes necessary to debug the record source. Set a breakpoint in the VBA code after closing the query and before opening the report. When the code is run, it will stop at the breakpoint; pressing F11 will bring up the database container. At this point, open the query or design it to see whether the VBA code generated the expected results.

Solving Problems with Queries

The following sections cover some common business questions and how they can be solved by using queries. You can adapt the queries to cover various questions common to developers. Some business questions answered are as follows:

- How many times do I have to rent out a video before it pays for itself? The section “Grouping to Get Percentages” shows how to get the percentage of a rental cost to retail cost of a video, thus allowing you to use it to get the number of times you need to rent it out.
- In some cases, duplicate records slip in and need to be taken care of. The section “Finding and Deleting Duplicate Records” shows how to perform this task.
- Sometimes it takes a few levels of queries to get to the desired results when analyzing data. This is discussed in the section “Nesting Groups to Get the Complete Solution.”

Additional answers are brought forth by using different methods with queries.

Grouping to Get Percentages

Sometimes you need to calculate percentages in a query. If the numbers required for the calculation are in the same record, the process is easy and can be accomplished in a formula, as shown by the qryMovieTitlePercentOfRetail query in Figure 8.25. To see the percentage, use the query's Datasheet view (see Figure 8.26).

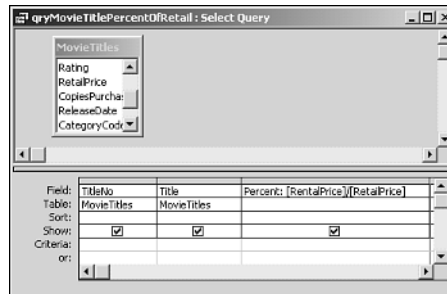


FIGURE 8.25

Creating a field giving the percentage of other fields is very straightforward.

TitleNo	Title	Percent
1	Die Hard	0.0286
2	Dumb & Dumber	0.0200
5	Schindler's List	0.0588
6	Armagedon	0.0882
7	Liar Liar	0.0308
8	Usual Suspects	0.0682
9	Good Fellas	0.0260
10	Hope Floats	0.0462
11	The Thing	0.0444
12	Predator	0.0667
13	Fargo	0.0633
14	Rumble in the Bronx	0.0882
15	Unforgiven	0.0633

FIGURE 8.26

The properties for the Percent column display the percentages with a fixed decimal point.

Most often, percentages are calculated based on the sum or count of all records in the query. The number isn't available on each record and must be calculated by using a query. Create the query (as shown in Figure 8.27) and save it.

To create the query with percentages, create a new query from the same table and choose Show Table from the Query menu to add the previously created query to the query design grid. Don't create any join lines between the table containing the totals and the query's base table.

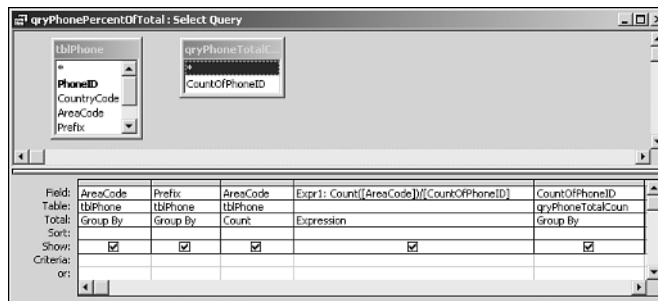
**FIGURE 8.27**

The count of total phone numbers is displayed for the `tblPhone` table.

NOTE

This query uses the Cartesian product of two tables. A *Cartesian product* results when a join line isn't used between two tables in a query. The resultset will contain one record for each combination of rows in the two tables. If each table has three rows, you get nine rows in the resultset ($3^2=9$). This can get ugly and dangerous for large datasets.

Create the formula based on the `Count()` function to get the group's record count, and divide by the summary query's total to get the percentage (see Figure 8.28).

**FIGURE 8.28**

This query presents the percentage of total phone numbers found in `tblPhone`.

Finding and Deleting Duplicate Records

Deleting duplicate records usually involves manually building a list of records and then going through them one by one to locate and delete the duplicates. You can use a summary query to identify the characteristics of a duplicated record, but summary queries don't provide a method for locating all but one record from the result list and then deleting them. (As usual, the queries discussed in this section can be found on this book's Web page in the Chap08.mdb database, in the \Examples\Chap08 folder.)

Use a unique identifier in the record and use nested queries to solve the problem. For this example, the qryMovieTitlesDuplicated query begins coming up with a solution to the task with a list of duplicated titles.

By using the select query qryMovieTitlesDuplicated as the basis, you can create a summary query that groups by titles and grabs the lowest (minimum) title number for each title. Join the list of duplicated titles so that the records returned by the query include only duplicated titles, as in the qryMovieTitlesDuplicatedFirst summary query. Notice in Figure 8.29 that the Total row is visible in the grid.

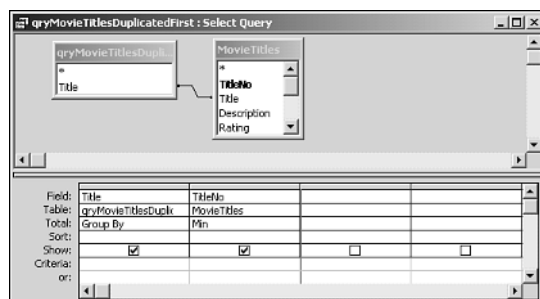
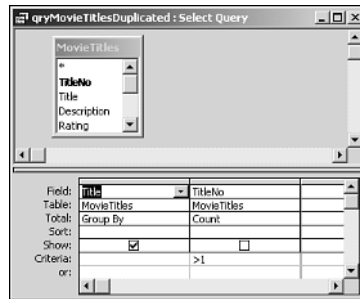


FIGURE 8.29

Create an ID of the first duplicated titles by using this summary query.

With the list of duplicated titles identified and the list of the first record in each duplicate group identified, you can build the delete query. Do so by selecting all records that have duplicated titles but aren't the first record of the duplicate group. Figure 8.30 shows the delete query that removes the duplicate records.

**FIGURE 8.30**

This delete query will remove duplicated records.

NOTE

To adapt this procedure to work against tables that don't already have a unique record identifier, create a field in the table and give it a unique value for each record. You might need to use VBA and increment a value as you loop through each record and store the value in the record. When you have a unique identifier, the procedure described here should work to remove duplicate records.

8

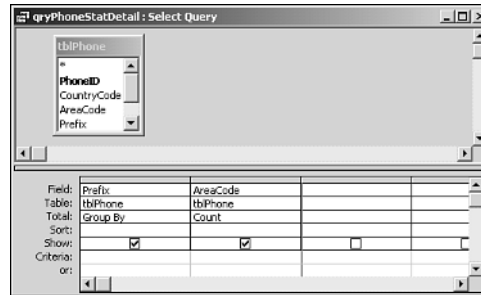
USING QUERIES TO
GET THE MOST OUT
OF YOUR DATA

Nesting Groups to Get the Complete Solution

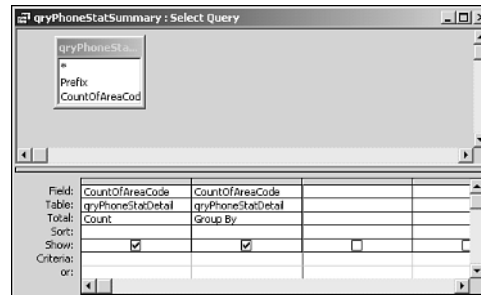
Business, financial, and manufacturing analysts ask similar questions about statistics from databases. For example, the owner of the phone list wants to know how randomly scattered the people are on the phone list. A count of the number of entries for each prefix isn't enough to solve the problem because it shows only what codes have the highest number of entries. You want to know how many prefix codes have that same number of records. The problem is solved by nesting two queries.

To solve the first part of the problem, build a summary query that groups and counts on the phone prefix. Figure 8.31 shows this query, named `qryPhoneStatDetail`.

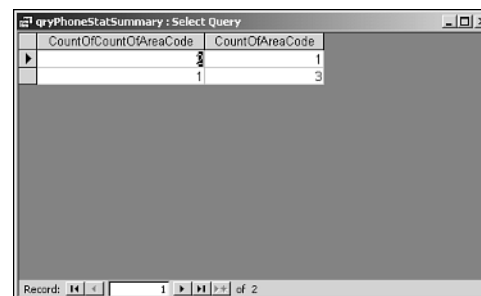
With the detail query as a basis, create a new summary query. Copy the count field from the detail query into the query design grid twice. Set the first Total to Count and the second Total to Group By in the `qryPhoneStatSummary` query (see Figure 8.32). Running the query shows that two prefixes have one phone number, and one prefix code has three different phone numbers (see Figure 8.33).

**FIGURE 8.31**

This query, in Design view, counts the number of records with the same phone prefix.

**FIGURE 8.32**

This query gives the phone status.

**FIGURE 8.33**

Queries within queries can perform even obscure tasks, as shown with the qryPhoneStatSummary query.

Distinguishing Between New and Old Records

In this example, a user has two tables that list movie titles available in January and February. The user wants a single list showing the movies deleted from the list in January (Deleted) and the new movies added in February (Added).

To solve this problem, give the records a number, depending on which table they're selected from. If the title is in the Jan table, give it a 1. If the title is in the Feb table, give it a 2. The following union query results in records that follow this model:

```
SELECT DISTINCTROW MovieTitlesJan.TitleNo, 1 AS OldNew
FROM MovieTitlesJan
UNION
SELECT DISTINCTROW MovieTitlesFeb.TitleNo, 2 AS OldNew
FROM MovieTitlesFeb;
```

Save this query as qryMovieTitlesUnion and use it as the basis for a summary query. The summary query qryMovieTitlesSummary shows the summarized number and uses an IIf() statement to display the added/deleted text (see Figure 8.34).

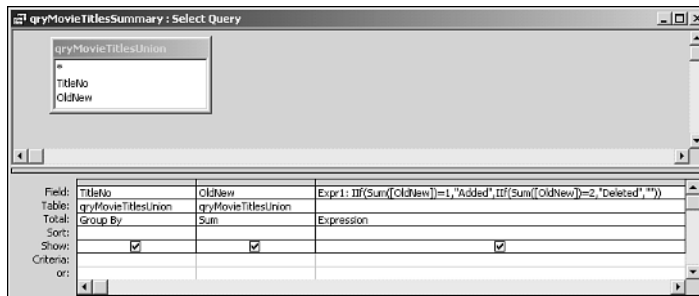


FIGURE 8.34

The query summarizes old and new records.

Creating a Total Row for Crosstab Queries

When using Access crosstab queries, you can total columns by including additional row summary fields. But to calculate column totals at the bottom of the datasheet, your only choices used to be using either Excel pivot tables or a report. However, you also can include a total row on all your crosstabs, if you want.

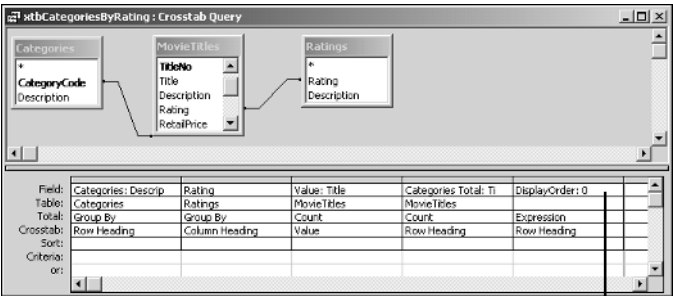
Creating row totals for crosstabs is one of those topics that, until you have a need for it, you'd never find it to be convenient. When you have a need and solve the problem, however, you'll find uses for it all over the place.

The solution combines a summary query, two crosstab queries, and a union query. To start, you first need to follow these steps before you can combine all those queries into an intelligent answer:

- 1. Create a crosstab query to handle the detail.
- 2. Create a summary query to total the values for each rating.
- 3. Create a crosstab query from the summary query, giving it the same field layout as the first crosstab. This is to prepare for using the union query, which needs to have fields be in the same order.
- 4. Create a union query to combine the two crosstabs, grouping by the DisplayOrder field. This makes sure that the total crosstab query values are displayed last.

Handling the Detail with a Crosstab Query

First, create a crosstab query that will handle the data’s main detail. This query will actually be the xtbCategoriesByRating crosstab created for the report solution, with one added field. This field, DisplayOrder, is set to 0 in the column’s Field property (see Figure 8.35).



New field added and set to 0

FIGURE 8.35

The crosstab query has the DisplayOrder calculated field added.

NOTE

The Categories field is so named so that the word *Categories* will appear for the header instead of *Description*. You can also accomplish this by changing the field’s Caption property.

Totaling the Values for Each Rating with a Summary Query

This simple summary query will count how many titles in the `MovieTitles` table are each assigned a rating. The first field is an expression called `Categories` and is set to `Rating Totals`. (Although it might sound strange to have a field named `Categories` containing the value `Rating Totals`, doing so works to your advantage in the next step.)

`Rating` is the next field to set `Group By` on. Last, you'll want to set a `Group By` on the count of the titles for the given ratings. Figure 8.36 shows `ttlRatingsTotals` in Design view, and a copy called `ttlRatingsTotalsRunning` in Datasheet view.

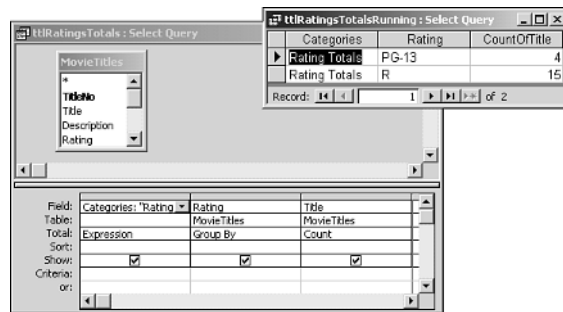


FIGURE 8.36

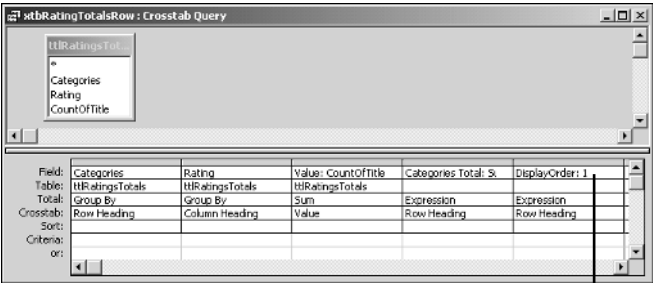
This summary query will be used as a base for another crosstab query.

By looking at the datasheet in Figure 8.36, you can see that this is now set up to use in a crosstab query. The data is ready because you have the required fields for a crosstab query: a Row Headings field (`Categories`), a Column Headings field (`Rating`), and Value field (`CountOfTitle`).

Creating a Crosstab Query from the Summary Query with the Same Field Layout

Now that the summary query is created, you need to place it in a format that will work with the original crosstab query you created. To do so, you must create a new crosstab against the summary query. This actually isn't as tough as it sounds because you can base queries off other queries as easily as you can tables.

Figure 8.37 shows what the new query will look like. Aptly named `xtbRatingTotalsRow`, it is, in fact, almost identical to the original crosstab created. One difference is that although both queries contain the calculated field `DisplayOrder`, the `xtbCategoriesByRating` crosstab has the `DisplayOrder` field set to 0, whereas it's set to 1 in `xtbRatingTotalsRow`.



New field added is now set to 1

FIGURE 8.37

If you compare this crosstab query to the query in Figure 8.35, you'll see that they're similar.

Combining the Two Crosstabs with a Union Query

It's now time to combine the two crosstabs with a union query. Union queries combine two recordsets with the same field structure—in this case, xtbCategoriesByRating and xtbRatingTotalsRow.

You can't create union queries by using the query design grid. Instead, create a new query in Design view, and then choose SQL from the View menu. Then type the following SQL string:

```
SELECT * FROM xtbCategoriesByRating UNION  
(SELECT * FROM xtbRatingTotalsRow)  
ORDER BY DisplayOrder;
```

Notice where the parentheses appear. By having the ORDER BY where it is, the operation of ordering by DisplayOrder is performed on both recordsets after they're unioned. Figure 8.38 shows the final recordset.

Categories	G	PG	PG-13	R	NC-17	Categories Total
Action/Adventure				1	3	
Comedy				2	1	3
Drama				1	9	10
Science Fiction				2		2

FIGURE 8.38

Here is the final product.

This method works great when you include it as a subform to display information. As with all these methods, the data isn't updatable because the fields are what are known as *aggregate values*.

Extra Credit: Specifying Column Headings in a Crosstab

If you open the `tblRatings` table, notice that there are five ratings instead of the two showing, as you just saw in Figure 8.38. The other three ratings didn't show up because no movie titles currently in the `MovieTitles` table (the basis for these queries) have those ratings. Not having a column display is either a feature or flaw of using crosstabs, depending on your situation.

You can get the ratings to show in a couple of ways, of course. One way is to hard-code them in the crosstab's Column Headings property. To see this property, choose Properties from the View menu while in the top half of the desired crosstab's query design grid. You'll then see the property sheet for the query (see Figure 8.39).

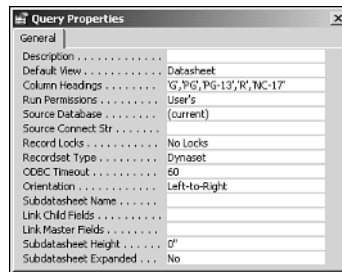


FIGURE 8.39

You can set the query property Column Headings to display all columns in a given order.

The biggest issue with solving the problem this way is that if the columns change at all, you must go in and change the property setting. This means that, in this case, two crosstab queries must be updated. The alternative answer is to use code. For more information on creating dynamic columns in crosstab queries by using code, see Chapter 11.

Examining the Architecture of the Query Resolution Process

Before you spend quality time tuning query performance, it's important that you understand the process that resolves the request. From the first instant you decide to pose a question to the database in terms of a query, you begin to follow a series of steps that you hope will result in an accurate, timely result. When you begin to optimize the query, it helps to understand these steps and know which part of the process you're optimizing.

Defining the Query

Many developers don't include a definition as part of query resolution, but it does affect how the query is resolved. When deciding how to pose the query to the database, you've probably already determined which tables to use and how to relate them. It's very important to review your thinking on this and make sure that you're asking the right question.

Compiling the SQL Statement

The SQL statement is compiled when it's stored in the query. This process is usually very quick, even when large numbers of records are involved. The time spent during the compile process isn't significant.

TIP

Having list boxes and combo boxes filled from saved queries rather than from SQL statements is beneficial because then the compile phase doesn't need to occur.

Preparing the Execution Plan (Optimization)

The Jet database engine lists the available indexes and tables that you can use to resolve the query. Even for queries that return large numbers of records or require complex joins, Jet looks at each possible way to retrieve the records. The time for each part of the query is estimated based on record counts and statistical information in the index. Some statistics used in this process include the following:

- **The table's size.** If the table is small, it might be faster to read the data from the table than to use indexes. A table's size is related to the number of records and amount of information in each record. This statistic is based on the number of pages the table takes up, not the number of records.
- **Index size.** This again isn't based just on the number of records but on the size of the indexed fields, which could be quite large when compound indexes are used.
- **Rushmore technology.** The optimizer, discussed fully in the next section, looks at the potential for merging two or more indexes to help solve parts of the query without reading the base table directly.

After many different possible resolutions are worked out, a cost is assigned to each. The plan expected to take the least time to complete is chosen as the winner, and the query proceeds. The final step is when Jet actually begins retrieving records and building the snapshot or dynaset.

Optimizing Queries with Jet

The following sections cover some of the optimization methods that the Jet database engine applies during the optimization porting of the query resolution process. This includes taking advantage of Rushmore technology, paying attention to when to use indexes, and understanding how Jet optimizes queries.

Using Rushmore Technology

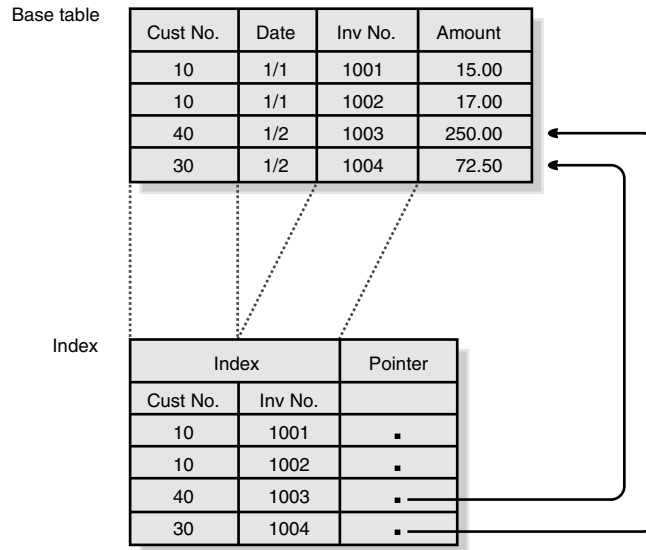
Rushmore technology is a query technology that takes advantage of indexes. First introduced in FoxPro 2 by Fox Software, its creator got its name from the Hitchcock classic *North by Northwest*. Access 2 inherited some of the Rushmore technology, and you can greatly increase performance by taking advantage of it.

Rushmore technology involves using the indexes on tables to solve WHERE clauses in SQL statements. Rushmore takes advantage of a table's multiple indexes and takes effect when the WHERE clause references multiple indexed fields. To understand how this works, take a quick look at Access's index structure.

When you add an index to an Access table, you're essentially running a make table query (or make index, in this case), which creates a miniature version of the table with a pointer back to the main record. In this case, the pointer isn't the primary key but instead is a bookmark or internal pointer that contains the information to locate and retrieve the main record quickly. Each index contains an entry for every record in the original table. Figure 8.40 shows an example of how indexes work internally.

Rather than use one index to limit the record search and resolve the rest of the WHERE clause by looking at each record, Rushmore looks at multiple indexes and applies part of the criteria to the appropriate indexes. When a list of the matching bookmarks is obtained from a single index, the bookmarks are compared with the bookmarks from other index searches. The bookmarks are unioned (or) or joined (and) to get a resulting list of bookmarks that you can use to retrieve the actual data records. If the result of the query is a count and the field you're counting on is in the index, the query will perform the count on the index rather than retrieve actual records from the table.

Rushmore also applies when compound indexes are present and contain fields used in the queries. The restriction here is that the fields in your criteria must be included as the first fields, not the last fields, in the index.

**FIGURE 8.40**

This is how indexes look behind the scenes, with pointers to specific records.

Examining the Clustered Primary Index

Access stores records in tables in the order they're entered into the system. Creating indexes and relationships for tables doesn't affect the data's order in the table in most cases. However, during a repair and compact operation, Access reorders a table's records to put them in primary index order. Depending on how you use data from a table, this could be a significant advantage because data within a range of values will be stored near other data with the same values.

The following is an example of a customer invoice table, showing the data at the end of a day of sales on January 2. Typically, data is entered randomly for customers as they call with new orders and are invoiced for the cost of the materials they purchase.

<i>Record</i>	<i>Cust</i>	<i>Invoice</i>	<i>Date</i>	<i>Amt</i>
1	010	1020	1/1	\$153.00
2	100	1022	1/1	\$760.00
3	219	1023	1/1	\$210.00
4	430	1021	1/1	\$120.00
5	100	1024	1/2	\$150.00

<i>Record</i>	<i>Cust</i>	<i>Invoice</i>	<i>Date</i>	<i>Amt</i>
6	430	1025	1/2	\$550.00
7	010	1026	1/2	\$286.00
8	219	1027	1/2	\$110.00

Most often, this table locates the amount of a specific invoice or determines the total of unpaid invoices for a customer. An index on *Customer*, *Invoice* would allow quick access to the records in the table but would require jumping around in the table to get all the information for one customer. For example, retrieving customer 010 would require records 1 and 7, which aren't next to each other in the table. In a system with thousands of records, the records could be quite far apart, requiring that pages be retrieved in very different areas of the database and, thus, slowing performance.

Generally, you would choose to make the invoice number a primary key on this table because it's a non-null unique key. In this case, you can increase performance by making the primary key on *Customer*, *Invoice* and creating a unique index on the invoice number. After a repair and compact, the actual records in the table are reordered by customer:

<i>Record</i>	<i>Cust</i>	<i>Invoice</i>	<i>Date</i>	<i>Amt</i>
1	010	1020	1/1	\$153.00
2	010	1026	1/2	\$286.00
3	100	1022	1/1	\$760.00
4	100	1024	1/2	\$150.00
5	219	1023	1/1	\$210.00
6	219	1027	1/2	\$110.00
7	430	1021	1/1	\$120.00
8	430	1025	1/2	\$550.00

Because the data is now closer together for a given customer, the caching that takes place when the first record is retrieved for a customer will move several of the customer's records into memory. Access won't need to retrieve the second record for the customer from the disk.

NOTE

If you're wondering why the invoice number is included in the primary key when it isn't required to accomplish the clustering goal, keep in mind that a primary key must be unique. In this situation, the combination of fields is needed to ensure uniqueness in the primary key. In the example, the invoice number is an AutoNumber field and is unique for all records.

Before implementing this technique, you need to understand its disadvantages. Most importantly, the database must be repaired and compacted for records to be reordered. This needs to occur regularly to take full advantage of clustering. As new records are added to the table, they're still added at the end in the order they were entered. The customer data in the example becomes fragmented at the end of the table each day; after several hundred records are added, performance can begin to drop off again.

The other disadvantage is that the primary key no longer represents the real primary key for the table. The primary key in the database schema is still the invoice number. This can be misleading when others look at the database, so it's best to include notes about this type of optimization in the system documentation.

Finally, clustering might actually slow down some of the other queries run against the table. If you have a report that creates a sales analysis listing the total sales for each day and month to date, it might begin to run more slowly because now the data is ordered by customer rather than by date. When the data was in invoice number order, it was very close to date order. Therefore, as you retrieved a record for a given date, you probably read several other records for the same date into cache. This no longer will happen, so you need to carefully analyze the impact on all your queries before implementing this technique.

Working with Read-Ahead

Jet can't have multiple threads reading or writing at the same time. It can have one thread reading and one thread writing, but threads aren't assigned specific tasks.

NOTE

Carefully define your tests and measure your results on a machine you have complete control over. Keep in mind that the operating system might do some caching as well. If you're retrieving data from a remote server, make sure that the server load is consistent and that the server is caching the data the same way for each operation.

You can adjust read-ahead settings in the Windows Registry. Refer to the Access help topic "Customize Jet database engine (ISAM) settings" for the current Registry keys and a description of how to change Registry entries. After you determine where you need to make the changes in the Registry, you can alter the thread setting to allow more read-ahead threads. One application saw a 30% improvement in the performance of a query when this setting was changed from 3 (the default) to 10.

CAUTION

Be very careful when modifying Registry settings. Make sure that you know what you're doing and have backups.

NOTE

The thread setting doesn't appear to change the performance of queries that are resolved purely with Rushmore technology, but it does seem to affect queries that must retrieve non-indexed fields from the table data directly.

Understanding Optimization Techniques

The following sections cover some simple techniques to optimize Access performance. Included are tricks dealing with relationships, indexes, whole database structures, and performance tuning.

Increasing Performance with Table Relationships

To ensure that Access has the greatest possible choice when determining indexes to use, you should create relationships between tables that will be joined. When you set up relationships in Access, not only are you defining referential integrity and cascading operations, but you're also creating indexes on the tables.

Some indexes are exposed when a table's indexes collection is viewed. The `Foreign` property indicates an index created because of a relationship defined in the database. Other internal indexes are created when tables are joined and can be found by searching `MSysObjects` (the hidden system table in which Access stores information on its objects) for type 8 objects, or by looking at `MSysRelationships`.

TIP

Giving Access a wide choice of indexes to resolve query operations increases the chance that Access will choose an index that can provide good performance on large amounts of data.

Adding Indexes

Indexing every field might seem like a good idea, but keep in mind the drawbacks to this approach:

- Adding new records becomes increasingly slower as you add indexes to your table, especially when your table has large numbers of records. This also means more index locks and more multiuser locking issues for the database to manage. As a result, performance will suffer.
- The size of your database grows for each index. With large Access databases, this growth can sometimes be a problem when disk space is limited.
- Creating unnecessary indexes can fragment your base table data more than necessary. The rule of thumb here is to add indexes when they solve an optimization problem. Don't just add indexes to every field and call it an optimization pass.
- You can have only 32 indexes on a table, so use them wisely. In practice, this isn't much of a drawback; rarely will you find the need for more than 32 indexes.

With these restrictions in mind, begin optimization by looking at some preliminary points that set the foundation for optimizing individual queries:

- Build relationships that represent your data and are part of the most common join operations on which the queries will be based. Build a list of the different queries that will be used in the system. Put the list in order of priority: Queries that interactive users will run often and queries that must provide a high level of performance are at the top of the list; reports that are run overnight or when system impact is low are listed at the bottom.
- Go through the prioritized list of queries and begin adding indexes for the fields that are part of join clauses not included in the relationships you set up. Don't add just one index for each field in the join; instead, look at the entire join and the sort fields to determine whether multiple fields are involved. If the join occurs across two fields, you should build one compound index that includes both fields. If one field is used the most for sorting after the join, include it as the first field.
- Add more indexes for single fields and field combinations found in WHERE, GROUP BY, and ORDER BY clauses. Carefully consider how each query will be used. Here are three queries and some good choices for indexes:
 - Customer=100
 - InvoiceDate > #1/1/95#
 - Customer=100 And InvoiceDate > #1/1/95#

NOTE

If you use a compound index, as discussed shortly, you can't optimize this second restriction (`InvoiceDate > #1/1/95#`).

You can optimize the first two queries by putting indexes on `Customer` and `InvoiceDate`. The third query might use one of the two indexes and then scan the table data to apply the rest of the criteria. Jet also can invoke `Rushmore` and query against both indexes, returning only the records found in both index scans.

- Add a compound index on `Customer`, `InvoiceDate` to provide Jet with a faster alternative, and the query would be resolved quickly based on the compound index. If the table isn't used for other queries involving `Customer` and the 32-index allocation is used up, dropping the `Customer` index might be necessary because the compound index can be used for customer retrieval against the table.

Tweaking the Database Structure to Affect Performance

On one of my projects, an Access database served as a referral list to be searched by many different desktop PCs. A main record was linked to more than 15 child tables by using an integer as the join field, and data was displayed from all 15 child tables on forms after the query ran. Some child tables consisted of 10,000 records of detail. A mainframe supplied the data, and users would never update the information directly.

The main requirement was to make the system fast. My first version had a completely normalized schema and was perfect in all ways. The queries ran well, but too many tables were involved in most joins, and I was summarizing the same base table information repeatedly. The users wanted to see detail and were selecting summary information.

After some soul-searching, I summarized many of the tables into two main tables that were built as the data was downloaded from the mainframe. Now, most queries could be answered by joining only two or three tables, which dramatically improved performance. But after query performance improved, the customers began complaining about display performance.

Summarizing was taken one step further, and almost all the display data was stored in the main referral record. This meant creating large memo fields with imbedded carriage returns so that the data could be displayed in text boxes instead of subforms that were linked to the child tables. After I implemented the changes, the final data display didn't require the links to so many tables and therefore ran much more quickly when users navigated from one record to another in the results list.

The moral of this story is to not overlook changing the database's structure when looking for ways to optimize query performance. In this case, the relational design was nearly abandoned in the name of better performance.

Optimizing Join Performance

Join performance can vary widely based on the types and numbers of fields used in the join. Access works best when it can join on small numeric fields. It's also wise to join on fields with the same data type to avoid data conversion during the join operation.

Using Unconventional Optimization Techniques

When you begin with a clean database, optimizing it is easy. Sometimes, dealing with a system that someone else has created takes some creative—or unconventional—optimization techniques. When cleaning up and adding indexes to a table to increase query performance, for example, you might botch performance in importing or executing bulk queries to those same tables.

Understanding Some Performance-Tuning Pitfalls

The biggest mistake in performance tuning is not measuring the gains. Before starting a performance-tuning pass on a database, make sure that an accurate list of processing and response times is available, and then measure the differences after the optimization pass.

Improving performance in one area generally means sacrificing something elsewhere. For example, adding an index speeds up table access based on information in that index. The new index, however, also increases the time it takes to add records to the database. Pay close attention to possible tradeoffs being made during the optimization pass.

In one situation, a form ran very slowly after it was changed to read-only. After some research, I determined that making it read/write allowed the query to run much more quickly. I changed the form back to read/write and locked all its controls to prevent accidental editing of the data.

Diagnosing Slow Queries

Developers who work with Access over a period of time develop a good feel for how soon to expect results from a query. Sometimes queries run much more slowly than they should and need to be investigated. Before you start diagnosing the problem, try resolving it by repairing and compacting a backup copy of the database. More than a few times a few hours have been

wasted trying to solve query problems that were caused by a bad index and easily corrected with a repair and compact. Before you start spending a lot of time diagnosing a problem, try the easy solution.

Next, set a breakpoint in the code and examine the SQL statement in the Debug window. If the SQL statement calls a VBA function, the function is usually removed and returns the raw data. Then eliminate the function during the select process by moving it to code or to a form's control source. If that isn't enough to correct the problem, highlight it and paste it into the SQL view of the query design grid.

Now switch to Design view and run the query. Make sure that the query's performance is really the problem you're looking for, and then proceed with optimizing it.

NOTE

Before switching to Design view, note the fields being selected in the table. If you see `SELECT *`, replace the lone asterisk (*) with `tablename.*` so that it properly shows in the query design grid when you go into Design view.

Look at the join fields and make sure that they're either indexed or have relationships created for them. Look for criteria in the grid and see whether the fields are included in indexes that would help the Jet optimization process.

NOTE

When you work in the query design grid, Jet optimization performance might vary slightly from running VBA code. Run the query from the Immediate window or code and then from the query design grid to ensure that you have similar performance before you begin optimizing. Sometimes it helps to size down all the fields to fit onscreen so that they all display when the query runs.

After you cover all standard optimization techniques, begin removing tables from the query until performance dramatically changes. When the table associated with the problem is identified, research the indexes on that table's fields and the joins being performed. Keep in mind that building compound indexes might help when you're selecting and ordering on different fields in the same table.

After playing with the query a bit, you'll most likely find the root of the problem. At that point, try one of the optimization techniques mentioned earlier in the section "Understanding Optimization Techniques" to help speed up the query.

Resolving Ambiguous Field References

A good way to avoid getting an `Ambiguous field reference` error is to fully qualify all object names in SQL statements. If you encounter errors during query execution, fall back on the reliable query design grid. Use the same method as described in the preceding section for optimizing queries in the query design grid, and you should be able to spot the problems right away.

This technique also works well when you get a `Parameters not supplied` error and can't determine visually which field or table name isn't spelled correctly. Follow the same technique as described earlier in "Understanding Optimization Techniques" and keep eliminating tables and fields until the problem is resolved. Then go back and investigate the last field or table that was removed.

Using the Analyzer Wizards

Although most wizards are for the lower end of the end-user-to-developer spectrum, Access provides Analyzer wizards for the power-user-to-developer spectrum. These wizards can save much time and frustration. You have three choices to work with using these tools: the Table Analyzer wizard tests table structures for normalization, Performance Analyzer tests performance, and Database Documentor prints out system information.

Table Analyzer Wizard

The Table Analyzer wizard helps you normalize your data graphically, including the task of breaking up your structures for you. To use the Table Analyzer wizard, from the Tools menu choose Analyze and then Table.

NOTE

The most normalized database structure isn't always the most optimal. For more information about tuning a database for performance and normalization, refer to the earlier section "Tweaking the Database Structure to Affect Performance."

You can use the Table Analyzer wizard to take data imported from other systems, especially flat files. Use it to get the data into a relational structure and then go from there.

Performance Analyzer

Performance Analyzer examines the different objects in your database and then suggests how to set them up for optimal performance. In the dbQryEmp database, you can use Performance Analyzer by choosing Analyze, Performance from the Tools menu. After the analyzer is opened, choose the objects you want analyzed. Select Query for the Object Type, and then choose Select All. After you select all the objects you want to analyze, click OK.

Although some suggestions are obvious, it's still a good idea to run Performance Analyzer on your applications every so often as you expand them. Highlight any of the suggestions, and then click the Optimize button to have Access perform the optimization(s).

Database Documentor

This wizard helps you document your database by allowing you to print definitions for all the various objects in the database container. Database Documentor doesn't give a cross-reference, however, that tells you which fields are used in which queries (third-party products do this). When you need to get some quick system documentation to the client, however, this works in a pinch to get them started.

Looking at Access 2002's New Query Features

Access 2002 adds some new features for queries. One feature is the choice of being able to specify Access to use ANSI 92 compatible SQL. The other two are new views available for queries: PivotChart and PivotTable.

Using ANSI 92 SQL Mode

Up through Access 2000, Access used ANSI 89 (Jet SQL). Access 2002 lets you have Access use SQL Server Compatible Syntax (ANSI 92) or Access SQL (default). Using SQL Server-compatible syntax makes upsizing to SQL Server that much easier. To change this option, choose Options from the Tools menu, click the Table/Query tab, and then select This Database or Default for new databases under the SQL Server Compatible Syntax (ANSI 92) option.

Viewing Data Using PivotTable and PivotChart Views

One of the most exciting features added to Access 2002 is the ability to view major objects: tables, queries, stored procedures, functions, and forms.

For example, the crosstab query `xtbCategoriesByRating` cross-tabulates the number of movie titles for each category and each rating. (More on this query can be found earlier in the section

“Solving Problems with Queries.”) After opening the query in datasheet view, you can view the query in PivotTable or PivotChart view by using the View menu. Figure 8.41 shows these views of the query.

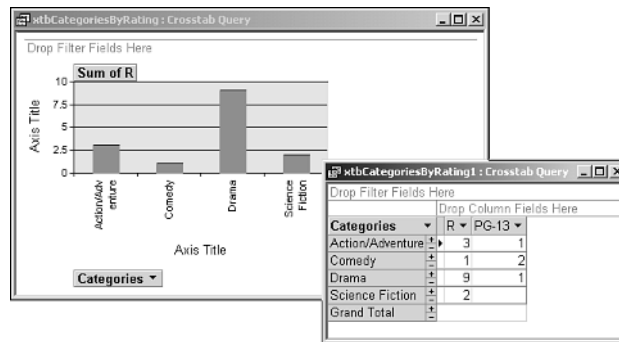


FIGURE 8.41

PivotTable and PivotChart views are now available to queries, giving you more power for viewing data.

With the new views comes programmatic control through properties, methods, and events. You can find more details on PivotTable and PivotChart views in Chapter 9, “Creating Powerful Forms.”

Summary

Setting up and using queries correctly can sometimes make or break an application. Access gives you a number of powerful query types, so you don’t have to go to code to extract information you need. The way data is set up goes a long way to creating optimized queries.

- Chapter 2, “Coding in Access 2002 with VBA,” discusses some of the VBA syntax and how to use the DoCmd object to a greater extent.
- Chapter 5, “Introducing ActiveX Data Objects,” gets into working with queries and ADO.
- Chapter 11, “Creating Powerful Reports,” shows how to use crosstab queries to create dynamic reports.

IN THIS CHAPTER

- **Increasing Form Performance** 242
- **Looking at Access 2002's New Form Features** 243
- **Taking Advantage of Other Form Features** 245
- **Reusing Forms to Perform Standard Tasks** 251
- **Increasing Tabbed Form Performance** 260

One of the most powerful features of Access is the control and flexibility available when creating forms. Microsoft has continued to listen to developers using Access. Control has been enhanced over how forms look and behave—not only at design time, but also at runtime. With Access, you can create good-looking forms to perform tasks that would take reams of code in other languages.

Access gives you more power for managing forms. In this chapter, you get to see these features, as well as see some techniques that help you create impressive forms.

Increasing Form Performance

Although Microsoft has worked on form performance throughout past and current versions, you can still increase performance even more on forms by using ingenuity and VBA:

- If there's no CBF (code behind forms), you can specify a form as lightweight by changing its HasModule setting to No on the Other page of the form's property sheet. HasModule changes automatically from No to Yes when you create event routines.
- Don't place code in OnOpen or OnLoad events if you can help it. You'll get better performance because VBA won't load at the same time.
- When using datasheets in subforms, have the initial record source display zero records. That way, you don't have the populating time at load time of the form.
- Better yet, if you have a multipage tab form, and the page with the datasheet is on a tab that's not used every time, set the subform control's SourceObject at runtime. You can do this in the Tab control's OnChange event, and then just do it when the value of the Tab control equals the page with the subform control on it.
- Use dLookup() functions and other domain functions sparingly.
- Combo boxes, although giving better performance than previous versions, still take time. Be careful of how many you have on the form.
- Make sure that you compile and save all modules before you distribute your application.

NOTE

Some ideas discussed in this chapter will actually increase performance in the user's *perception*, which most of the time is as important as really gaining in performance.

Looking at Access 2002's New Form Features

Access 2002 actually comes through with some useful feature enhancements through new properties and events.

New Base Form Events

NOTE

This section discusses new properties and events for regular forms in standard Access databases (.mdb). For new features in Access Database Projects, check out Chapter 24, “Developing SQL Server Projects Using ADPs.”

Microsoft has added a number of new properties to give you control at the form level. Here are a few:

- `FetchDefaults` allows you to specify whether to display default values on forms.
- `Moveable` specifies whether users can move a form (or report). Even if you specify a form as not being moveable using this property, you can use the `Move` method on the object.
- `Printer` specifies which `Printer` object is assigned to the form. For more information on the `Printer` object, refer to Chapter 4, “Working with Access Collections and Objects.”

The last new event with forms to look at is the `Undo` event. This event fires when you use the `Form` object's `Undo` method.

Using the New PivotTable and PivotChart Views

Another big enhancement in Access 2002 is the ability to open some Access objects in the `PivotTable` and `PivotChart` views, which in Access 2000 were available as `Web` components. These objects include tables, queries, stored procedures, functions, and forms.

Although all the objects mentioned can use these new views, only forms give you some events to go along with them. Some of these events fire when tasks are performed against the form `PivotTable` and `PivotChart` views. Other events can be programmed to execute when Access menus and toolbars commands are performed. Table 9.1 shows the events used.

TABLE 9.1 Events for PivotTable and PivotChart Form Views

<i>Event</i>	<i>Description</i>
<i>Command Events</i>	
CommandEnabled	When a command is enabled
CommandChecked	When a command has been checked
CommandBeforeExecute	Before a command executes
CommandExecute	After the command executes
<i>Change Events</i>	
DataChange	When something has changed that requires a new data object
DataSetChange	When the data set the chart is bound to changes
PivotTableChange	When a PivotTable list field, field set, or total is added or deleted
SelectionChange	Whenever a user makes a new selection
ViewChange	Whenever a new PivotTable “view” is opened
<i>Miscellaneous Events</i>	
Connect	When Access connects the PivotTable to the underlying recordset
Disconnect	When Access disconnects the PivotTable from the underlying recordset
On PivotTable Change	When a PivotTable list field, field set, or total is added or deleted
On Selection Change	Whenever a user makes a new selection
On View Change	Whenever a new PivotTable “view” is opened

I will leave playing with each event up to you. You’ll find them on the forms’ Event tab, along with all the other form events.

Taking Advantage of Other Form Features

The following sections cover some of the more useful features used in this book. You can use these additional features and techniques to make your forms even more powerful. You can find the examples in the following sections on this book's Web page at www.sampublishing.com.

Using the Form Recordset Property

In older Access versions, you could use the `RecordsetClone` property to retrieve a recordset object containing a form's underlying data. This technique is still valid in Access 200x. Unfortunately, changes to a recordset obtained via `RecordsetClone` happen independently of the form itself. For example, record navigation on the form isn't reflected in such a recordset.

As of Access 2000, Access goes a step beyond this by exposing the form's actual recordset—not just a clone—via the `Recordset` property. You can now manipulate the data on a form precisely as you manipulate a recordset in code. What's even better is that this property is read/write—you can build a recordset in code and then hook it up to a form.

CAUTION

When working with an MDB, a form's recordset is *always* a DAO recordset—even when you're using ADO in your database. Failure to take this into account can lead to unexpected errors at runtime.

To see the form `Recordset` property in action, look at the `CustomersRecordset` form and its friend, `CustomersRecordsetRelations`, in Figure 9.1. Both forms can be found in `Chap09.mdb` on this book's Web page at www.sampublishing.com.

Both forms are bound to the `Customers` table. When you click the `Show Relations` button on the `CustomersRecordset` form, the `CustomersRecordsetRelations` form opens to the same record but displays `CustomerRelations` data in a subform.

NOTE

The subform on the `CustomersRecordsetRelations` form takes advantage of Access's capability to bind subforms directly to tables or queries. Its source object is specified as `Table.CustomerRelations`, which causes the appropriate fields from that table to appear in a datasheet.

**FIGURE 9.1**

The Recordset property keeps these forms in sync.

Now actions on these forms are completely in sync. For instance, if you move to the next record on one form, the other form moves to the next record, too. The following code creates this effect:

```
Private Sub cmdShowRelations_Click()
    DoCmd.OpenForm "CustomersRecordsetRelations"
    Set Forms("CustomersRecordsetRelations").Recordset = Me.Recordset
End Sub
```

This routine simply opens the “child” form and sets its Recordset property. Achieving this effect in pre-2000 versions of Access was very difficult.

Using the Dirty Event

Access fires a form-level event when a record is first dirtied in the user interface. (In a database, you *dirty* a record when you make any change to its data.) You can use the Dirty event to do any special processing, and the event can be canceled so that, if necessary, you can deny users the ability to make changes.

The CustomersDirty form demonstrates the Dirty event by adding (Changed) to the window caption (title bar) when a record is first changed. Many Windows applications use this technique to stress to users that there are unsaved changes and they should therefore be careful. You can see this in Figure 9.2.

RelatedName	RelationCode	DOB
T.J. Barker	3	9/19/1978
Linda Barker	2	4/30/1958

FIGURE 9.2

This form's title bar is modified when its data is dirtied.

When the changes are committed, the caption is changed back in the `AfterUpdate` event. Again, the code to do this is very simple:

```
Private Sub Form_AfterUpdate()
    Me.Caption = "Customers"
End Sub

Private Sub Form_Dirty(Cancel As Integer)
    Me.Caption = "Customers (Changed)"
End Sub
```

Specifying a Splash Screen Form at Startup

Access allows you to specify what you want it to do when opening a database. You can have Access open a form of your choice, also referred to as a *splash screen*, by specifying the form's name in the Startup dialog. (Splash screens display company logos or system information while the application is loading.)

To open this dialog, select the database container and then choose Startup from the Tools menu. You'll see a dialog similar to the one in Figure 9.3.

At this time, the setting to emphasize is the Display Form/Page combo box. By setting Display Form/Page to the name of the form you want displayed, you can have the form provide an attractive introduction to your application. This example specifies the `SplashScreen` form, which you can see opened in Figure 9.4.

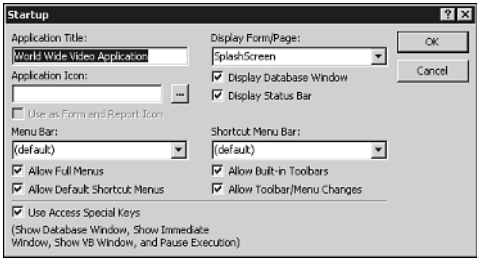


FIGURE 9.3
Use the Startup dialog to specify a splash screen form.



FIGURE 9.4
Splash screens help give your application a professional look.

Using Form Background Properties

In Figure 9.4, notice the attractive bitmap used for the splash screen. Forms have four properties that allow you to create backgrounds (see Table 9.2).

TABLE 9.2 Background Properties to Create Forms

Property	Settings	Description
Picture metafile		Either a bitmap or Windows
Picture Type		Either embedded or linked
Picture Size Mode	Clip Stretch Zoom	Controls how the picture is displayed; same as the Object Frame control

TABLE 9.2 Continued

<i>Property</i>	<i>Settings</i>	<i>Description</i>
Picture Alignment	Top-Left Top-Right Center Bottom-Left Bottom-Right Form Center	Where you want the picture positioned
Picture Tiling	Yes No	Specify whether to tile the form if it doesn't fill the page

You can use the sample metafiles and bitmaps sent with Access, or you can create your own. The background for the splash form in Figure 9.4 is called *Globe.wmf* and is found in the *\Access\Bitmaps\Styles* subfolder.

TIP

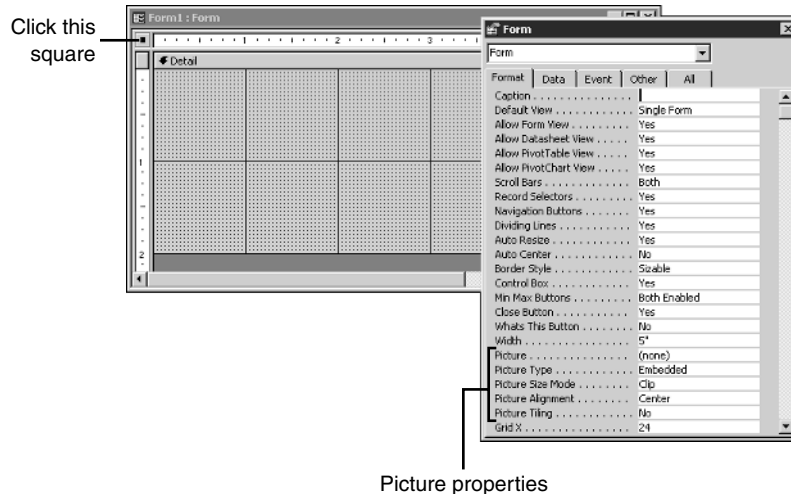
To get better performance out of your form, hold off using a background picture.

To assign the picture you want for a form's background, follow these steps while in the data-base container:

1. From the Insert menu, choose Form to create a new form; or open an existing form in Design mode by choosing the Forms tab, highlighting the form to edit, and clicking Design.
2. Just under the upper-left corner of the form is a square, where the rulers meet. If the property sheet is showing, click the square. If not, double-click the square. With the form's properties showing, the screen will resemble Figure 9.5.

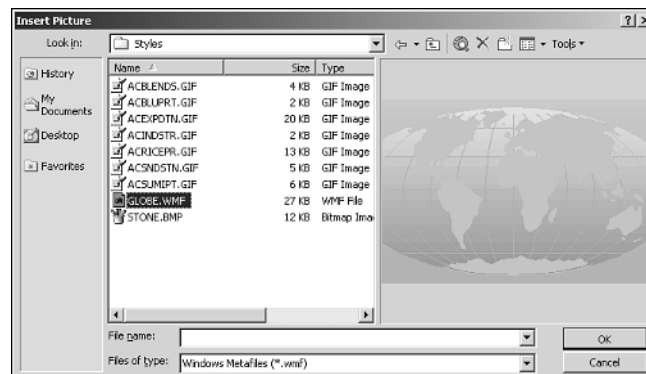
NOTE

After you perform one of the actions in step 2, a black cube appears within the square. When the cube is showing, the property sheet is reflecting the properties of the form itself. If the cube isn't showing, properties of another section or control are being shown.

**FIGURE 9.5**

Double-clicking the square in the form's upper-left corner opens the form's property sheet.

3. Switch to the property sheet's Format page.
4. Place the cursor next to the Picture property. The Builder button, with the ellipsis (...), appears.
5. Click the Builder button. The Insert Picture dialog appears.
6. While in the Access folder, double-click the Bitmaps folder and then double-click Styles. You should see a number of files with the .wmf extension, which denotes a Windows metafile (see Figure 9.6). You'll find Globe.wmf in this folder.

**FIGURE 9.6**

Choose from many of the sample bitmaps included with Access, or create your own.

TIP

If your Insert Picture dialog varies from the one in Figure 9.6, it's because the Preview view of the list was used in this figure. To use Preview view, click the second toolbar button from the right in the dialog.

7. Highlight the graphics file you want and click OK. The file's full path appears in the Picture property. When you move off the property, the word (picture) appears.

You can now play with the other Picture properties to position your graphic on the form.

Reusing Forms to Perform Standard Tasks

When creating applications in a language that's pure code (such as C++), you want to design your code so that you can reuse it for numerous reasons, including the following:

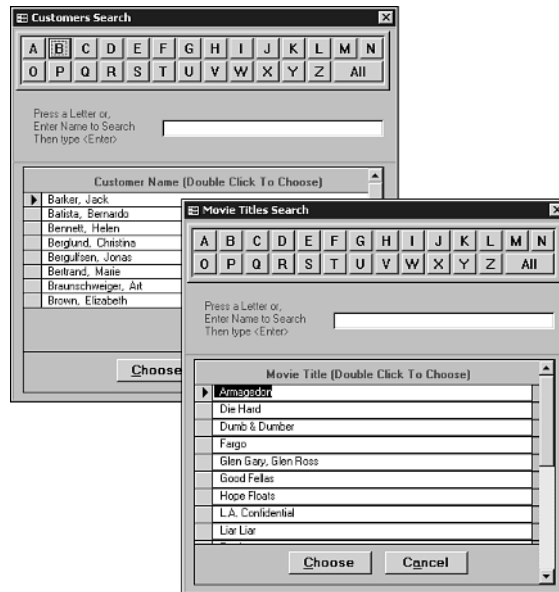
- The main code for the reusable routine must be written only once.
- After the code is debugged, make sure that only the calling routine passes the correct parameters.

These two points are also true when you create reusable code in Access. In Access, you also can create reusable objects. The following example looks at a reusable form and VBA code. The form allows users to search for customers or movie titles, depending on where the search form was called.

NOTE

The database for the rest of this chapter can be found in Chap09.mdb, in this book's section of www.sampublishing.com.

Three forms (StandardSearch, CustomersSearchSubform, and MovieTitlesSearchSubform) make up the standard search form routines. The number of SearchSubform[–named forms will grow as more areas need to have a search form used. In this case, let's stick with the two, Customers and Movie Titles (see Figure 9.7). You'll start off by working with the Customers form.

**FIGURE 9.7**

Look closely at these two search forms. Notice that the caption and the subform are the only things noticeably different.

By using the same form for different subjects, as in looking for customers or movie titles, you're also providing a consistent interface for your user. This is always desirable.

On the calling form side (the form that initiates the opening of another form), look at two properties that Access provides for your purposes: `OpenArgs` and `Tag`.

The first property, `OpenArgs`, can't be accessed from Design mode. This property is set when a form is opened at runtime with the `OpenForm` method, off the `DoCmd` object.

NOTE

Before Access 95, `DoCmd` was a statement that you used for calling macro actions from Access Basic. Since Access 95, you can call the `DoCmd` object to perform the same actions as in prior versions. The `DoCmd` statement is changed to the `DoCmd` object when existing applications are converted to Access versions later than Office 97.

The call to open the search form from the Customers form can be found in the event routine attached to the Search button's `OnClick` event. The routine in Listing 9.1 performs this call.

LISTING 9.1 Chap09.mdb: Opening a Reusable Search Dialog

```

Private Sub Search_Click()
    On Error GoTo Error_Search_Click

    DoCmd.OpenForm "StandardSearch", acNormal, , , acNormal, acDialog, Me.Name

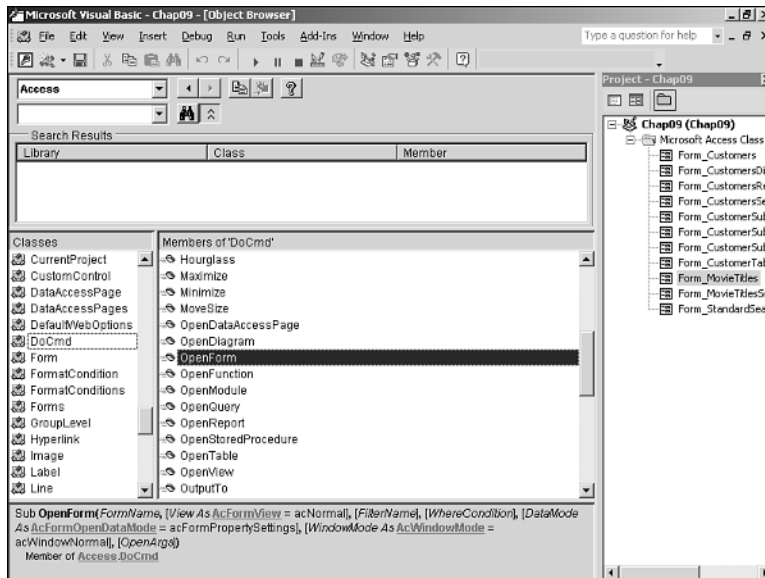
Exit_Search_Click:
    Exit Sub

Error_Search_Click:
    MsgBox Err.Description
    Resume Exit_Search_Click

End Sub

```

This routine will be the same whether it's called from the Customers or Movie Titles form. The part that makes this code so versatile is the last parameter, `Me.Name`, in the `DoCmd` statement, where `Name` is the name property of the current form (`Me`). This parameter is the `OpenArgs` property and can be accessed from the form object being opened. Figure 9.8 shows the syntax for the `OpenForm` method off the `DoCmd` object, using the Object Browser.

**FIGURE 9.8**

The Object Browser is one of the best tools for examining the various objects, methods, and properties.

PART II

Follow these steps to view a complete list of methods of the DoCmd object in the Object Browser:

1. In any module in the VBE, press F2, click the Object Browser toolbar button, or open the View menu and choose Object Browser.
2. Select Access from the Project/Libraries combo box.
3. Select DoCmd from the objects in the Classes list.

You now see the complete list of methods for the DoCmd object in the Object Browser. (For complete information about the Object Browser, see Chapter 2, “Coding in Access 2002 with VBA.”)

By placing Me.Name in OpenArgs, you can copy and paste the Search control and its OnClick event routine into other forms with *no* changes. The only other item you need to worry about setting up on the calling form side is the Tag property.

This property is user-definable, which means that Access pays no attention to it whatsoever, and you get to use it however you want. What you’ll use it for in this example is to keep track of the field on which you want the standard search routine to perform the lookup in the calling form’s recordset. In the Customers form, the key field is CustomerID. Figure 9.9 shows how the Tag property is set.

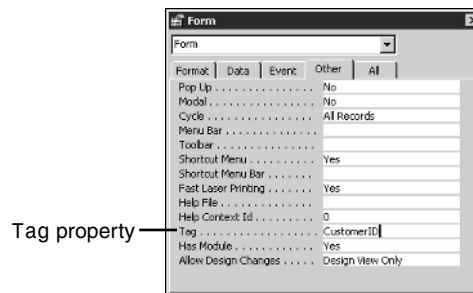


FIGURE 9.9

You can use the Tag property to store any information you want.

You’re now done setting up the calling form. Next, look at what makes up the standard search form, StandardSearch. In looking at the standard search form, you are brought back to the OpenArgs property, which the code in Listing 9.2 uses. In the StandardSearch form’s OnLoad event, set the Caption property to reflect the calling form, and set the SourceObject property for the subSearch subform control.

LISTING 9.2 Chap09.mdb: Setting Form Properties at Runtime

```

Private Sub Form_Load()
    On Error GoTo Error_Form_Load

    Me.Caption = Forms(Me.OpenArgs).Caption & " " & Me.Caption
    Me!subSearch.SourceObject = Me.OpenArgs & "SearchSubform"

Exit_Form_Load:
    Exit Sub

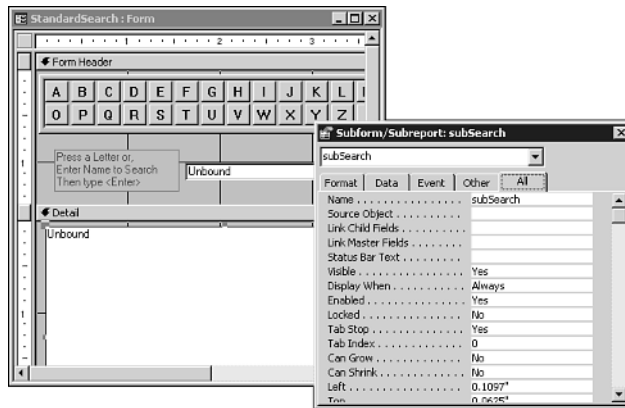
Error_Form_Load:
    MsgBox Err.Description
    Resume Exit_Form_Load

End Sub

```

The `Me.Caption` line concatenates the caption of the calling form (Customers) with a space and the caption of the standard search form.

In the next code line, the `OpenArgs` property of the calling form—in this case, Customers—is concatenated to the literal `SearchSubform`. This creates the complete name of the search subform, `CustomersSearchSubform`. It then assigns that to the `SourceObject` property of the `SearchSubform` subform control. You can see this subform control in Figure 9.10.

**FIGURE 9.10**

Initially, the `SearchSubform` subform control has no form assigned to its `SourceObject` property.

NOTE

The Link Child Fields and Link Master Fields properties aren't used. This means that you have a lot of power because the form's RecordSource property is used as the subform—in this case, CustomerSearchSubform—linked to whatever control you want, if at all. This also allows you to be able to modify this link at any time during runtime. The Link Child Fields and Link Master Fields properties are two of the only ones limited to update only at the OnOpen event of a form.

This form uses two other interesting routines. The first one is attached to the OnClick event for each of the Letter command buttons and the AfterUpdate event of the SearchText text box. You can see the SearchText text box in Figure 9.10 (the Unbound control above the property sheet). The function attached to the OnClick event, ap_Requery_SearchSub(), is shown in Listing 9.3 and is in the StandardSearch form's code behind form.

LISTING 9.3 Chap09.mdb: Requerying the StandardSearch Subform

```
Function ap_Requery_SearchSub()  
    On Error GoTo Error_ap_Requery_SearchSub  
  
    Select Case Screen.ActiveControl.Name  
        Case "txtSearchText"  
            Me.Tag = Me!txtSearchText  
        Case "All"  
            Me.Tag = ""  
            Me!txtSearchText = ""  
        Case Else  
            Me.Tag = Screen.ActiveControl.Caption  
            Me!txtSearchText = ""  
    End Select  
    Me!subSearch.Requery  
  
Exit_ap_Requery_SearchSub:  
    Exit Function  
  
Error_ap_Requery_SearchSub:  
    MsgBox Err.Description  
    Resume Exit_ap_Requery_SearchSub  
  
End Function
```

This routine assigns a value to the StandardSearch form's Tag property based on which control was used, in the following order:

- The txtSearchText control
- The All button
- One of the Letter controls

The routine then requeries the subform control, subSearch. The value in the Tag property is used in the RecordSource property of the subform's SourceObject—in this case, CustomerSearchSubform.

The other routine is also called from the subform. There are three main points of interest concerning the form used as the customer search subform. On the CustomerSearchSubform form (see Figure 9.11), let's look at these points of interest in order:

- The CustomerSearchSubform's RecordSource is set to the following SQL statement:

```
Select CustomerID, FirstName, LastName from Customers where
Customers.LastName like Forms!StandardSearch.Tag & "*"
Order by LastName, FirstName;
```

As you can see, this query refers to the Tag property on the StandardSearch main form. This statement will change to reflect the current subform recordset, but needs to compare the recordset's fields to the standard search form's Tag property. If used by the Movie Titles form, the statement resembles this:

```
Select TitleNo, Title from MovieTitles where MovieTitles.Title
like Forms!StandardSearch.Tag & "*" Order by Title;
```

The key fields are included in the SQL statements, even though they aren't displayed.

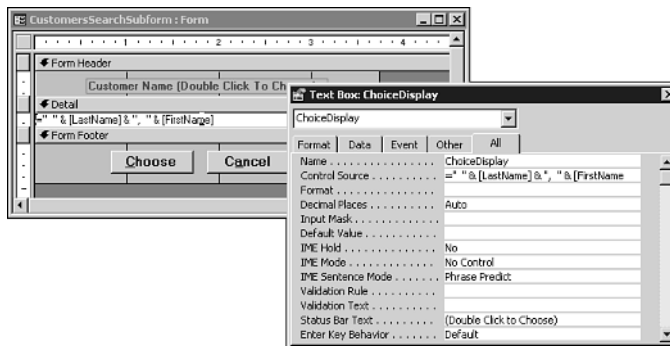


FIGURE 9.11

CustomerSearchSubform displays the LastName and FirstName of customers for users to choose.

- The ChoiceDisplay text box's Control Source property, also shown in Figure 9.11, is set to the expression

```
= " " & [LastName] & ", " & [FirstName]
```

- The ChoiceDisplay text box's OnDb1Click event and the Choose command button's OnClick event both call the same routine from StandardSearch. Here's the event routine for OnDb1Click:

```
Private Sub ChoiceDisplay_Db1Click(Cancel As Integer)
    dummy = Me.Parent.ap_StandardSearchChoice()
End Sub
```

NOTE

You can call routines located in one form from other forms. The preceding code calls the `ap_StandardSearchChoice()` function in the `StandardSearch` form's module. You can find this form in `Chap09.mdb` on this book's Web page at www.sampublishing.com. For more information about calling routines in other forms, see Chapter 2.

Listing 9.4 shows the code for `ap_StandardSearchChoice()`. This routine clones the calling form's recordset, looks up the key value of the chosen value in the clone, and then sets the bookmark of the calling form to that record.

LISTING 9.4 Chap09.mdb: Locating the Record for the Choice Made

```
Public Function ap_StandardSearchChoice()
    Dim frmCalling As Form          '-- Form that called Search Form
    Dim frmSearchSub As Form        '-- Search Form

    On Error GoTo Error_ap_StandardSearchChoice

    Set frmSearchSub = Forms!StandardSearch!subSearch.Form
    Set frmCalling = Forms(frmSearchSub.Parent.OpenArgs)

    '-- Check to see if the search value is numeric or not.
    '-- Search in cloned recordset for chosen value.
    If VarType(frmCalling.RecordsetClone(frmCalling.Tag)) <> vbString Then
        frmCalling.RecordsetClone.FindFirst frmCalling.Tag & " = " & _
            frmSearchSub(frmCalling.Tag)
    Else
        frmCalling.RecordsetClone.FindFirst frmCalling.Tag & " = '" & _
            frmSearchSub(frmCalling.Tag) & "'"
    End If
```

LISTING 9.4 Continued

```
'-- If a record is found, set the bookmark of the calling form.
If frmCalling.RecordsetClone.NoMatch Then
    MsgBox "An Error has occurred, No match record found!", 32, _
        "Search Error!"
Else
    frmCalling.Bookmark = frmCalling.RecordsetClone.Bookmark
End If

DoCmd.Close acForm, "StandardSearch"

Exit_ap_StandardSearchChoice:
Exit Function

Error_ap_StandardSearchChoice:
MsgBox Err.Description, vbCritical, "Search Error"
Resume Exit_ap_StandardSearchChoice

End Function
```

Now that all the code is created, if you want to have the search routine work for the Movie Titles form, you only have to do the following:

1. Click the Forms tab in the database container.
2. Select CustomersSearchSubform.
3. Open the Edit menu and choose Copy.
4. Open the Edit menu and choose Paste.
5. Type **MovieTitlesSearchSubform** for the new form name.
6. Open the new form (MovieTitlesSearchSubform) in Design mode by highlighting it and clicking the Design button.
7. Change MovieTitlesSearchSubform's RecordSource property to reflect the desired data:
Select TitleNo, Title from MovieTitles where MovieTitles.Title
like Forms!StandardSearch.Tag & "*" Order by Title;
8. Change the DisplayChoice control's Control Source property to reflect the new display:
=" " & [Title]
9. Set the Tag property of the Movie Title form to TitleNo.
10. Copy the Search command button and the code attached to the OnClick event routine over to the Movie Title form.

CAUTION

Code isn't copied automatically when you only copy a control from one form to another. You must copy and paste the code in the modules themselves. This can also be a problem if you rename controls after you attach event routines to them. You can re-attach event routines by finding the code and renaming the routine to match the object's new name.

One technique demonstrated in this section was swapping in a form name into a subform's `SourceObject` property. In this case, it was used to substitute the version of the search subform necessary for the subject being sought—Customers or Movie Titles. This technique is also used for swapping in pages used with the Tab control. The following section further discusses this technique.

Increasing Tabbed Form Performance

Before Access 97, most developers created multiple-page forms by simply using the Page Break control and the `GoToPage` method off the Form object.

Since Access 97, you've had the built-in Access native Tab control to use. Although you can put fields and controls directly on all pages of the Tab control, doing so can sometimes bog down form loading. You can also put subforms on some of the pages. The problem when you start getting a number of pages, even using subform controls on them, is that performance can degrade pretty rapidly.

TIP

If you're using a Page Break control with the `GoToPage` method, you can use the `Cycle` property to help control how your control keys (such as Page Down and Page Up) are used. This property also allows you to specify whether to cycle to the next record and/or page when the last field is reached. The settings are as follows:

- **All Records.** When the last field is reached on a form, the cursor continues to the next record.
- **Current Record.** The cursor stays on the current record when the last field is reached and simply cycles around to the first field.
- **Current Page.** The cursor stays on the current page and cycles to the first field on the page. This is useful only when a form has multiple pages.

To enhance performance, bind subforms to their Source Object properties later during runtime. This works especially well for forms that have a good number of pages on the Tab control but don't necessarily use them all that often.

To understand this, look at the `CustomerTabbedFormLateLoadExample` form, which consists of three pages on the `tabInfo` Tab control. The first page has only the data on it so it displays when the form is opened. The other two pages, `Relations` and `Rental History`, load the first time the tabs are chosen for those pages. Notice in Figure 9.12 that the `subCustomer2` subform control's Source Object property isn't set, although Link Master Fields and Link Child Fields are set. The Rental History page is set up basically the same way.

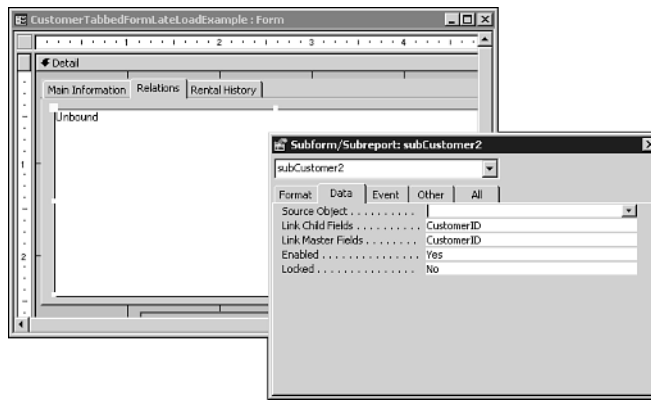


FIGURE 9.12

The SourceObject property is set at runtime for this subform control.

TIP

For subform controls that will have their Source Object properties set at runtime, set the Visible property to False. That way, you can set Source Object first and not have the control flash at all.

The main work for this technique is performed in the Tab control's OnChange event, shown in Listing 9.5.

LISTING 9.5 Chap09.mdb: Loading a Subform Control at Runtime

```
Private Sub tabInfo_Change()  
    Select Case tabInfo  
        Case pgRelation  
            If Len(Me!subCustomer2.SourceObject) = 0 Then  
                Application.Echo True, "Loading Page, Please wait..."  
                Me!subCustomer2.SourceObject = "CustomerSub2Relations"  
                Me!subCustomer2.Visible = True  
                Application.Echo True  
            End If  
        Case pgHistory  
            If Len(Me!subCustomer3.SourceObject) = 0 Then  
                Application.Echo True, "Loading Page, Please wait..."  
                Me!subCustomer3.SourceObject = "CustomerSub3RentalHistory"  
                Me!subCustomer3.Visible = True  
                Application.Echo True  
            End If  
        End Select  
    End Sub
```

For the Relations and Rental History pages, the steps are the same:

1. Test the current value of the tabInfo control against the constants assigned for the pages. (These were assigned in the Declaration section of the form's class module.) Here are the constants listed for you:

```
Const pgMain = 0  
Const pgRelation = 1  
Const pgHistory = 2
```

NOTE

The constants were assigned to make testing for the pages simpler. The constant pgMain isn't actually used for this example.

2. If the tabInfo value matches the page number, the subform's Source Object contained on the page is tested to see whether it's already assigned. If it is, nothing is done. If the Source Object property hasn't been assigned, a message is displayed and the Source Object is set. Then the status bar is cleared.

That's it. The beauty behind this technique is that

- The pages aren't loaded if they aren't used.
- Each page is loaded only once if it is used.

The only downside is that it takes more effort to set up the form because you have to create the subforms, rather than just slap the data on the main form, and then you have to create the code. After you do this on a couple of key forms, you will definitely see a big benefit immediately.

Summary

Access has a number of very powerful form features. By using VBA, along with the Tag and OpenArgs properties, you can create forms that can be reused for different tasks. By using the native Tab control found on Access forms, you can increase performance dramatically with very little effort. The following chapters contain more information about VBA, multiple instances, and the Tab control:

- In Chapter 4, "Working with Access Collections and Objects," you learn how to create multiple instances of any of the Access objects and ways to use this technique.
- Chapter 10, "Expanding the Power of Your Forms with Controls," gives you more information on programming the Tab control.

Expanding the Power of Your Forms with Controls

CHAPTER

10

IN THIS CHAPTER

- **Setting Up a Field's Lookup Properties for Use on Forms** 266
- **Tapping into the Power of Combo Boxes** 268
- **Working with the Access Tab Control** 282
- **Morphing Access Controls** 289
- **Programming Multiselect ListBox Controls** 292
- **Getting Relief with the Subform/Subreport Wizard** 299
- **Giving Controls Spreadsheet-Type Cursor Movements** 300
- **Manipulating Controls Through Code** 304

As with other objects and areas, Access gives you a lot of power to manipulate controls. You can use this power at design time or runtime, via macros and VBA.

Access also gives you a number of versatile controls. The Access toolbox is brimming with various controls that help you do the job. Before learning how to manipulate the different form controls, however, see how Access lets you set up a field that's used for a lookup at the table level.

Setting Up a Field's Lookup Properties for Use on Forms

Access allows you to create the lookup at the table level and have it propagate when you use the field on the various forms, using any type of lookup control that you specify.

You can choose from three control types: TextBox (the default), ListBox, and ComboBox. Figure 10.1 shows an example of a lookup control combo box for the Rating field in the MovieTitles table, which you can find in the Chap10.mdb database on this book's Web page at www.sampublishing.com.

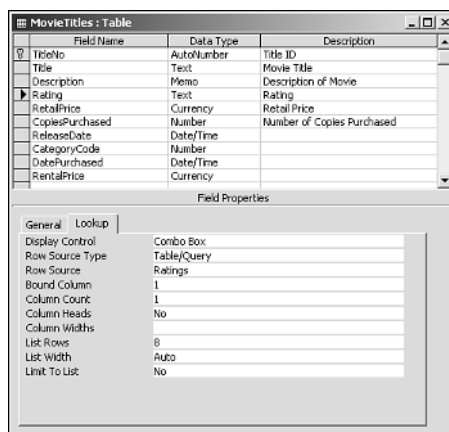


FIGURE 10.1

Setting lookup fields at the table level can save time later in the application cycle.

Be aware of a couple of items when specifying lookup properties at the table field level:

- The Ratings table specified in the Row Source property in Figure 10.1 isn't located in the back end. You can specify tables from the front end as row sources for the lookup field properties of a field in tables on the back end.

- Unlike some of the other properties of a linked table, the lookup properties of a field in the linked table can be set on the front end.

CAUTION

When a linked table has lookup properties set in the front end and then is deleted, even when the table is relinked, the lookup properties are reset. To keep the lookup properties for a linked table, reset and refresh the Connect property. The best place to set lookup properties is on the tables in the back end.

To set the lookup properties at the table level, follow these steps:

1. Open the table you want to add the lookup properties for in Design view.
2. Highlight the field for which you want to specify the lookup properties. The field used for a lookup is usually a foreign key for data in another table—in this case, the Rating field, which is the foreign key for the Ratings table’s primary key.
3. Set the Row Source Type and Row Source just as you do on a form.

After you set the field’s properties at the table level, the control type you chose at the table level is reflected when you place the field on a form. Figure 10.2 shows an example of the Rating field with its lookup properties set at the table level.

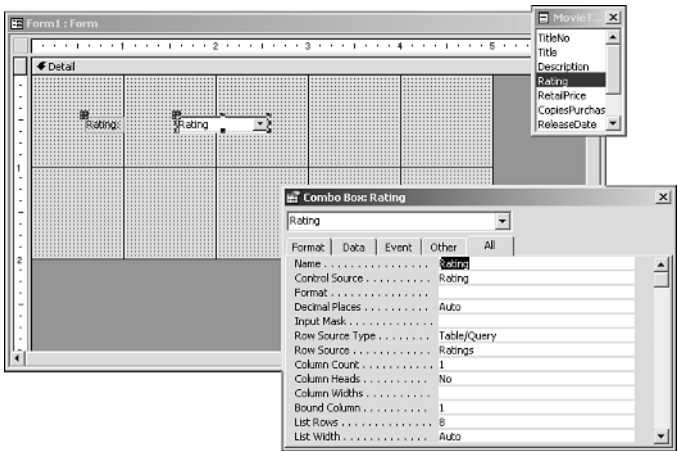


FIGURE 10.2
The lookup properties, when set at table level, propagate to any new forms created.

Tapping into the Power of Combo Boxes

Next to text boxes, combo boxes probably are the most-used control type for data entry. Using them just to look up and display one item of data from another table is fairly straightforward. Access also includes a Combo Box Wizard that makes creating combo boxes even easier.

Using the Combo Box Wizard

Because the Combo Box Wizard is powerful and helps create a rather complex control, it's covered here—unlike most of the other wizards that would be presented in a lower-end book. You'll see how to use this wizard to create a combo box that performs the following tasks:

- Allows users to look for a record on the current form and display one field while searching on another value
- Performs a lookup and updates the foreign key in the current record while displaying another value

To set up a combo box for the first task, follow these steps:

1. Create a form by using the AutoForm: Columnar Wizard, with `MovieTitles` as the base table (see Figure 10.3).

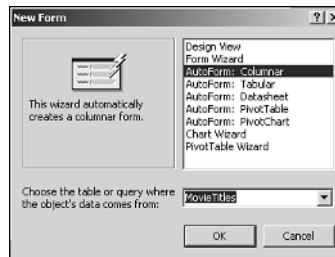
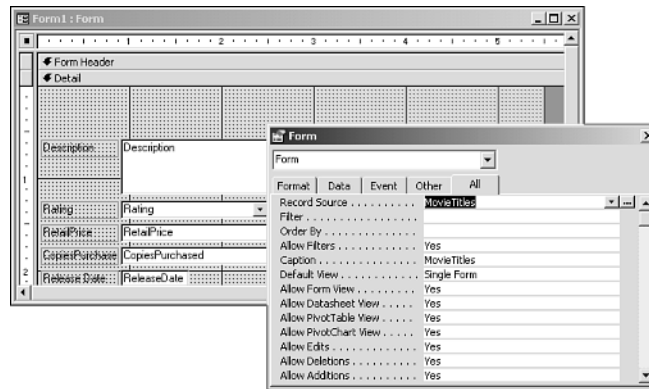


FIGURE 10.3

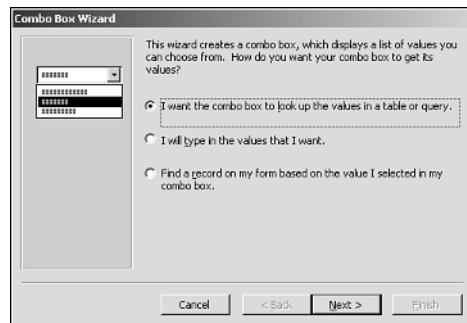
Creating quick forms is easy with all the form wizards available in Access.

2. Switch to the form's Design view by choosing Form Design from the View menu.
3. For this example, delete the ScreenShot, Trailer, TitleNo, and Title fields from the form. (Don't worry, this simply removes the fields from the form, not from the database itself.) Your screen should now resemble Figure 10.4.

**FIGURE 10.4**

This base form is used to add a combo box lookup in the example.

4. Make sure that the Wizard toggle button is toggled so that wizards are turned on. You'll find this button in Form Design view in the toolbox.
5. Click the ComboBox control in the toolbox and drag it onto the form, just above the Description field. The Combo Box Wizard appears, showing the three choices available (see Figure 10.5).

**FIGURE 10.5**

Using the Combo Box Wizard is a great way to jump-start creating a ComboBox control.

NOTE

You're given the first two choices when the combo box is used on a form that has no recordset, called an *unbound form*.

6. Select Find a Record on My Form Based on the Value I Selected in My Combo Box; then click Next.
7. Choose the fields to include in the combo box—in this case, TitleNo and Title. Click Next.
8. At the next dialog, adjust the columns onscreen. This dialog also hides the key column for you. For now, simply click Next.
9. The next dialog lets you change the label of the combo box. However, use the default in this case. Then click Finish.

Now when you change to Form view, you can choose a title from the combo box and have Access automatically go to that record. Figure 10.6 shows the module editor displaying the AfterUpdate event procedure of the Combo20 combo box, which the wizard created.

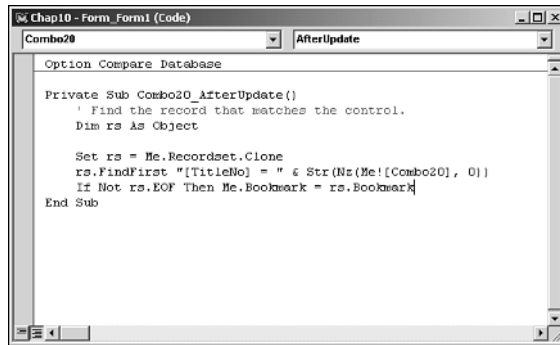


FIGURE 10.6

The Combo Box Wizard automatically created this VBA code for the AfterUpdate event.

Programming Combo Boxes Beyond the Wizard

One weakness of the Combo Box Wizard is that you can't show calculated values in a combo box created through the wizard. For example, when you want to display an employee's first and last names, the Combo Box Wizard produces a combo box with three columns. One column, usually for the last name, appears in the field (see Figure 10.7). The first column is displayed in the list, but the EmployeeID field isn't displayed at all. However, the combo box is bound to that column.

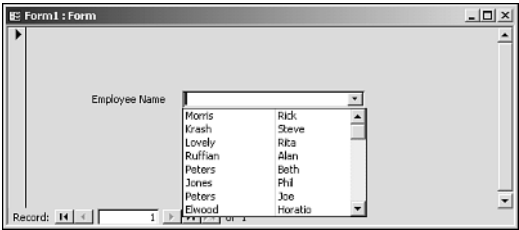


FIGURE 10.7
Although the list displays last and first names, it would be nice to see both in the field.

Changing this combo box to display the last and first names in the field itself isn’t too tough. The trick is to concatenate the two values into one value and change a couple of other properties. You need to change these properties and their current values:

<i>Property</i>	<i>Current Value</i>
Row Source	SELECT DISTINCTROW [Employees].[EmployeeID], [Employees].[LastName], [Employees]. [FirstName] FROM [Employees];
Column Count	3
Column Widths	0";1";1"

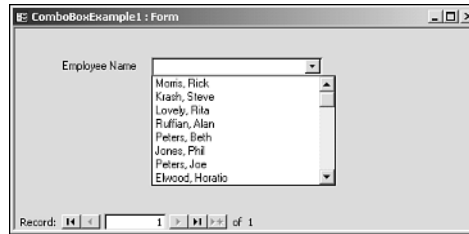
First change the Row Source property’s SELECT statement to display the FirstName and LastName fields in one expression:

```
SELECT DISTINCTROW [Employees].[EmployeeID], [Employees].[LastName] & ", " &  
& [Employees].[FirstName] FROM [Employees];
```

The other two properties also need to be changed to reflect two fields rather than the original three. The following table shows the new property values:

<i>Property</i>	<i>New Value</i>
Row Source	SELECT DISTINCTROW [Employees].[EmployeeID], [Employees].[LastName] & ", " & [Employees].[FirstName] FROM [Employees];
Column Count	2
Column Widths	0";2"

Figure 10.8 shows the combo box using these new values.

**FIGURE 10.8**

The combo box shows the LastName and FirstName fields concatenated, giving the full name in the field.

Using a Union Query to Give the Choice of One or All

Sometimes when you display information in a report or query, you might want to let users show information either for one choice or for all choices. To do so, use the UNION SQL statement in the combo box, along with a query that knows how to handle it. (For more information on queries and using SQL statements, see Chapter 8, “Using Queries to Get the Most Out of Your Data.”)

The following steps create a standard form and query used to view information for a particular piece of data. In this case, the query returns the full record of information found in the Employees table for the employee requested. This is the starting point for showing how to use the union query after the basics are completed.

NOTE

The ComboBoxExample1 form and qComboBoxExample1 query used in this example both are available from Chap10.mdb, which you can find on this book's Web page at www.sampublishing.com.

Follow these steps to create a standard form and query to view information for a particular employee:

1. Name the combo box you created in the preceding section cboEmployeeToQuery.
2. Add a command button named cmdShowInfo and give it the caption Show Information.
3. With the cmdShowInfo command button you just created, open the property sheet and go to the Events page.

4. Create an event procedure for the `OnClick` event on the `cmdShowInfo` button. Then insert the following code line:

```
DoCmd.OpenQuery "qComboBoxExample1"
```

5. Save and close the form with the name `ComboBoxExample1`.

Now create the `qComboBoxExample1` query by following these steps:

1. Create a new query with the `Employees` table as the source.
2. Drag down the `Employees.*` field, and then drag down the `EmployeeID` field.
3. Deselect the `Show` check box in the `EmployeeID` field.
4. In the criteria portion of the `EmployeeID` field, enter the following statement:

```
Forms!ComboBoxExample1!cboEmployeeToQuery
```

Your query should now resemble the one in Figure 10.9. Note that Access inserts brackets if you don't have any spaces in your statement.

5. Save and close the query under the name `qComboBoxExample1`.

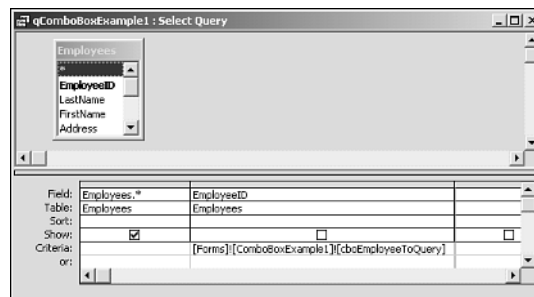


FIGURE 10.9

The `qComboBoxExample1` query now looks directly to the `ComboBoxExample1` form.

Now, by opening the `ComboBoxExample1` form in Form view, you can select an employee from the list and then click the Show Information button to display all available information about that employee.

The preceding steps work fine for one employee, but suppose that you want to see all the information listed for all employees at the same time. To do so, you use the wonderful `UNION SQL` statement. First, copy and paste the two objects from the preceding steps into new objects. Name the new form `ComboBoxExample2` and the new query `qComboBoxExample2`. (Again, this form and query are already created in the `Chap10.mdb` database, which you can find on this book's Web page.)

Next, modify the `ComboBoxExample2` form by following these steps:

1. Open the form in Design view.
2. Edit the `OnClick` event procedure of the `cmdShowInfo` command button to read as follows:
`DoCmd.OpenQuery "qComboBoxExample2"`
3. Close the module editor.
4. Open the property sheet for the `cboEmployeeToQuery` combo box. On the Data page, the current value for the Row Source property should be

```
SELECT DISTINCTROW [Employees].[EmployeeID],  
[Employees].[LastName] & ", " & [Employees].[FirstName]  
FROM [Employees];
```

Here, the process gets tricky. Remember that the `UNION SQL` statement takes the two sources listed and combines them into one list. Next, you'll trick Access into thinking that two sources exist when only one source actually exists—`Employees`. Literal values are used with the `Employees` table.

Add the following statement to the front of the current `SQL` statement:

```
SELECT 0, "<< All Employees >>" FROM Employees UNION
```

Enclose the rest of the statement in parentheses, excluding the semicolon on the end. The resulting `SQL` statement should resemble this:

```
SELECT 0, "<< All Employees >>" FROM Employees UNION (SELECT DISTINCTROW  
[Employees].[EmployeeID], [Employees].[LastName] & ", "& [Employees].[FirstName]  
FROM [Employees]);
```

The reason for using `0` in the `EmployeeID` field is that because you know it's an `AutoIncrement` sequential type field, it will never be `0`, so the query can safely look for `0` as a value to display all the information. Of course, if you use the `AutoIncrement Random` setting for replication purposes, there is a very slim chance one of the entries could be `0`, but it's still very unlikely.

TIP

To make this even more slick, have the form default to the `<< All Employees >>` choice by placing a `0` in the `Default Value` property of the `cboEmployeeToQuery` combo box. This is nice when you want the listing to default to showing all employees.

By entering the preceding statement, when you run the form and click the combo box, a list similar to the one shown in Figure 10.10 appears.

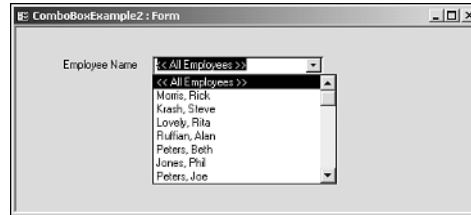


FIGURE 10.10

Here, the combo box gives a choice of viewing all employees' information.

Although the form design looks the way you want it with the combo box displaying << All Employees >> as a choice, your work is only half done. You still need to modify the query to handle the cboEmployeeToQuery combo box correctly in its criteria.

Open the qComboBoxExample1 query and modify the criteria for the EmployeeID field from

```
[Forms]![ComboBoxExample1]![cboEmployeeToQuery]
```

to

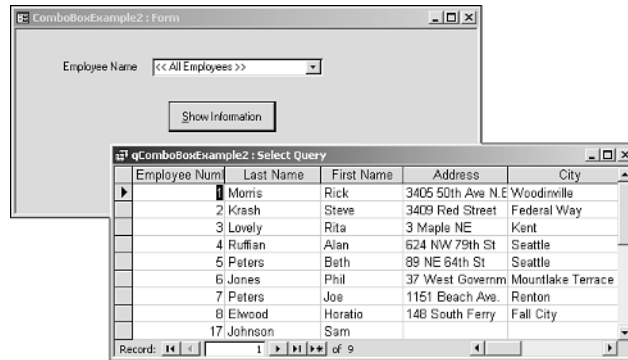
```
IIf([Forms]![ComboBoxExample2]![cboEmployeeToQuery]=  
0,[EmployeeID],[Forms]![ComboBoxExample2]![cboEmployeeToQuery])
```

By using the IIf() function, the [Forms]![ComboBoxExample2]![cboEmployeeToQuery]=0 portion checks whether the << All Employees >> choice, which is 0, has been made from the cboEmployeeToQuery combo box. If it was, the current EmployeeID is compared to itself, which always returns true. Otherwise, the current EmployeeID is compared to the value in the cboEmployeeToQuery combo box, just as in the first example. Figure 10.11 shows the query results, with the << All Employees >> choice made.

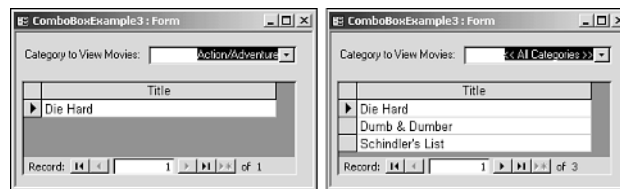
Using a UNION SQL Statement to Requery All in a Subform

In addition to using the UNION SQL statement to open a separate window and display a list for all choices, you can also use it to requery a subform on the same form as the combo box.

Figure 10.12 shows two versions of the same form, with a subform based on a query that uses an IIF() statement similar to the one covered in the preceding section. The left form shows a specific category being entered; the right form shows the entry for All Categories. (As usual, the form used for the example in this section, ComboBoxExample3, can be found in Chap10.mdb on this book's Web page at www.sampublishing.com.)

**FIGURE 10.11**

Here are the query results when choosing << All Employees >>.

**FIGURE 10.12**

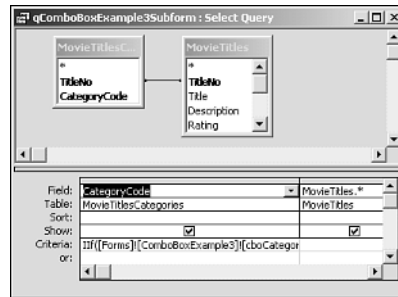
First choose an individual category for a movie (left), and then choose All Categories of movies (right).

You can see the row source for the `cboCategories` combo box in the following code, which looks similar to the SQL code used in the preceding combo box example:

```
SELECT 0, "<< All Categories >>" FROM Categories UNION (SELECT DISTINCTROW
[Categories].[CategoryCode], [Categories].[Description] FROM [Categories]);
```

The query that reads the combo box is also similar but isn't called by a `DoCmd.OpenQuery` method. Instead, it's the record source for the `ComboBoxExample3Subform` subform control.

The subform control itself doesn't use the `LinkChildFields` and `LinkMasterFields` properties, unlike most subform controls. By contrast, the query itself reads the master field directly—in this case, the `cboCategories` combo box. Figure 10.13 shows this query in Design view.

**FIGURE 10.13**

This query, in Design view, reads the cboCategories combo box directly.

NOTE

One reason not to use the `LinkChildFields` and `LinkMasterFields` properties is if the link between the subform control and the main form has to change. Changing the link on-the-fly can sometimes be hard. If you control the link in the query with the criteria, however, it's pretty easy to change the link.

Here's the code for the criteria portion of the `CategoryCode` field:

```
IIf([Forms]![ComboBoxExample3]![cboCategories]=0,[MovieTitlesCategories]![CategoryCode],[Forms]![ComboBoxExample3]![cboCategories])
```

Displaying Combo Box Columns Outside the Control

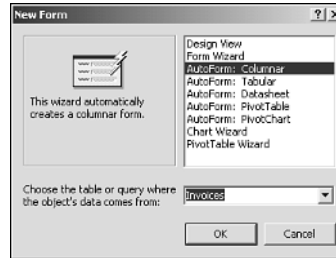
In some cases, you might have information that you can access through a combo box. This saves your application the overhead of joining another table in the underlying query that makes up a form's record source.

Combo boxes (and list boxes) have a `Columns` property that sets an array of the columns you can specify for a combo box. These columns, as with the earlier `CategoryCode` field, don't need to be visible for you to access them. You can access this array either in VBA code behind forms or as a control's source for another control. The latter method is used for this example.

In this example, you use the Combo Box Wizard again because it's the quickest and easiest method. The combo box that's created also passes back other information that, in turn, appears on the form. Follow these steps:

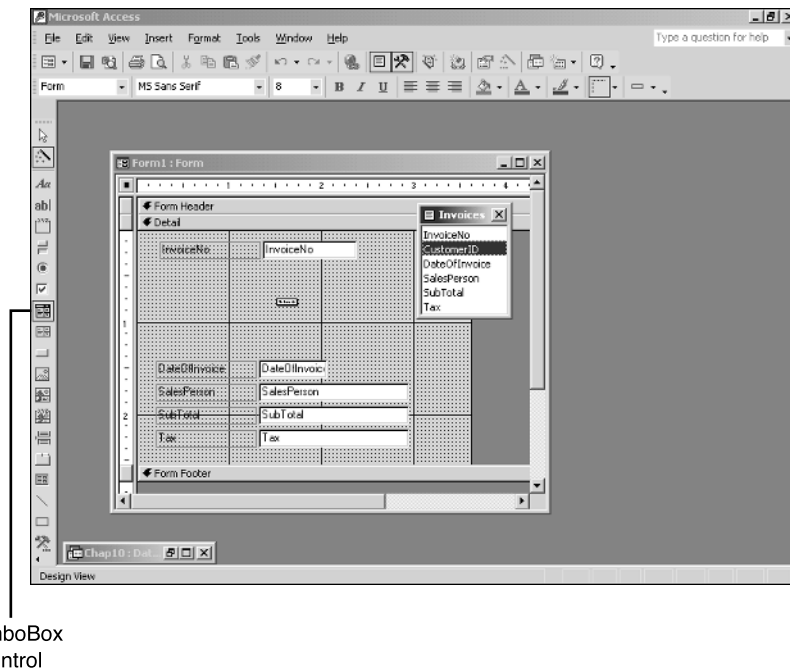
PART II

1. Create a form by using the AutoForm: Columnar Wizard. Select the Invoices table as the record source (see Figure 10.14).

**FIGURE 10.14**

Here's the starting dialog for creating the ComboBoxExample4 form.

2. After the form is created, open it in Design view and delete the CustomerID field.
3. Open the field list by choosing Field List from the View menu.
4. Click the ComboBox control in the toolbox.
5. Click and drag the CustomerID field onto the form (see Figure 10.15).

**FIGURE 10.15**

The ComboBox control is selected, along with the CustomerID field that's being dragged onto the form.

Tip

Use steps 4 and 5 to drag specific control types for fields for other control types in addition to combo boxes. This method works for list boxes as well.

Use the toolbox method only with the field list if you want a nondefault control type. The field list is very smart about the kinds of controls it drops: check boxes for Boolean fields, combo boxes for fields with table-level lookups, and so on.

6. Accept the default choice on the first dialog of the Combo Box Wizard. Click Next.
7. In the next dialog, choose Customers for the table to use and then click Next.
8. Select the CustomerID, LastName, FirstName, and PhoneNo fields and click Next.
9. Drag the right edge of the PhoneNo field to the left to shrink and hide the field (see Figure 10.16). Click Next.

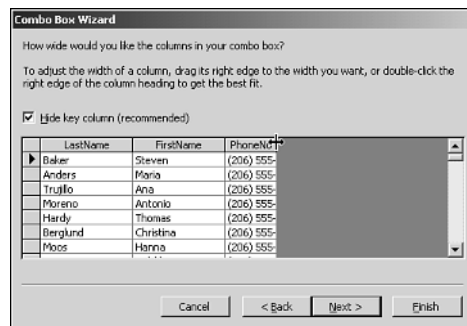


FIGURE 10.16

If you shrink the PhoneNo field, it won't be displayed, but can still be accessed.

10. In the next dialog, simply click Next because the correct field, CustomerID, is already highlighted (because you dragged it from the field list in step 5).
11. Change the label to Customer instead of CustomerID, and then click Finish.
12. Change the name of the new combo box to cboCustomer.

Now if you look at the property sheet for cboCustomer—in particular, the Format properties—you see that the Column Count is 4 and the Column Widths are 0";1";1";0".

You're ready to create a control to display the phone number from this combo box. Create a text box named txtDisplayPhone and set the label to Phone. Then set the Control Source of the txtDisplayPhone text box to =cboCustomer.Column(3). Figure 10.17 shows the completed form in Design view, with the property sheet of txtDisplayPhone showing.

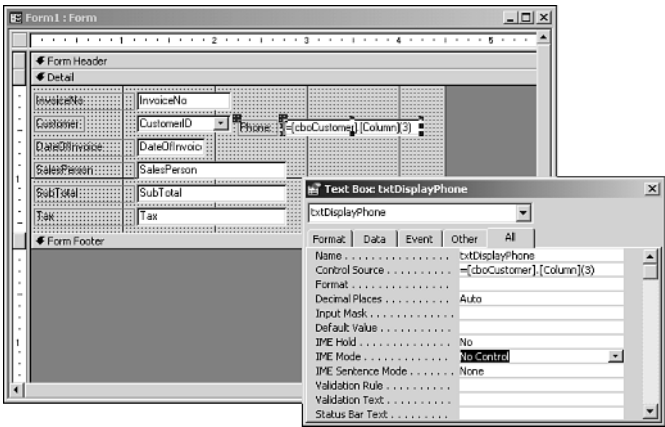


FIGURE 10.17

Using a combo or list box's Column property is another great way to display information from a different table.

TIP

Although the column you want to show (PhoneNo) is the fourth column, because Access objects are zero-based, it ends up being 3.

To prevent users from trying to update the text box displaying the phone number, lock and disable it.

NOTE

As with other methods for displaying information, using combo box columns has a diminishing return on performance. Always try a couple of different ways, such as joined queries and combo box columns, when you want to display information from more than one table.

Adding New Combo Box Items Based on User Input

Frequently, you will want to give users a list of input choices in a combo box but also let them add items that aren't in the list. You can set the combo box's Limit To List property to No, but doing so makes maintaining data integrity more difficult because you have less control over the values entered into the field. Fortunately, Access provides the NotInList event for combo and list boxes to help you deal with this situation.

With the `NotInList` event, you can modify the form in the preceding example to prompt users to enter a new customer record when an unknown name is entered into an invoice. The code in Listing 10.1 handles the `NotInList` event for the `cboCustomerID` combo box.

LISTING 10.1 Chap10.mdb: Handling a Customer Not in the List

```
Private Sub cboCustomerID_NotInList(NewData As String, Response As Integer)
    Dim intReply As Integer

    intReply = MsgBox("The Customer '" & NewData & _
        "' is not in the list. Would you like to add?", vbYesNo)

    If intReply = vbYes Then
        '-- Open CustomersForm in data entry mode, passing the new data as
        '-- an OpenArg
        DoCmd.OpenForm "frmCustomers", , , , acFormAdd, acDialog, NewData
        '-- Record added, so cancel Access's default processing
        Response = acDataErrAdded
    Else
        MsgBox "Please select an item in the list."
        '-- We have handled the error, so tell Access not to put up
        '-- its own default error.
        Response = acDataErrContinue
    End If
End Sub
```

This code tells users that the customer isn't in the list and asks whether they want to add the customer. If so, it opens the `CustomersForm` form as a dialog in Add view so that they can fill in the new customer record.

NOTE

The `NewData` event argument contains the string that users entered, which isn't in the list. It's passed as the `OpenArgs` parameter to the `OpenForm` method so that `CustomersForm` can fill in its `LastName` field, making it faster and easier for users to add the new record. For more information on using `OpenArgs`, refer to Chapter 9, "Creating Powerful Forms."

Finally, the `Response` argument is set depending on the outcome of the event processing. The following table shows the possible settings for `Response`, the situations in which you should use them, and Access's corresponding actions.

<i>Constant</i>	<i>Usage</i>
<code>acDataErrAdded</code>	The item was added, averting the not-in-list condition. No errors are displayed.
<code>acDataErrContinue</code>	The item wasn't added. No default error is displayed.
<code>acDataErrDisplay</code> (default)	The item wasn't added. The default Access error is displayed.

The code in Listing 10.1 sets `Response` to `acDataErrAdded` after the `CustomersForm` is closed, indicating that the customer is added. When users choose not to add the customer, a custom message is displayed and `Response` is set to `acDataErrContinue` to prevent Access from displaying its default message.

The resulting form can be found as `ComboBoxExample5` in `Chap10.mdb` on this book's Web page at www.sampublishing.com.

Working with the Access Tab Control

Tabbed forms are very attractive and add a new dimension to your application. They allow you to display multiple pages of information without using either multiple forms or command buttons to switch between pages.

Some solutions have been provided by ActiveX control solutions. The problems with the ActiveX solutions provided thus far, however, are that

- They need to be included with the application when distributing them.
- They take up memory.
- They can cause performance degradation.
- Registration problems can occur.
- Depending on where you get them, they might cost you money.
- Code is needed to control paging.

Some alternative utilities have been provided, but sometimes can be cumbersome to modify and maintain. However, Access provides its own Tab control to take care of all these situations.

The Access Tab control includes the following capabilities:

- Displaying multiple rows of tabs
- Including bitmap images for each tab
- Programmatically accessing the tab pages collection

The Access Tab control is very quick and isn't hard to use after you figure it out. The example in this discussion is the same basic customer form as in Chapter 9. Figure 10.18 shows this

form, `TabControlExample1`, in Design view. As usual, you can find this form in `Chap10.mdb` on this book's Web page at www.sampublishing.com.

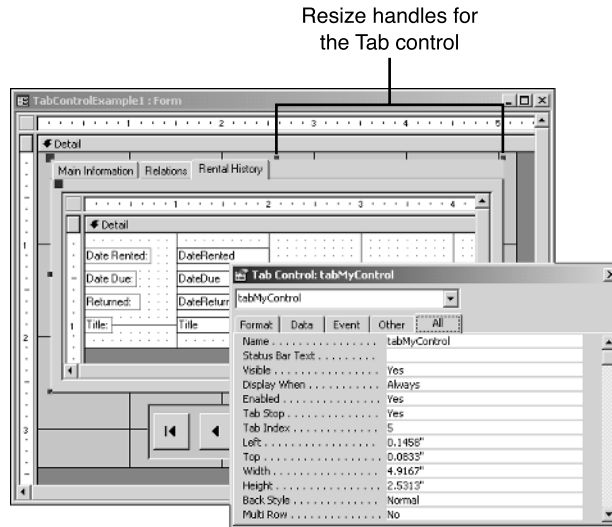


FIGURE 10.18

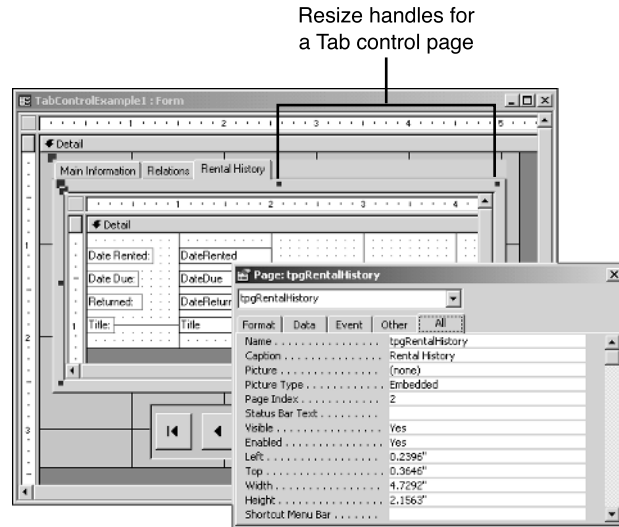
This Tab control is included in the toolbox.

In Figure 10.18, you can see that the Tab control, rather than the individual pages, is highlighted. This is shown in two ways:

- The most obvious is that the property sheet displays the words **Tab Control** on its title bar, along with the name of this Tab control. If a page instead is highlighted, the property sheet says the word **Page** and the name that you gave the page.
- Look at the handles for controlling the height of the control. They're above the tabs. If a page is highlighted, the sizing handles appear below the tabs, as in Figure 10.19.

Make sure that the tab you want to place the controls on is in the foreground, shown in Figure 10.18 with the **Rental History** tab.

Placing controls on the tab pages is easy—you do it in the same way as on the main form, only on the individual pages. If you click the various tabs found on the `TabControlExample1` form, you can see how they work.

**FIGURE 10.19**

Watch the sizing handles and property sheet name to see which page you're on in the Tab control.

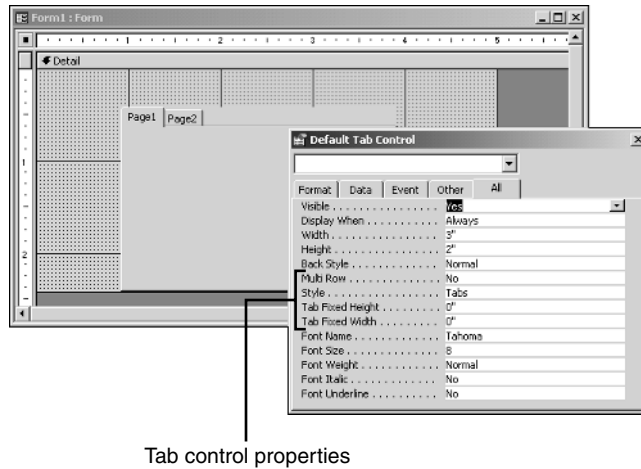
Creating and Editing a New Tab Control

To start working with the Tab control, create a form that's similar tab-wise to the TabControlExample1 form. You even borrow the subforms. The first set of steps creates a Tab control, inserts a new tab, and then tells you how to set the right caption on the tabs:

1. Create a new blank form.
2. In the New Form dialog, leave Design View as the choice to create the form with and pick the Customers table to base the form on. Click OK. You now have a blank form in Design view.
3. Click the Tab control in the toolbox. Figure 10.20 shows a Tab control already placed on the form.

TIP

Notice that the property sheet says *Default Tab Control*. By displaying the property sheet and then clicking the specific control, you can set default values for that control type. This way, you can set up defaults for the different control types for this form. After you set them here, the properties reflect the settings whenever a new control of this type is placed on the form.

**FIGURE 10.20**

The Tab control begins with two tab pages when you first place it on a form.

NOTE

The four properties used by the Tab control, as you can see in the property sheet, include Multi Row, used for specifying using multiple row tabs, and Style, which lets you use tabs, commands, or nothing to switch pages on the Tab control. The other two properties deal with the size and height of the tabs.

4. Right-click the Tab control. On the shortcut menu is a section for the Tab control itself: Insert Page, Delete Page, and Page Order. The last choice grouped with these commands, Tab Order, is actually for the whole form.
5. Choose Insert Page. You now have three pages on the tab form.
6. Change the name of each tab via the Caption and Name properties. Set the Caption property for the first tab (to the left) to **Main Information** and the Name property to **tpgMainInformation**.
7. Click the second (middle) tab. Type **Relations** in the Caption property for this tab and make the Name property **tpgRelations**.
8. Click the third tab (to the right). Type **Rental History** for the Caption property and make the Name property **tpgRentalHistory**.

You've now set all the tabs to reflect the pages of information you want them to hold.

Moving Pages in the Tab Control

If you decide that you need the Relations page last and the Rental History page second, at this point you can simply change the Caption property for each page. However, if you have data on the pages, you have to cut and paste the controls on each page. Rather than deal with this hassle, however, you can have Access take care of it for you.

Highlight the Tab control or one of its pages to show the resize handles. Then right-click and choose Page Order. In the Page Order dialog (see Figure 10.21), highlight the Relations page and click Move Down. Click OK to show the page change in the Tab control and to update each page's Page Index property. All controls are also carried with the moved page.

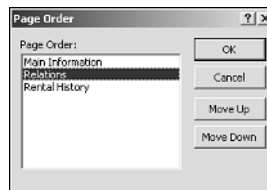


FIGURE 10.21

Using the Page Order dialog makes moving tab pages a snap.

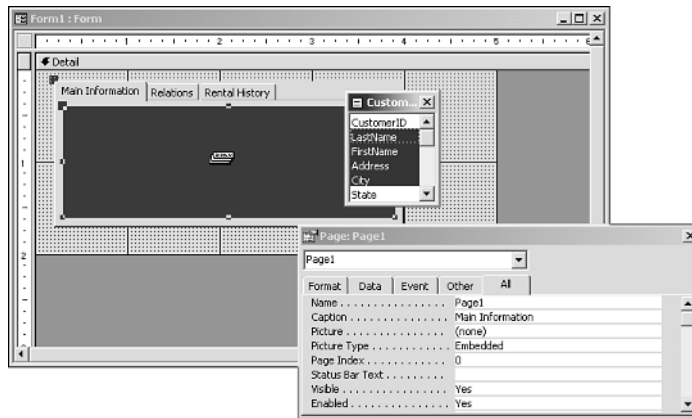
TIP

The way these tab pages are named, it doesn't matter that they're reordered. If the Name properties of the pages are left as is (Page3, Page2, and so on), they would be totally out of order and confusing. By giving the pages more topical names, you don't care what the order is when you address them directly by their names.

Adding Controls to the Tab Pages

Now it's time to add controls to the Tab control pages you've created. To get ready for this, choose Field List from the View menu. Then click the Main Information tab so that the Name property reflects tpgMainInformation in the property sheet.

From the field list, drag LastName, FirstName, and Address onto the tpgMainInformation page of the Tab control. When you have the fields over the tab page, Access reflects which page has the focus by shading the page (see Figure 10.22). This is similar to what happens when you add a radio, toggle, or check box to an Option Group control.

**FIGURE 10.22**

The selected fields are placed on the first page of the Tab control.

Now click the Relations tab. Notice that the Tab control looks cleared because a new, empty page is displayed. (If you don't believe me, click the Main Information tab and the fields you just placed reappear. Then click the Relations tab again.)

To place a subform on the Relations page of the Tab control, follow these steps:

1. Make sure that the Wizard button is turned off in the toolbox.
2. Click the Subform control and draw out a subform. Notice that the tab page area is shaded again. You now have an unbound Subform control on the Relations page.
3. Use the property sheet's Data page to set the Source Object property to the TabControlExampleSubTabPage2 form. The Link Child Fields and Link Master Fields properties are set automatically to CustomerID.

You've set up two of the tab pages. To set up the Rental History page, follow the steps just given, but set the subform's Source Object property to the TabControlExampleSubTabPage3 form.

Using Code with the Access Tab Control

Whereas code is necessary to switch pages with the various alternatives to the Tab control, none is necessary with the Tab control itself. However, sometimes you might want to change properties of the Tab control and individual pages on-the-fly.

The following code listings are on command buttons located on the TabControlExample2 form (a copy of the TabControlExample1 form, except with additional command buttons). The first code example, in Listing 10.2, merely goes through the Pages collection, which is included in the Tab control, and prints the caption to the Immediate window.

LISTING 10.2 Chap10.mdb: Printing the Caption Property for Each Tab Control Page

```
Private Sub cmdPrintCaptions_Click()  
    Dim tabMyControl As TabControl  
    Dim pgCurrTabPage As Page  
    Set tabMyControl = Me!TabCtl0  
  
    For Each pgCurrTabPage In tabMyControl.Pages  
        Debug.Print pgCurrTabPage.Caption  
    Next pgCurrTabPage  
  
End Sub
```

This code uses a `For . . . Each` statement to iterate through the `Pages` collection. You must have the Immediate window open to see the values.

The last piece of code that manipulates the Tab control actually switches the Relations and Rental History tab pages (see Listing 10.3).

LISTING 10.3 Chap10.mdb: Switching the Order of Pages in the Tab Control

```
Private Sub cmdSwitchTabOrder_Click()  
    Dim tabMyControl As TabControl  
    Dim pgCurrTabPage As Page  
    Set tabMyControl = Me!TabCtl0  
  
    For Each pgCurrTabPage In tabMyControl.Pages  
        If pgCurrTabPage.Name = "tpgRelations" Then  
            If pgCurrTabPage.PageIndex = 1 Then  
                pgCurrTabPage.PageIndex = 2  
            Else  
                pgCurrTabPage.PageIndex = 1  
            End If  
        Exit For  
    End If  
Next pgCurrTabPage  
  
End Sub
```

As with Listing 10.2, the code in Listing 10.3 iterates through the Tab control's `Pages` collection. However, rather than simply print the `Caption` property, this code checks whether the `Name` property contains the `tpgRelations` string. If it does, it toggles the `Page Index` property to either 1 or 2, thus switching the Relations and Rental History tabs.

NOTE

As with other Access collections, the Pages collection is based at 0, so page 1 is 0, page 2 is 1, and so on.

You also occasionally might want a control appearing on all tabs, such as a close button or your record navigation buttons. What you can do is to actually put those controls not on a particular tab, but on the form itself and then overlay them over the tabform. Now you have the appearance that each tab has this control. (Code can determine the which tab is being shown, to take appropriate actions.)

This should get you started creating tabbed forms using the Access Tab control. After you play with it for a while, I'm sure you'll find it as easy to use as most of the other controls.

Morphing Access Controls

As sometimes happens when you develop an application, you (or the user) might change your mind about what kind of control you want to use to represent a particular field. Access has a way of dealing with this situation. You can morph controls either through the user interface (UI) at form design time with Access menus, or in code using VBA at runtime.

Morphing Controls at Design Time

Changing controls from one type to another at design time, also called *morphing*, is simple. For example, open the `ComboBoxExample1` form in Design view. Now highlight the `cboEmployeeToQuery` combo box and then choose `Change To` from the `Format` menu.

Notice that because a combo box is being changed, the only choices are `Text Box` and `List Box`. For the most part, it makes no sense to try to change a combo box into a label or toggle button. Various items are enabled and disabled, depending on the type of control being changed.

Morphing Controls with VBA at Runtime

Changing control types for a field at runtime takes a bit more work. By controlling morphing programmatically, you can display one type of control for one purpose and a different type of control for a different purpose, all by using the same form and field. To do so, you need to change the control's `ControlType` property. But first, open the form in Design view. If the form isn't open, you can open it by using the following command:

```
DoCmd.OpenForm "NameOfForm", acDesign, , , , acHidden
```

This code opens the form in Design and Hidden views. You can then change the type of the control you want by assigning the new type to the `ControlType` property, as in the following syntax:

```
Forms!NameOfForm!NameOfControl1.ControlType = ControlTypeConstant
```

Replace *ControlTypeConstant* with one of the constants in Table 10.1.

TABLE 10.1 Control Type Constants

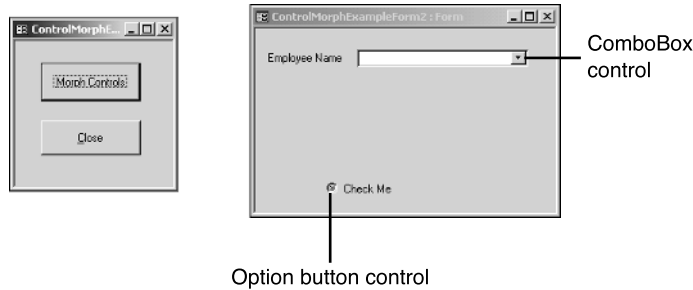
<i>Constant</i>	<i>Control</i>	<i>Constant</i>	<i>Control</i>
acLabel1	Label	acTextBox	Text box
acImage	Image	acListBox	List box
acCommandButton	Command button	acComboBox	Combo box
acOptionButton	Option button	acCustomControl	Custo control
acCheckBox	Check box	acToggleButton	Toggle button

NOTE

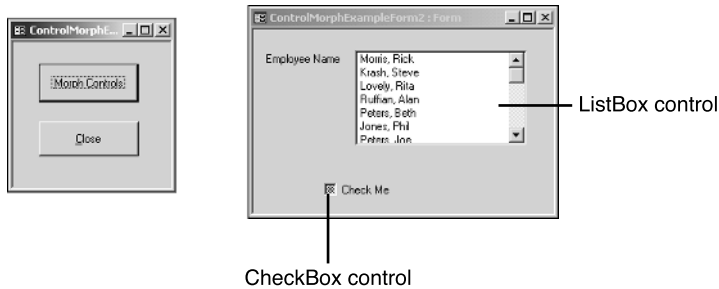
Just as when you use menus to change control types at design time, at runtime not all types can be changed to other types, and the types that can change are limited in what type they can be changed to. To find out whether one type can be changed to another, test it out by using the menus at design time.

Chap10.mdb, found on this book's Web page at www.sampublishing.com, uses two forms to show how to morph controls on another form while the form is open:

- `ControlMorphExampleForm1` has two command buttons: one with code behind it for changing the types of the two controls on the second form, and one that simply closes the form.
- `ControlMorphExampleForm2` can hold different types of controls, depending on whether the routine from the first form is run on it. It has either a combo box and an option button on it, or a list box and a check box. Figures 10.23 and 10.24 show these two possibilities.

**FIGURE 10.23**

There's nothing up your sleeve as the form starts out with a combo box and an option button.

**FIGURE 10.24**

Click the magic button and voilà—the form now has a list box and a check box.

The code to do the morphing is on the cmdPerformMorph command button on the ControlMorphExampleForm1 form. Listing 10.4 shows the code attached to the OnClick event. Notice that the constants are used for testing and assigning the control types to the ControlType property.

LISTING 10.4 Chap10.mdb: Morphing a Combo Box/Option Button into a List Box/Check Box

```
Private Sub cmdPerformMorph_Click()
    On Error Resume Next

    DoCmd.Echo False, "Morphing controls, please wait..."
    DoCmd.SelectObject acForm, "ControlMorphExampleForm2"

    '--- Switch to Design Mode
    DoCmd.DoMenuItem acFormBar, 2, 0
```


LISTING 10.4 Continued

```
If Forms!ControlMorphExampleForm2!cboEmployeeToQuery.ControlType _  
    = acListBox Then  
    Forms!ControlMorphExampleForm2!cboEmployeeToQuery.ControlType = acComboBox  
Else  
    Forms!ControlMorphExampleForm2!cboEmployeeToQuery.ControlType = acListBox  
End If  
  
If Forms!ControlMorphExampleForm2!optMorphing.ControlType _  
    = acOptionButton Then  
    Forms!ControlMorphExampleForm2!optMorphing.ControlType = acCheckBox  
Else  
    Forms!ControlMorphExampleForm2!optMorphing.ControlType = acOptionButton  
End If  
  
'-- Switch back to form mode  
DoCmd.DoMenuItem acFormBar, 2, 1  
DoCmd.SelectObject acForm, "ControlMorphExampleForm1"  
DoCmd.Echo True  
  
End Sub
```

After changing the control type through the UI and through code, you might have to set other properties, depending on which type the control was changed to. Otherwise, Access sets the controls to the default settings for the new control type.

Programming Multiselect ListBox Controls

In addition to giving the same capability to do lookups that combo boxes give, the Access ListBox control allows you to do the following:

- Display lists so that the user can see more than one—if not all—of the choices onscreen
- Have users select more than one choice from the list and keep track of them so that you can use the choices in a task

NOTE

All examples given in the earlier sections for combo boxes also can be used with list boxes.

Using list boxes in the standard way takes no real programming; using them for multiselection, however, requires some code to read them.

List Box Properties Dealing with Multiple Selection

When working with list boxes in multiselect mode, you can use certain properties. Table 10.2 lists properties used when dealing with list boxes in multiselect mode. These properties can be accessed through code to manipulate the selected items.

TABLE 10.2 List Box Properties and Their Functions

<i>Property</i>	<i>Description</i>
MultiSelect	Can be set to None, Simple, or Extended. When it's set to Simple, you can select and deselect multiple items in a list box by clicking the items or by pressing the spacebar. When it's set to Extended, Shift+clicking or pressing Shift+arrow extends the selection from the previously selected item to the current item. Ctrl+clicking an item selects or deselects that item.
ItemData	Contains the bound column in an array. The order of the array elements is the same as those in the list box.
Selected	An array that reflects the selected status of items in a list box. The property is set to 0 if the current item isn't selected, or -1 if the current item in the array is selected. To look at a specific array element, the syntax is <code>ListboxControlName.Selected(rownumber)</code> . To get just the selected items, see <code>ItemsSelected</code> .
ItemsSelected	A collection of the selected items, including any other columns specified for the list box. When you access it through VBA, you can use the <code>ItemsSelected</code> collection, which retains the properties, as does the main list box. You can also access the <code>Columns</code> array from individual elements.

NOTE

The arrays and collections, such as the `ItemsSelected` collection used by the `ListBox` object, use a base 0, as do other Access collections. If a list box contains three items, the index is 0 through 2.

Manipulating Items Selected in a Multiselect List Box with VBA

As in most programming with VBA and Access, the trick to understanding how to manipulate multiselect list boxes in VBA is to understand the properties and collections—and to have a simple example.

A Simple Example for Getting Selected Items

In the Chap10.mdb database is the MultiSelectListBoxExample1 form, on which you'll find two controls: a list box and a text box. When this form is run, the lboEmployeeToChoose list box is filled with the last and first names of employees (see Figure 10.25). When you click a name in the list box, it appears in the txtNamesChosen text box.

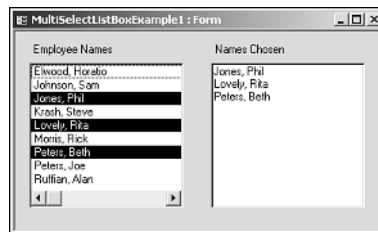


FIGURE 10.25

This form is an example of using VBA to manipulate a list box with its MultiSelect property set to Simple.

The row source for the lboEmployeeToChoose list box is the following SQL statement:

```
SELECT DISTINCTROW [Employees].[EmployeeID], [Employees].[LastName]& ", " &
[Employees].[FirstName] FROM [Employees] order by LastName, FirstName;
```

In addition to having the MultiSelect property set to Simple, the real power behind this form is the Control Source for the txtNamesChosen text box:

```
=ShowNamesChosen()
```

You can find the ShowNamesChosen() function—shown in its entirety in Listing 10.5—in the general routines section of the MultiSelectListBoxExample1 form module.

LISTING 10.5 Chap10.mdb: Displaying the Current Names Chosen

```
Function ShowNamesChosen() As String
    Dim varEmployee As Variant
    Dim strTemp As String
```

LISTING 10.5 Continued

```
'-- for each of the items in the ItemsSelected collection
For Each varEmployee In Me!lboEmployeeToChoose.ItemsSelected()
    '-- If not the first item, put a carriage return line feed
    '-- in front of the item
    If Len(strTemp) <> 0 Then
        strTemp = strTemp & vbCrLf
    End If
    '-- Grab the second column over of the current item
    strTemp = strTemp & Me!lboEmployeeToChoose.Column(1, varEmployee)
Next varEmployee

'-- Assign the final string pass it back
ShowNamesChosen = strTemp
```

End Function

The last example is fairly straightforward: it simply shows how to iterate through the list box's `ItemsSelected` collection. This is done with the following code:

```
For Each varEmployee In Me!lboEmployeeToChoose.ItemsSelected()
```

Notice that in the following line of code, the second column is the one being concatenated to the text box:

```
strTemp = strTemp & Me!lboEmployeeToChoose.Column(1, varEmployee)
```

Finally, for the `txtNamesChosen` text box to be recalculated, you need to put the following code line in the list box's `AfterUpdate` event:

```
Me.Recalc
```

Although this example is useful for showing how to access the `ItemsSelected` collection, it shows only one aspect of manipulating a multiselect list box. It doesn't show the whole picture of setting preselected items in a list box or saving selected items to a table. The following example does this very task.

Example for Getting/Setting Selected Items from/to a Table

There are a number of reasons for needing routines for storing selected items to a table, reading them from that table later, and then preselecting the items back in the same list box. Some examples might include choosing

- The reports that the user wants to print at the end of day
- The tasks that must be performed before another can be marked as completed
- The tables that should be chosen for a particular task, such as exporting

The last task is what the following example, found on the `MultiSelectListBoxExample2` form, is based on. This example uses two tables: `TablesInSystem` and `TablesChosen`. Both tables contain one field, `TableName`. For the sake of simplicity, this example performs three tasks:

- It compares the current tables listed in `TablesChosen` with the `lboTableToChoose` list box and marks particular table names as selected in the list box, if they exist in `TablesChosen`.
- It lets users select or deselect tables from the list box and allows the display of them in the `txtTablesChosen` text box.
- It allows users to save the new set of selected tables.

The example doesn't do anything with the tables chosen—this is left up to you. Figure 10.26 shows what the `MultiSelectListBoxExample2` form looks like when it's first opened in Run mode.

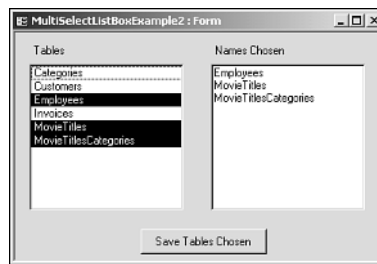


FIGURE 10.26

Selections can be saved to a table and then reselected by opening the form.

The first task in the list is done in the `OnLoad` event of the form, shown in Listing 10.6.

LISTING 10.6 Chap10.mdb: Noting the Previously Chosen Tables

```
Private Sub Form_Load()
    Dim rstTablesChosen As New ADODB.Recordset
    Dim intCurrTable As Integer

    '-- Open the table containing previously selected choices
    rstTablesChosen.Open "TablesChosen", CurrentProject.Connection, adOpenDynamic

    '-- Loop through the selected choices
    Do Until rstTablesChosen.EOF
        '-- For each of the items in the listbox, see if it matches the
        '-- current table name.
```

LISTING 10.6 Continued

```
For intCurrTable = 0 To Me!lboTableToChoose.ListCount - 1
    '-- If there is a match, mark it in the list box as selected
    If rstTablesChosen!TableName = _
        Me!lboTableToChoose.ItemData(intCurrTable) Then
        Me!lboTableToChoose.Selected(intCurrTable) = True
    Exit For
End If
Next intCurrTable
rstTablesChosen.MoveNext
Loop

End Sub
```

Again, there are more ways to perform the preceding task, and some might even be more efficient. But this way is easier to follow. The first step is that the `TablesChosen` table is opened and looped through. Each list box's `ItemData` collection is compared to each record in the recordset. This is done with the following code lines:

```
For intCurrTable = 0 To Me!lboTableToChoose.ListCount - 1
    '-- If there is a match, mark it in the list box as selected
    If rstTablesChosen!TableName = _
        Me!lboTableToChoose.ItemData(intCurrTable) Then
        Me!lboTableToChoose.Selected(intCurrTable) = True
    Exit For
End If
Next intCurrTable
```

Notice the use of the `ListCount` property and the `Selected()` array. A list box's `ListCount` property gives the number of entries in the overall list itself. The `Selected()` array holds an array of Booleans that specify whether the individual rows are selected.

TIP

After the table name is found, the `For` loop is exited by using the `Exit For` command. Although this isn't considered "top-down" programming, it does help performance and, therefore, it's your decision whether to use it.

Again, the `Form_Load` routine loads the list box with the preselected items saved from the last time this routine was used.

PART II

The function to display the selected items in the txtTablesChosen text box on the right is similar to the ShowNamesChosen() function earlier in Listing 10.5. Listing 10.7 shows the ShowTablesChosen() function.

LISTING 10.7 Chap10.mdb: Displaying Selected Table Names

```
Function ShowTablesChosen() As String
    Dim varTable As Variant
    Dim strTemp As String

    '-- for each of the items in the ItemsSelected collection
    For Each varTable In Me!lboTableToChoose.ItemsSelected()
        '-- If not the first item, put a carriage return line feed
        '-- in front of the item
        If Len(strTemp) <> 0 Then
            strTemp = strTemp & vbCrLf
        End If
        '-- Grab the current item
        strTemp = strTemp & Me!lboTableToChoose.ItemData(varTable)
    Next

    '-- Assign the final string pass it back
    ShowTablesChosen = strTemp

End Function
```

The only difference to Listing 10.5, besides using different variable names, is that the ShowTablesChosen() function gets the actual bound column to concatenate to the temporary string, as shown in the following code line:

```
strTemp = strTemp & Me!lboTableToChoose.ItemData(varTable)
```

Another item similar to the example discussed in the section “A Simple Example for Getting Selected Items” is that you need to put the Me.Recalc command in the list box’s AfterUpdate event.

The final piece of the puzzle in this example is the code for saving the current items that were selected during the session. Listing 10.8 shows this code, which is placed on the cmdSaveTablesChosen command button.

LISTING 10.8 Chap10.mdb: Saving Currently Selected Items

```
Private Sub cmdSaveTablesChosen_Click()
    Dim rstTablesChosen As New ADODB.Recordset
    Dim intCurrTable As Integer
    Dim varTable As Variant
```

LISTING 10.8 Continued

```
'-- Delete the previous choices from the table
CurrentDb.Execute "Delete * From TablesChosen"

'-- Open the TablesChosen table
rstTablesChosen.Open "TablesChosen", CurrentProject.Connection, _
    adOpenKeyset, adLockOptimistic, adCmdTable

'-- For each of the items in the ItemsSelected collection
'-- Add a new record in the TablesChosen table.
For Each varTable In Me!lboTableToChoose.ItemsSelected()
    rstTablesChosen.AddNew
    rstTablesChosen!TableName = Me!lboTableToChoose.ItemData(varTable)
    rstTablesChosen.Update
Next varTable

rstTablesChosen.Close

End Sub
```

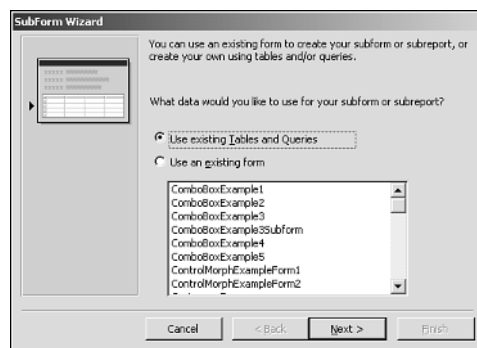
This code first deletes the current entries in the TablesChosen table and then loops through the ItemsSelected collection, adding a new record for each entry.

These methods are only a couple of examples of the power of the multiselect list box. As you work with them, you'll come up with literally hundreds of ways to use them as you build your applications. More examples of using the multiselect list box can be found in Chapter 11, "Creating Powerful Reports."

Getting Relief with the Subform/Subreport Wizard

One irritation in using subform controls is having to create the form used as the subform. Although it's not the most advanced topic, using the Subform/Subreport Wizard can save a great deal of time and hassle. Rather than create the form used as a subform or subreport ahead of time, follow these steps:

1. Create the main form as you normally do.
2. While still in Design view on the main form, make sure that the Wizard control is toggled on in the toolbox.
3. Click the Subform/Subreport control. The first dialog of the SubForm Wizard appears (see Figure 10.27).
4. Follow the directions given by the wizard.

**FIGURE 10.27**

The SubForm/SubReport Wizard can save some time by setting up certain properties for you.

Again, using this wizard saves the trouble of creating your subforms before dealing with the main form. For more information on working with subforms, refer to Chapter 9.

Giving Controls Spreadsheet-Type Cursor Movements

One thing that past and present spreadsheet users find frustrating is the inability to *cursor*, or move, around on a form as you can on a spreadsheet. This is especially true when the form itself is set up in such a way that it resembles a spreadsheet. An example of this might be financial information for a customer going over three years.

Figure 10.28 shows an example of a form set up in columnar format. This form, `SpreadSheetTypeMovementExample`, is found in the file `Chap10.mdb`, on this book's Web page at www.samspublishing.com.

FIGURE 10.28

Although this kind of form setup is easy on the eyes, maneuvering through it can be frustrating for some users.

Looking at the Problem

The form is set up in such a way that when you press the down- or up-arrow key, the cursor goes to the field horizontally either after or before, wrapping at the edges of the screen. The other alternative is to set the tab order vertically, but you still can't go all four directions by using the arrow keys. The following technique takes care of this problem by using two VBA routines and some form settings.

Creating a Solution

In the `SpreadSheetTypeMovementExample` form, an event procedure is added to each `KeyDown` event of the text boxes that contain input. Figure 10.29 shows an example of this for the `Year1` field.

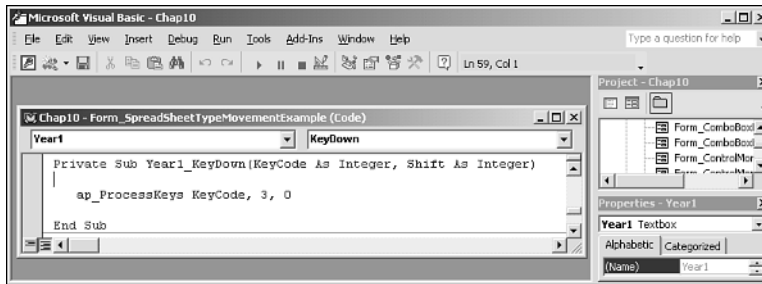


FIGURE 10.29

The KeyCode system parameter contains the ASCII value of the last pressed key.

The call to `ap_ProcessKeys` takes three parameters (see Listing 10.9). The first parameter, `intKeyCode`, is the ASCII value last pressed. The next parameter, `intDownMove`, is the number of fields to move down. Last is `intUpMove`, which is the number of cells to move up.

LISTING 10.9 Chap10.mdb: Controlling Keystrokes

```
Sub ap_ProcessKeys(intKeyCode As Integer, intDownMove As Integer, _
    intUpMove As Integer)

    Dim strFieldToMoveTo As String
    Dim frmCurrent As Form
    On Error GoTo ap_ProcessKeys_Error

    '-- Grab the current form
    Set frmCurrent = Screen.ActiveForm
```

LISTING 10.9 Continued

```
'-- If the current keystroke is the Down Arrow
If intKeyCode = apDownArrow Then
    '-- If a value has been entered for moving down, do it
    If intDownMove Then
        frmCurrent(ap_NextFieldTab(frmCurrent, intDownMove)).SetFocus
        intKeyCode = 0
    Else '-- else stay put.
        intKeyCode = 0
    End If
'-- If the current keystroke is the Up Arrow
ElseIf intKeyCode = apUpArrow Then
    '-- If a value has been entered for moving up, find
    '-- it and set focus to it.
    If intUpMove Then
        frmCurrent(ap_NextFieldTab(frmCurrent, -intUpMove)).SetFocus
        intKeyCode = 0
    Else '-- else stay put.
        intKeyCode = 0
    End If
End If

Exit Sub

ap_ProcessKeys_Error:
    MsgBox Err.Description
    Exit Sub

End Sub
```

You can find the code for the `ap_ProcessKeys` routine in the `ap_SpreadSheetkeystrokes` module, also in the `Chap10.mdb` database on this book's Web page.

Tip

Place these routines in a global module and use the `Screen.Active` objects to make these routines available to as many different forms as you need.

The main task that the `ap_ProcessKeys` routine does is to `SetFocus` on the new field, based on the number of field requests to skip. If a zero is passed, the keystroke is ignored and the cursor

stays on the current field. This process handles the first and last rows on the form. For the first row in the field, the following call would be used because you want to go three fields down but no fields up:

```
ap_ProcessKeys KeyCode, 3, 0
```

TIP

If you added another column to your form, simply change the 3 to a 4. It's a good idea to use a constant value for this call throughout your form. That way, if you need to change the value, you do so only once.

If you were in a middle row, you would make the following call for three up or down:

```
ap_ProcessKeys KeyCode, 3, 3
```

On the bottom row, you would make this call:

```
ap_ProcessKeys KeyCode, 0, 3
```

TIP

Set the form's `Cycle` property to `Current Record` to keep from tabbing off the record. If you're using multiple pages, set it to `Current Page`.

The code for locating the new field to jump to is actually in the `ap_NextFieldTab()` function, which takes the current form and the number of fields to move forward or backward in the tab index (see Listing 10.10).

LISTING 10.10 Chap10.mdb: Handling the Keyboard with VBA

```
Function ap_NextFieldTab(frmCurrent As Form, intMove As Integer) As String
    Dim ctlCurrent As Control
    Dim intCurrTab As Integer

    '-- Errors will be trapped.
    On Error Resume Next

    '-- Examine each of the controls on the current form.
    For Each ctlCurrent In frmCurrent
        '-- This next code provides two purposes, one is if the current
        '-- control doesn't have a TabIndex property, you don't want it.
        '-- The other is that the TabIndex is stored in a variable.
```

LISTING 10.10 Continued

```
intCurrTab = ctlCurrent.TabIndex
If Err = 0 Then
    '-- Look for the control that is intMove's away from the current control
    If intCurrTab = Screen.ActiveControl.TabIndex + intMove Then
        '-- Store the name and exit the function.
        ap_NextFieldTab = ctlCurrent.Name
        Exit Function
    End If
Else
    '-- Reset the error and try the next control
    Err = 0
End If
Next ctlCurrent

'-- If none is found, an error has occurred and just stay put.
ap_NextFieldTab = Screen.ActiveControl.Name

End Function
```

This function takes into consideration the display-only field by working off the `TabIndex` property. This function also takes the current control's `TabIndex` property and adds the number of tabs, up or down, to move to. It then passes back the name of the new control.

Manipulating Controls Through Code

Another task that's sometimes useful is displaying certain command buttons that are based on other choices. To perform this task, you must be able to manipulate controls through code. Using VBA to manipulate controls is easy. Access lets you control the various properties needed, even at runtime.

Figure 10.30 shows a form that's used for a main switchboard. This form uses an option group with toggle buttons to emulate a tabbed menu.

NOTE

This example can be done by using the Access Tab control described earlier. I use the option group here to show manipulating controls on-the-fly and creating dynamic interfaces.



FIGURE 10.30
These command buttons, used as menu choices, are swapped out with code based on the toggle button clicked.

Examining the Pieces of the Option Group Menu Form

Three different Access objects make up this Option Group menu form: a table named `Menus`, a form named `ManipulatingControlsExample`, and the VBA code behind the form.

Each category used on this menu has a number of tools assigned to it. This is done by using the `Tag` property of each command button, which is discussed in a moment. First, look at the `Menus` table, which keeps track of the number of tools for each category. The current contents of this table are as follows:

<i>MenuID</i>	<i>Description</i>	<i>NumberOfTools</i>
1	Order Entry	9
2	Sales	4
3	Purchasing	4

The `MenuID` field is used for locating the corresponding tab value during program execution. The `NumberOfTools` field is used to control an array. A description is there for your own purposes, such as documentation.

Introducing the `ManipulatingControlsExample` Form

Figure 10.31 shows the `ManipulatingControlsExample` form in Design view. Most controls are stored on page two of the form, after the Page Break control, and are moved around when the form is loaded.

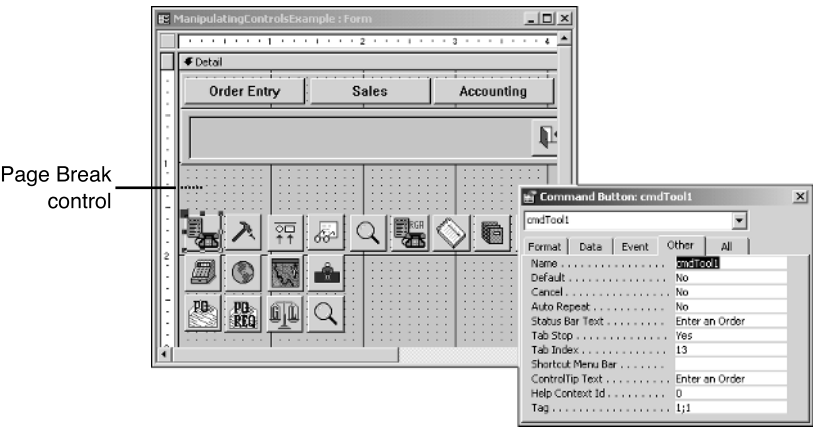


FIGURE 10.31

When you use a combination of property settings and VBA, these command buttons are placed in their proper locations when needed.

Also notice in Figure 10.31 that the first command button's Tag property contains 1;1. The first 1 represents which toggle button this command button corresponds with; the second 1 refers to the order on the menu. The second command button on the first tab choice will have a 1;2 for its Tag property, and so on.

The next items to examine are the various code routines used. All the code is behind the ManipulatingControlsExample form.

Looking at the Code Behind the Form

Before looking at the individual routines, look at the declarations section, which contains various dimension statements:

```
Option Compare Database 'Use database order for string comparisons
Option Explicit
Option Base 1
```

```
Dim rstMenus As New ADODB.Recordset
Dim aryCurrTools() As String
Dim dblTopStorage As Double
```

The Option Base 1 line sets to 1 the base for arrays used for this form. The following table lists the variables and describes their purposes. Note that all these variables are used throughout the form.

<i>Variable</i>	<i>Purpose</i>
<code>rstMenus</code>	Reference to the Menus table
<code>aryCurrTools()</code>	Array that contains the tools used on the currently selected tab
<code>dblTopStorage</code>	Double that contains the top location to store unused controls

The first subroutine to look at is attached to the form's `OnOpen` event. This code establishes the reference to the Menus table with the `RecordSet` type variable, `rstMenus`, which the following code shows:

```
Private Sub Form_Open(Cancel As Integer)
    rstMenus.Open "Menus", CurrentProject.Connection, adOpenKeyset
End Sub
```

The `Form_Load` routine in Listing 10.11 establishes the first view of controls for the menu.

LISTING 10.11 Chap10.mdb: Loading the Initial Menu Settings

```
Private Sub Form_Load()
    Dim i As Integer
    Dim ctlCurrTool As Control

    '-- Establish where to store the prior tab choice by
    '-- picking the first tool top property
    dblTopStorage = Me!cmdTool1.Top

    '-- Establish the maximum number of controls for this tab choice
    ReDim aryCurrTools(rstMenus!NumberOfTools + 1) As String

    For i = 1 To Me.Tag
        Set ctlCurrTool = Me("cmdTool" & LTrim(Str(i)))
        '-- If the tool is used on the first menu pull it in
        '-- to the active tool array
        If GetToolMenu(ctlCurrTool.Tag) = 1 Then
            aryCurrTools(GetToolOrder(ctlCurrTool.Tag)) = ctlCurrTool.Name
        End If
    Next

    '-- Add on the Close button
    aryCurrTools(rstMenus!NumberOfTools + 1) = "cmdClose"

    '-- Moves the Buttons to their positions.
    SetToolForm

End Sub
```


This routine performs the following steps:

1. The Top position is established for storing controls out of the way:
`dblTopStorage = Me!cmdTool1.Top`
2. The `aryCurrTools` array is redimensioned to contain the current list of controls plus the Close button:

```
ReDim aryCurrTools(rstMenus!NumberOfTools + 1) As String
```

3. While the routine loops through the command buttons on the form, if the Tag property for the current control matches 1 (the first choice), the control name is stored in the array. It's stored in the position retrieved from the order portion of the Tag property:

```
For i = 1 To Me.Tag
    Set ctlCurrTool = Me("cmdTool" & LTrim(Str(i)))
    '-- If the tool is used on the first menu pull it in
    '-- to the active tool array
    If GetToolMenu(ctlCurrTool.Tag) = 1 Then
        aryCurrTools(GetToolOrder(ctlCurrTool.Tag)) = ctlCurrTool.Name
    End If
Next
```

Listing 10.12 shows the code for the `GetToolMenu()` function; Listing 10.13 shows the code for the `GetToolOrder()` function.

LISTING 10.12 Chap10.mdb: Returning the Tag Property's Menu-Level Portion

```
Private Function GetToolMenu(strToolTag As String) As Integer
    '-- Strips off the Menu level portion of the tools command
    '-- button tag property
    GetToolMenu = CInt(Left$(strToolTag, InStr(strToolTag, ";") - 1))
End Function
```

LISTING 10.13 Chap10.mdb: Getting the Current Command Button Order from the Tag Property

```
Private Function GetToolOrder(strToolTag As String) As Integer
    '-- Strips off the order portion of the tools command button tag property
    GetToolOrder = CInt(Mid$(strToolTag, InStr(strToolTag, ";") + 1))
End Function
```

4. The `cmdClose` control name is stored in the last position of the `aryCurrTools` array:
`aryCurrTools(rstMenus!NumberOfTools + 1) = "cmdClose"`
5. Listing 10.14 shows the final routine, `SetToolForm`, which runs through the array of controls for the current option group choice. Each control is positioned beside the previous control.

LISTING 10.14 Chap10.mdb: Arranging Command Buttons Programmatically

```

Private Sub SetToolForm()
'-- Moves the tools command buttons around on the form
    Dim dblCurrStart As Double, dblTop As Double
    Dim intCurrTool As Integer

    '-- Establish the initial positions of the Controls
    dblCurrStart = Me!recToolBox.Left + 25
    dblTop = Me!recToolBox.Top + 25

    '-- Loop through the array and set the layout properties for each control
    For intCurrTool = 1 To UBound(aryCurrTools)
        Me(aryCurrTools(intCurrTool)).Top = dblTop
        Me(aryCurrTools(intCurrTool)).Left = dblCurrStart
        dblCurrStart = dblCurrStart + Me(aryCurrTools(intCurrTool)).Width + 10
    Next intCurrTool

End Sub

```

These routines are run when the form is first opened. Some of them are also used when a new toggle button is chosen. The main routine, in Listing 10.15, is attached to the After Update event of the optTabs option group control.

LISTING 10.15 Chap10.mdb: Switching Menu Choices

```

Private Sub optTabs_AfterUpdate()
    Dim iNumControls As Integer, intIndex As Integer
    Dim ctlCurrTool As Control

    '-- Moves the prior level tools off the first page.
    ClearCurrTools

    '-- Get the total controls, and moves to the current menu level
    iNumControls = CInt(Me.Tag)
    rstMenus.Find "[MenuID] = " & Me!optTabs, , , 1

    '-- Stores the initial number of controls for the new menu level
    ReDim aryCurrTools(rstMenus!NumberOfTools + 1) As String

    For intIndex = 1 To iNumControls
        Set ctlCurrTool = Me("cmdTool" & LTrim(Str(intIndex)))
        '-- If for the current option group choice
        If GetToolMenu(ctlCurrTool.Tag) = Me!optTabs Then
            aryCurrTools(GetToolOrder(ctlCurrTool.Tag)) = ctlCurrTool.Name
        End If
    Next

```

LISTING 10.15 Continued

```
'-- Add on the Close button
aryCurrTools(rstMenus!NumberOfTools + 1) = "cmdClose"

'-- Moves the Buttons to their positions.
SetToolForm

End Sub
```

Because this routine is similar to the `Form_Load` routine in Listing 10.11, you can follow what's happening. One difference between this routine and the `Form_Load` routine is that this routine calls the `ClearCurrTools` routine to move the previous command buttons off the page (see Listing 10.16).

LISTING 10.16 Chap10.mdb: Clearing the Current Settings

```
Private Sub ClearCurrTools()
'-- This routine clears the previous tab choice by
'-- storing them on the second page
Dim intIndex As Integer
For intIndex = 1 To UBound(aryCurrTools)
    Me(aryCurrTools(intIndex)).Top = dblTopStorage
Next
End Sub
```

The other difference is that the `optTabs` value is used rather than the hard-coded value of 1 in the `Form_Load` routine.

If you open the `ManipulatingControlsExample` form and click the three toggle buttons, you'll see how the controls change with the option group. You can create many other form types by using methods similar to this one.

Summary

Access controls let you create powerful forms that can take care of almost any need. From combo boxes and list boxes to manipulating controls with Visual Basic for Applications, Access has the tools to get the job done. For more information on using VBA and collections, see the following chapters:

- Chapter 2, “Coding in Access 2002 with VBA,” adds insight to using the various commands found in VBA.

- Working with collections, although at first confusing, can open the whole Access world after you understand it. Chapter 4, “Working with Access Collections and Objects,” shows you the collections, along with their properties and methods.
- In Chapter 8, “Using Queries to Get the Most Out of Your Data,” you learn how to optimize your queries and make them really work for you, as well as when and where to use SQL statements rather than prebuilt queries.
- Chapter 9, “Creating Powerful Forms,” gives you additional techniques for swapping out subforms and using code behind forms for generic purposes.
- To learn additional techniques for using multiselect list boxes, see Chapter 11, “Creating Powerful Reports.”

Creating Powerful Reports

CHAPTER

11

IN THIS CHAPTER

- **Creating Summary, Detail, and Summary/Detail Reports from the Same Report** 314
- **Creating Dynamic Groupings for the Same Report with QBF** 320
- **The Elusive Feature: Creating Snaking Reports** 323
- **Printing Multiple Topics Through a MultiSelect List Box** 328
- **Creating a Wizard-Like Interface for Selecting Group-By Items** 335
- **Formatting Reports Dynamically** 347

For the most part, day-to-day reports are easy to use and don't need any code at all. Even creating reports by using subforms isn't that tough and can be figured out pretty quickly. Some things, however, might not be quite as straightforward and need a bit of code:

- Using the same report for your summary, detail, and summary/detail reports
- Creating dynamic groupings for the same report by using a query by form (QBF)
- Creating reports with snaking columns
- Choosing multiple topics to print by using a MultiSelect list box
- Creating a generic wizard-like interface for single-level grouped reports

As it does with forms, Access lets you do whatever you need to do to get the job done when working with reports. Creating versatile reports isn't a problem when you use VBA behind the report. A good example of this is creating a report that you can use to present data in detail, summary, or both.

NOTE

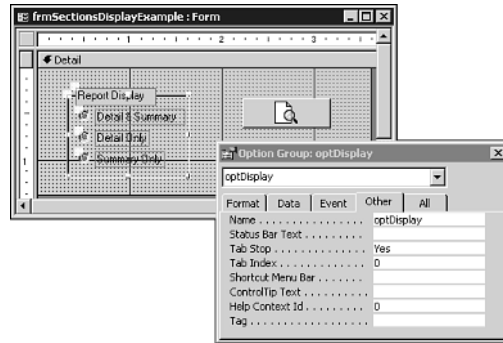
You can find all the examples given in this chapter in the Chap11.mdb database, located on this book's Web page at www.samspublishing.com.

Creating Summary, Detail, and Summary/Detail Reports from the Same Report

Usually, when clients ask that data be displayed in summary or detail format, developers create two versions of the same report. For this example, three choices are requested, so three reports are needed: summary, detail, and summary/detail. This example shows you how to do it with just one report.

This example uses the `frmSectionsDisplayExample` form, which calls the report of the same name. The form consists of the `optDisplay` option group and two command buttons: `cmdPreview`, which performs the `OpenReport` method, and `cmdClose`, which closes the form. Figure 11.1 shows the form in Design view.

The Command Button Wizard created the command buttons used on this form. This wizard provides not only the graphics for the buttons, but also the VBA code behind the forms. The Command Button Wizard generated the code in Listing 11.1 for the `cmdPreview` button.

**FIGURE 11.1**

This form calls a report with three different layouts.

LISTING 11.1 Chap11.mdb: Opening the rptSectionsDisplayExample Report in Print Preview

```
Sub cmdPreview_Click()
On Error GoTo Err_cmdPreview_Click

    DoCmd.OpenReport "rptSectionsDisplayExample", acPreview

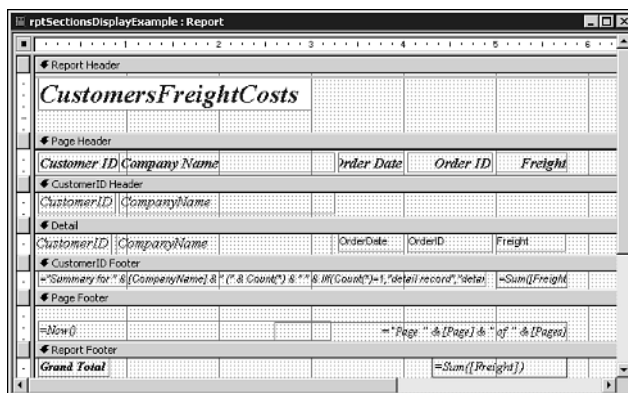
Exit_cmdPreview_Click:
    Exit Sub

Err_cmdPreview_Click:
    MsgBox Err.Description
    Resume Exit_cmdPreview_Click

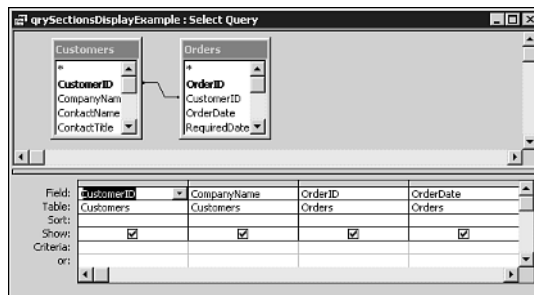
End Sub
```

The real power in this example is the code behind the rptSectionsDisplayExample report. Notice in Figure 11.2 that CustomerID and CompanyName show up twice in this report—once in the CustomerID Header section and once in the Detail section.

Before you look at the code behind the report, the report needs the qrySectionsDisplayExample query (see Figure 11.3).

**FIGURE 11.2**

Only one copy of CustomerID and CompanyName actually appears in the report when it's run.

**FIGURE 11.3**

The rptSectionsDisplayExample report uses the straightforward qrySectionsDisplayExample query.

The code that does the work (see Listing 11.2) is attached to the report's OnOpen event.

LISTING 11.2 Chap11.mdb: Setting Up How the Sections Will Look

```
Private Sub Report_Open(Cancel As Integer)
    On Error GoTo Report_Open_Error

    Select Case Forms!frmSectionsDisplayExample!optDisplay
        Case 1 '-- Display Detail and Summary
            Me.Caption = "Freight Charges - Detail & Summary"
            Me!CustomerID.Visible = False
            Me!CompanyName.Visible = False
```

LISTING 11.2 Continued

```

Case 2 '-- Display Detail Only
    Me.Caption = "Freight Charges - Detail Only"
    '-- Turn Group Header Section off
    Me.Section(5).Visible = False
    '-- Turn Group Footer Section off
    Me.Section(6).Visible = False
Case 3 '-- Display Summary Only
    Me.Caption = "Freight Charges - Summary Only"
    '-- Turn off Column Headings
    Me!CustomerIDLabel.Visible = False
    Me!CompanyNameLabel.Visible = False
    Me!OrderDateLabel.Visible = False
    Me!OrderIDLabel.Visible = False
    '-- Turn Detail Section off
    Me.Section(0).Visible = False
    '-- Turn Group Header Section off
    Me.Section(5).Visible = False
End Select

Exit Sub

Report_Open_Error:
    MsgBox Err.Description
    Exit Sub

End Sub

```

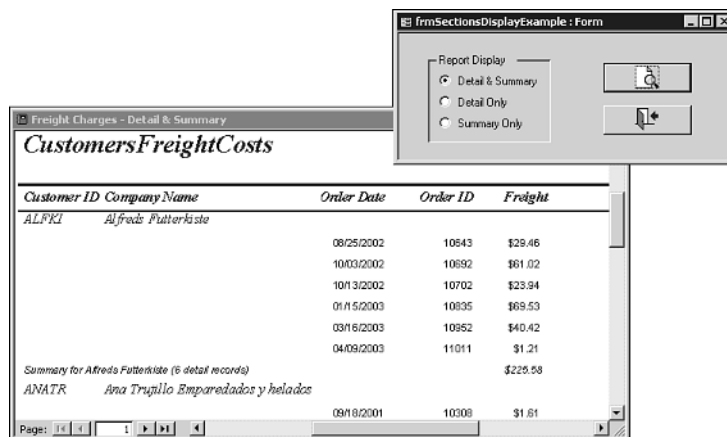
This code first examines the `optDisplay` option group on the `frmSectionsDisplayExample` form and reacts appropriately based on the setting. The first setting is for Detail and Summary:

```

Case 1 '-- Display Detail and Summary
    Me.Caption = "Freight Charges - Detail & Summary"
    Me!CustomerID.Visible = False
    Me!CompanyName.Visible = False

```

This part of the code sets the report's `Caption` property to reflect the display choice. Then it turns off the `CustomerID` and `CompanyName` fields in the Detail section because the `CustomerID` and `CompanyName` are displayed in the `CustomerID` Header section. Figure 11.4 shows the `rptSectionsDisplayExample` report in Print Preview with the Detail & Summary option chosen.

**FIGURE 11.4**

With an option group and an *OnOpen* event, you can have almost unlimited views of a particular report.

The next part of the code sets up the report to display only the Detail section. The first task sets up the caption correctly and then turns off the Group Header and Footer sections.

NOTE

Reports and forms have a *Section* property that's actually an array of all sections for that object. The following table lists the standard elements of the *Section* array. If more than two groupings are on a report, they continue by pairs from 9 on.

Section Index	Description
Section(0)	Form detail section or report detail section
Section(1)	Form or report header section
Section(2)	Form or report footer section
Section(3)	Form or report page header section
Section(4)	Form or report page footer section
Section(5)	Group-level 1 header section (reports only)
Section(6)	Group-level 1 footer section (reports only)
Section(7)	Group-level 2 header section (reports only)
Section(8)	Group-level 2 footer section (reports only)

Each section has its own set of properties, such as the *Visible* property used in this example.

The following code section displays the Detail section only; Figure 11.5 shows the resulting report.

```
Case 2 '-- Display Detail Only
Me.Caption = "Freight Charges - Detail Only"
'-- Turn Group Header Section off
Me.Section(5).Visible = False
'-- Turn Group Footer Section off
Me.Section(6).Visible = False
```

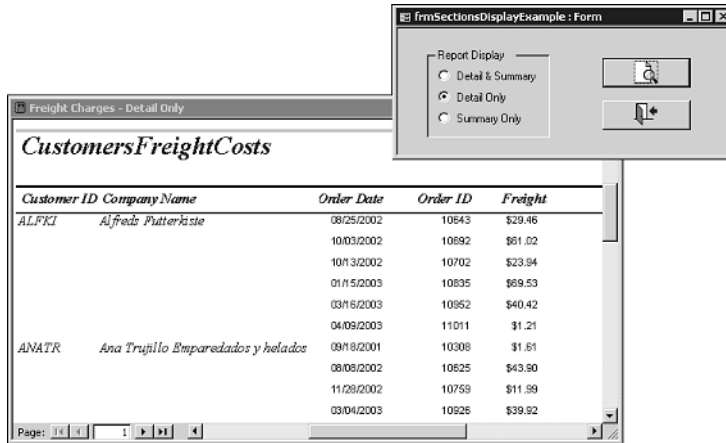
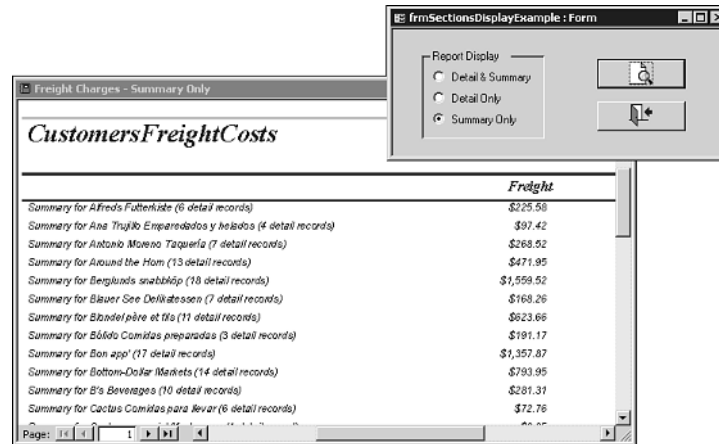


FIGURE 11.5

This version of the report displays just the detail information.

The last choice given is Summary Only. The following code section sets all column headings (except Freight) to not visible, and turns off the Detail and Group Header sections. Figure 11.6 shows the rptSectionsDisplayExample report in Print Preview, with the Summary Only option chosen.

```
Case 3 '-- Display Summary Only
Me.Caption = "Freight Charges - Summary Only"
'-- Turn off Column Headings
Me!CustomerIDLabel.Visible = False
Me!CompanyNameLabel.Visible = False
Me!OrderDateLabel.Visible = False
Me!OrderIDLabel.Visible = False
'-- Turn Detail Section off
Me.Section(0).Visible = False
'-- Turn Group Header Section off
Me.Section(5).Visible = False
```

**FIGURE 11.6**

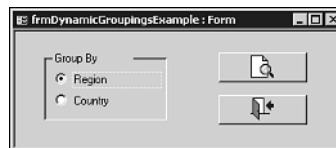
This version of the report displays just the summary information, with most column headings and the Detail and Group Header sections turned off.

You can toggle the visibility of sections even further by having report sections created by using subreports on the main report, toggled as visible or invisible at runtime based on the user's choice.

Creating Dynamic Groupings for the Same Report with QBF

Another nice technique is the ability to switch fields on which report groupings are performed. You can actually make dynamic groupings without using code on the report.

The report used for this example is rptDynamicGroupingsExample. Before you look at the report itself, however, notice that the form used with it is similar to the last example. This form, frmDynamicGroupingsExample, contains an option group (optGroupBy) and two command buttons (cmdPreview and cmdClose). Figure 11.7 shows the form in Run view.

**FIGURE 11.7**

By accessing the optGroupBy option group, you can group the report in different ways.

As with the preceding example, the Command Button Wizard created both command buttons. Listing 11.3 shows the code for the cmdPreview button, which opens the rptDynamicGroupingsExample report in Print Preview mode.

LISTING 11.3 Chap11.mdb: Opening the rptDynamicGroupingsExample Report in Print Preview

```
Sub cmdPreview_Click()
    On Error GoTo Err_cmdPreview_Click
    Dim stDocName As String

    stDocName = "rptDynamicGroupingsExample"
    DoCmd.OpenReport stDocName, acPreview

Exit_cmdPreview_Click:
    Exit Sub

Err_cmdPreview_Click:
    MsgBox Err.Description
    Resume Exit_cmdPreview_Click

End Sub
```

The report itself does most of the work. The IIf() (immediate if) function looks at the optGroupBy value in three places on the frmDynamicGroupingsExample form. The first place is in the Sorting and Grouping property sheet (see Figure 11.8).

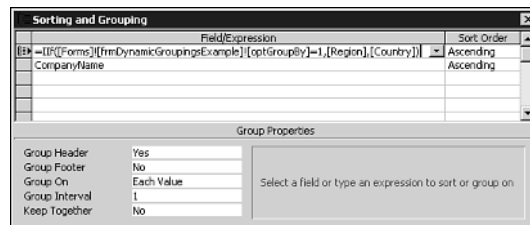


FIGURE 11.8

Most people don't realize that you can use an IIf() function to change groupings dynamically.

The following command does the comparison:

```
=IIf([Forms]![frmDynamicGroupingsExample]![optGroupBy]=1,[Region],[Country])
```

If optGroupBy equals 1, the grouping is set to the Region field; otherwise, it's set to the Country field.

The next place that uses `IIf()` to examine `optGroupBy` is the `txtRegionOrCountry` text box in the report's Group Header section. Here's the command used:

```
=IIf([Forms]![frmDynamicGroupingsExample]![optGroupBy]=1,[Region],[Country])
```

As you can see, this command is identical to the one used in the Sorting and Grouping dialog.

The last control to use `IIf()` is the `txtRegionOrCountryLabel1` text box, to display a heading for the first column. The statement itself is

```
=IIf([Forms]![frmDynamicGroupingsExample]![optGroupBy]=1,"Region","Country")
```

Although this looks similar to the first two uses, rather than display the Region and Country fields, the statement shows the literals `Region` and `Country`. Figures 11.9 and 11.10 show the `rptDynamicGroupingsExample` report in Print Preview mode—first with the region chosen and then with the country.

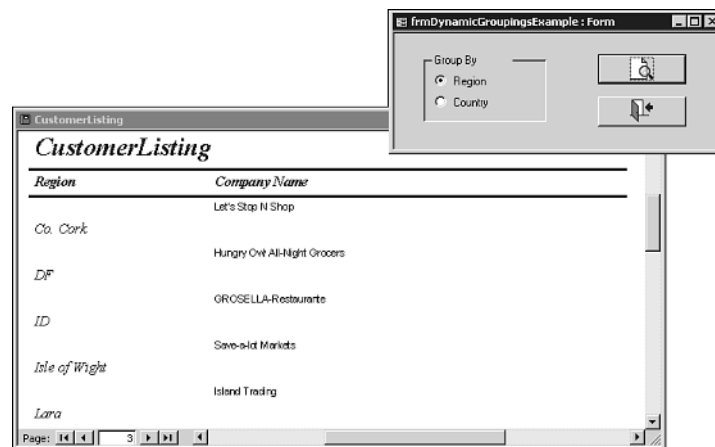
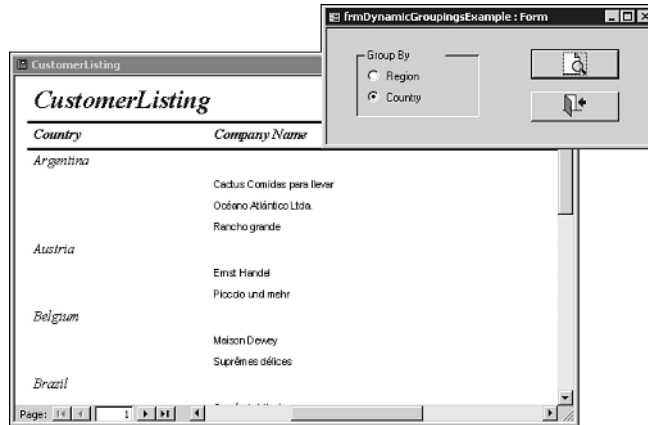


FIGURE 11.9

By reading this option group on the form, the report can control how it groups the information.

**FIGURE 11.10**

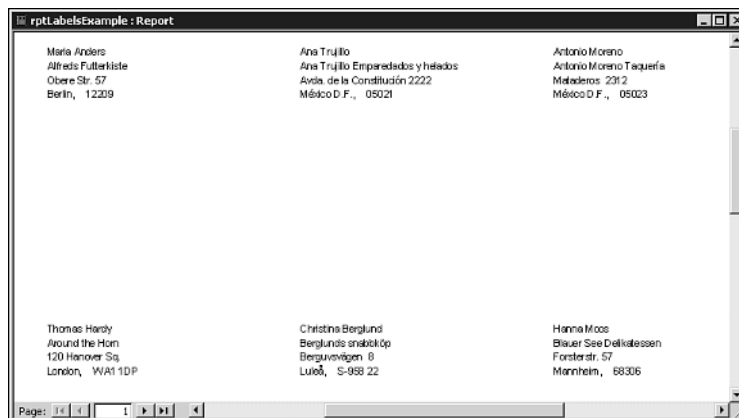
By choosing the *Country* option on the form, `IIIf()` helps control how to group the report's information.

The Elusive Feature: Creating Snaking Reports

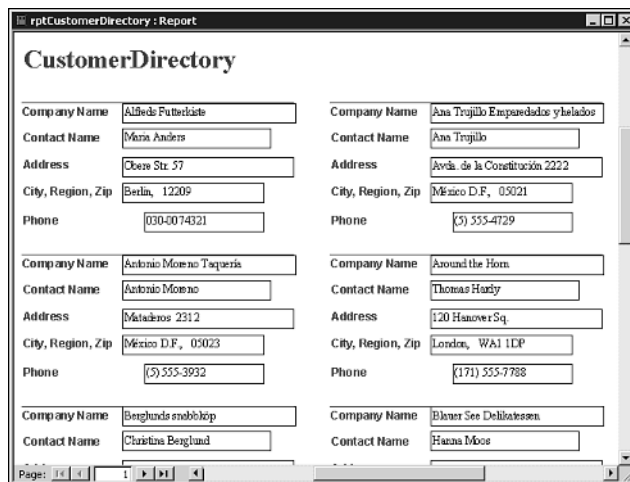
Another trick—actually just a feature developers don't think to use—is the snaking report feature. In *snaking reports*, data that normally runs down one column instead shows up in two or more columns, depending on how you want them. This feature is actually quite easy to use and takes no VBA coding whatsoever. Play around with a few of your single-column reports and dazzle your boss by adding a couple more columns of information.

Some examples of uses for snaking reports are as follows:

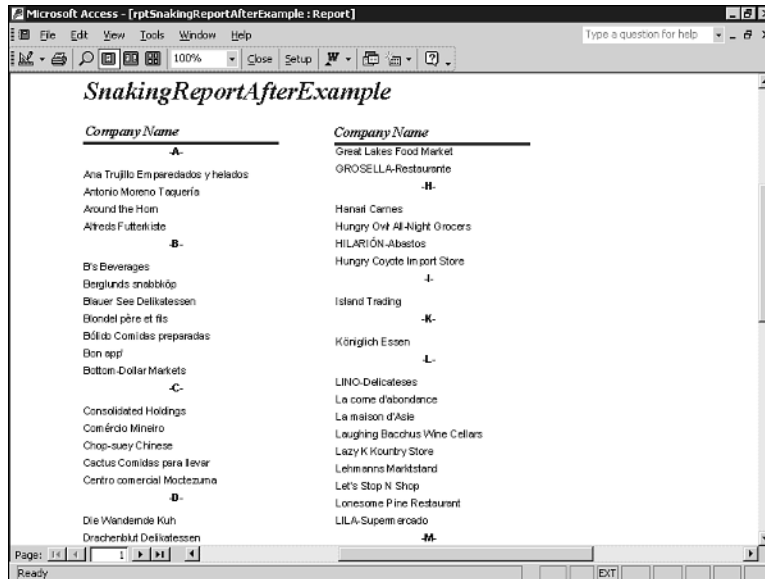
- **Mailing labels.** The Label Wizard uses the snaking feature in creating the different types of Avery mailing label sizes. Figure 11.11 shows an example of labels created with the Label Wizard.
- **Address book.** When you want more than one person's name and address across the page, the snaking feature is your tool. The `rptCustomerDirectory` report in Figure 11.12 is located in `Chap11.mdb`.
- **Multiple-column customer list.** Sometimes you want to show two columns of names to save paper. Figure 11.13 shows the `rptSnakingReportAfterExample` report (again, found in `Chap11.mdb`).

**FIGURE 11.11**

Use the layout features available through the File menu's Page Setup comment to snake the addresses for these mailing labels.

**FIGURE 11.12**

By using the snaking report feature, you can create this address book-style layout.

**FIGURE 11.13**

The snaking feature helps you save paper by fitting more than one column on a page.

TIP

Use the View toolbar button (the first button on the left) to go to Design view, even if you didn't go to Print Preview from Design view to begin with. This saves steps in closing and then reopening reports to switch between views.

Looking at the Before Report

This example uses a multicolumn customer list to show how to use the snaking feature. Open the rptSnakingReportBeforeExample report in the Chap11.mdb database on this book's Web page at www.sampublishing.com. This report was initially created with the Report Wizard and just showed the CompanyName for the Detail section.

The Group Header is created off the first character of the CompanyName field. The expression for this is

```
= "-" & Left$([CompanyName],1) & "-"
```

The text box is named—aptly enough—txtCompanyFirstLetter. In the Sorting and Grouping dialog, the CompanyName is the field used, with the Group On property set to Prefix Characters and Group Interval set to 1 (see Figure 11.14).

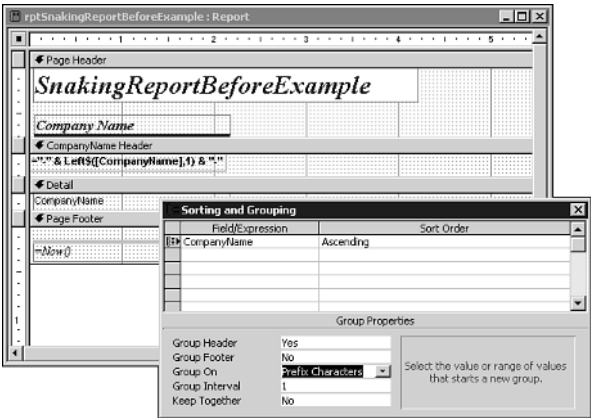


FIGURE 11.14

By using the Group Interval property, you can group on single characters.

Running this report now would create a single column. The magic that needs to take place is in the Page Setup dialog (see Figure 11.15). To get to this dialog, from the File menu choose Page Setup, and then click the Columns tab. Table 11.1 lists the different settings available and their effects on the snaking feature.



FIGURE 11.15

Use the Columns page of the Page Setup dialog to set up the snaking feature on reports.

TABLE 11.1 Columns Page Settings

Setting	Description
<i>Grid Settings</i>	
Number of Columns	Establishes how many columns to display on the page; 1 is the default. This setting affects many other settings in the dialog.
Row Spacing	Affects the space between rows, with 0 as the default.
Column Spacing	Controls how much space is placed between columns when Number of Columns is set to something other than 1. The default is .25 inch.
<i>Column Size</i>	
Width	Sets how wide each column in the detail will be. By default, it's set to the report's width and needs to be changed when more than one column is used.
Height	Affects the height of the row. It's set to whatever the detail height is.
Same as Detail	When selected, resets the width of the column to whatever the report's size is.
<i>Column Layout</i>	
Down, Then Across	Like the next setting, determines which way to snake the data in the report's columns. This setting takes the data all the way down the page, jumps to the top of the same page, and then repeats for the number of columns specified for Number of Columns.
Across, Then Down	Takes the data and goes across however many Number of Columns there are before going down the page.

CAUTION

Nine times out of 10, the Width setting screws you up when you switch from using one column to more than one. If you don't reduce this setting, you won't see your other columns, and you won't get an error. Start by reducing this property to 2.5 inches, as shown in the example in the following section.

Working with the After Report

Figure 11.15 shows the Page Setup dialog for the rptSnakingReportAfterExample report, which is the modified report that uses the snaking feature. Follow these steps to set up the Page Setup dialog for this report:

1. Put a 2 in the Number of Columns text box.
2. Change the item's Width to 2.5 inches. (This deselects the Same as Detail check box.)
3. Choose whether you want to have the data go Down, Then Across, or Across, Then Down.
4. Click OK and then open the report in Print Preview.

TIP

You might want to put a column header over the other columns you make. Access does *not* create these automatically.

Printing Multiple Topics Through a MultiSelect List Box

Using the ListBox control in MultiSelect view is extremely useful for generating reports and giving users more control over what information is printed.

In this example, users can select one, some, or all product categories for which to print a report (see Figure 11.16). It uses the list box quite extensively, including adding and removing categories to a temporary table for the report. This is a nice way of using the list box and an additional text box because users see exactly what has to be selected.

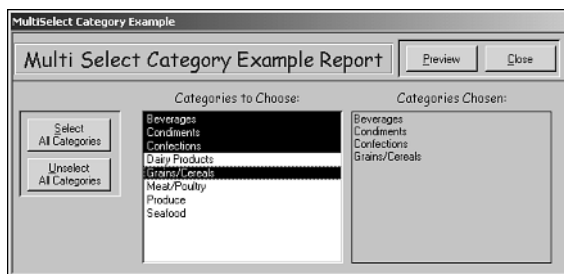


FIGURE 11.16

Using the MultiSelect version of the list box can make a powerful report.

Looking at the rptMultiSelectCategoryExample Report

The rptMultiSelectCategoryExample report, found in Chap11.mdb on this book's Web page at www.sampublishing.com, is a single grouped report that lists products by categories.

Figure 11.17 shows the report in Design view.

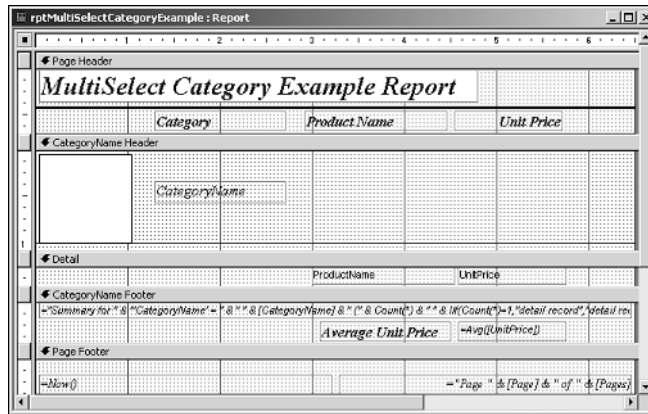


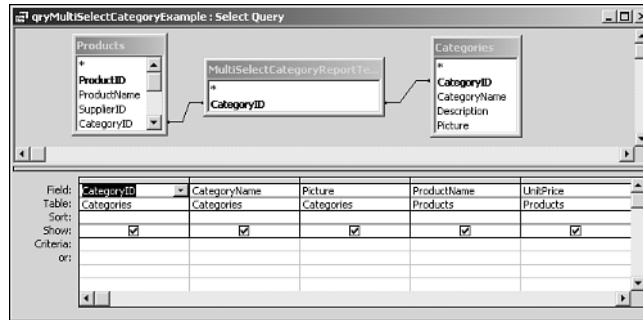
FIGURE 11.17

This report displays products for each category.

TIP

Graphics, such as the category picture used in the report shown in Figure 11.17 (in Design view, the blank square next to CategoryName), can greatly enhance reports. The graphics can be either a bound OLE object, as in this case, or an unbound object, such as a company logo. To add an unbound picture in a report's Design view, choose Picture from the Insert menu; then select the graphic to add.

For its record source, the report uses the qryMultiSelectCategoryExample query, which combines three tables: the Products and Categories tables, and a temporary table named MultiSelectCategoryReportTemp. The frmMultiSelectCategoryExample form (discussed in the next section) controls the temporary table, which consists of one field, CategoryID. Figure 11.18 shows the three tables with the rest of the query.

**FIGURE 11.18**

The temporary table (in the middle) drives this report.

Looking at the MultiSelect List Box Form

The frmMultiSelectCategoryExample form does the lion's share of the work and relies on the following controls:

- The lboGetCategory MultiSelect list box allows users to choose individual or multiple categories of the products they want to print.
- Two command buttons, cmdSelectAll and cmdUnSelectAll, provide the capability to select all the categories to print or none. Users appreciate the ability to select all, so that they don't have to click all the choices. They also appreciate being able to deselect all and choose them individually.
- The txtChosenCats text box displays the categories chosen.

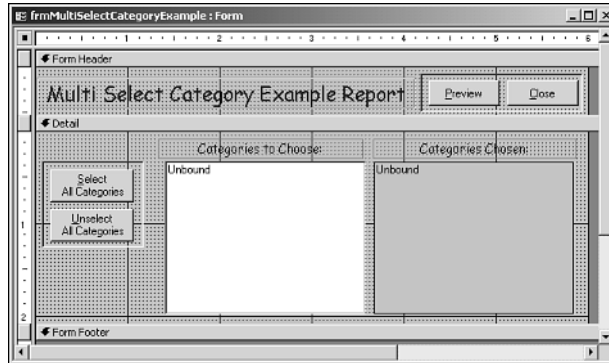
Two other command buttons are cmdPreview, for previewing the report, and cmdClose, for closing the form. Figure 11.19 shows the frmMultiSelectCategoryExample form in Design view.

Code Listings for the MultiSelect List Box Form

All the code used here is stored behind the form. To start, let's look at the declaration section behind the frmMultiSelectCategoryExample form:

```
Option Compare Database
Option Explicit
```

```
Private cmdCatExampleRemove As Command
Private cmdCatExampleAdd As Command
```

**FIGURE 11.19**

Most of these controls have VBA code behind them.

The two Command objects declared here are used for adding and removing categories from the temp table. The first routine to look at is in Listing 11.4.

LISTING 11.4 Chap11.mdb: Setting Up the Form

```
Private Sub Form_Open(Cancel As Integer)
    Dim catCurr As New ADOX.Catalog
    catCurr.ActiveConnection = CurrentProject.Connection

    '-- Clear the report's temporary table
    CurrentDb.Execute "Delete * from MultiSelectCategoryReportTemp;"

    '-- Set references up for adding and removing categories to the temp table
    Set cmdCatExampleAdd = catCurr.Procedures("qryCatExampleTempAdd").Command
    Set cmdCatExampleRemove = _
        catCurr.Procedures("qryCatExampleTempRemove").Command

End Sub
```

The Form_Open routine clears the temp table and sets up references to queries, using ADO Command objects, for updating the temp table. These queries are used in the cmdSelectAll_Click and cmdSelectCurrent routines.

The next routine is used with the lboSelectCurrent list box. Shown in Listing 11.5, the lboGetCategory_AfterUpdate routine selects a single category.

LISTING 11.5 Chap11.mdb: Using the Selected Collection off the lboGetCategory List Box

```

Private Sub lboGetCategory_AfterUpdate()
    Dim intCurrCat As Integer, strChosenCatsTemp As String

    '-- If the category is selected add it to the temp table
    If Me!lboGetCategory.Selected(Me!lboGetCategory.ListIndex) Then
        cmdCatExampleAdd.Parameters("CurrCategoryID") = _
            Me!lboGetCategory.ItemData(Me!lboGetCategory.ListIndex)
        cmdCatExampleAdd.Execute
    Else '-- Otherwise remove the category from the temp table
        cmdCatExampleRemove.Parameters("CurrCategoryID") = _
            Me!lboGetCategory.ItemData(Me!lboGetCategory.ListIndex)
        cmdCatExampleRemove.Execute
    End If

    '-- Re-create the string containing the chosen categories
    strChosenCatsTemp = ""

    For intCurrCat = 0 To Me!lboGetCategory.ListCount - 1
        If Me!lboGetCategory.Selected(intCurrCat) Then
            strChosenCatsTemp = strChosenCatsTemp & _
                Me!lboGetCategory.Column(1, intCurrCat) & vbCrLf
        End If
    Next intCurrCat

    '-- Update the txtChosenCats text box with the string
    Me!txtChosenCats = strChosenCatsTemp

End Sub

```

Although the lboGetCategory_AfterUpdate routine is documented pretty well with comments, I want to cover its highlights:

1. If the category is selected, the routine adds it to the temp table; otherwise, the routine removes it. Whether the category is added or removed is determined by examining the Selected collection and the ListIndex property, which track which item in the list box is currently selected. The following code performs the examination:

```

If Me!lboGetCategory.Selected(Me!lboGetCategory.ListIndex) Then
    cmdCatExampleAdd.Parameters("CurrCategoryID") = _
        Me!lboGetCategory.ItemData(Me!lboGetCategory.ListIndex)
    cmdCatExampleAdd.Execute
Else '-- Otherwise remove the category from the temp table
    cmdCatExampleRemove.Parameters("CurrCategoryID") = _
        Me!lboGetCategory.ItemData(Me!lboGetCategory.ListIndex)
    cmdCatExampleRemove.Execute
End If

```

- The routine re-creates the string containing the chosen categories, and then assigns that string to the txtChosenCats text box:

```
strChosenCatsTemp = ""

For intCurrCat = 0 To Me!lboGetCategory.ListCount - 1
    If Me!lboGetCategory.Selected(intCurrCat) Then
        strChosenCatsTemp = strChosenCatsTemp & _
            Me!lboGetCategory.Column(1, intCurrCat) & vbCrLf
    End If
Next intCurrCat

'-- Update the txtChosenCats text box with the string
Me!txtChosenCats = strChosenCatsTemp
```

The cmdSelectAll_Click routine, the event procedure for the cmdSelectAll command button's OnClick event, does a task similar to the last routine, only for all the unselected categories (see Listing 11.6).

LISTING 11.6 Chap11.mdb: Selecting All the Categories and Adding Them to the Temp Table

```
Private Sub cmdSelectAll_Click()
    Dim intCurrCat As Integer, strChosenCatsTemp As String

    strChosenCatsTemp = ""

    '-- for each of the items in the ItemsSelected collection
    For intCurrCat = 0 To Me!lboGetCategory.ListCount - 1
        '-- Create the temp string for choosen categories
        strChosenCatsTemp = strChosenCatsTemp & _
            Me!lboGetCategory.Column(1, intCurrCat) & vbCrLf
        '-- Add category to temp table if required
        If Me!lboGetCategory.Selected(intCurrCat) = False Then
            cmdCatExampleAdd.Parameters("CurrCategoryID") = _
                Me!lboGetCategory.ItemData(intCurrCat)
            cmdCatExampleAdd.Execute
        End If
        Me![lboGetCategory].Selected(intCurrCat) = True
    Next intCurrCat

    '-- Update the txtChosenCats text box with the string
    Me!txtChosenCats = strChosenCatsTemp

End Sub
```

PART II

The last routine, `cmdUnselectAll_Click`, deselects all the choices (see Listing 11.7). Before deselecting the choices in the list box, the routine executes a delete query, clearing the temp table. (Two more routines, `cmdPreview_Click` and `cmdClose_Click`, do exactly what their names indicate and aren't worth discussing.)

LISTING 11.7 Chap11.mdb: Deselecting the Categories and Clearing the Temp Table

```
Private Sub cmdUnselectAll_Click()
    Dim intCurrCat As Integer, strChosenCatsTemp As String

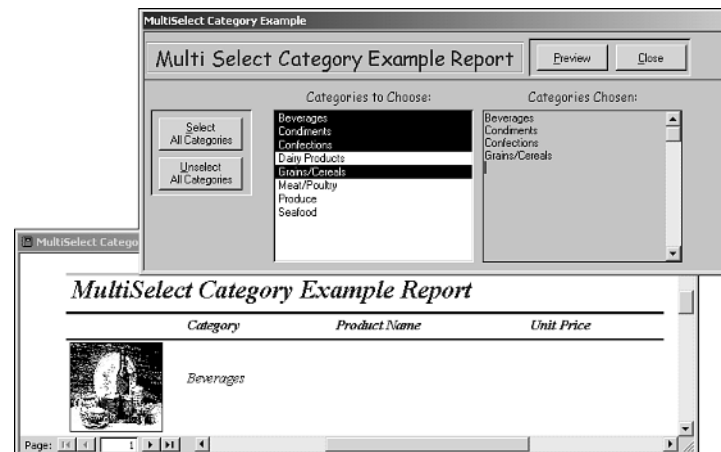
    '-- Clear the report's temporary table
    CurrentDb.Execute "Delete * from MultiSelectCategoryReportTemp;"

    '-- for each of the items in the ItemsSelected collection
    For intCurrCat = 0 To Me![lboGetCategory].ListCount - 1
        Me![lboGetCategory].Selected(intCurrCat) = False
    Next intCurrCat

    '-- Clear the txtChosenCats text box
    Me!txtChosenCats = ""

End Sub
```

Putting all this code to work—along with the tables, queries, and report—makes for a good way of giving users more control over which items are used for a report (see Figure 11.20).


FIGURE 11.20

Here are the report and form, after the choices are made.

The following section deals with a similar report, in that it allows you to choose the items or elements for a grouping that you want to print or preview.

Creating a Wizard-Like Interface for Selecting Group-By Items

A wizard-like interface, in addition to being much more powerful than the one presented in the previous section, makes the current example easier to adapt. Figure 11.21 shows the wizard-type interface for reports. (This interface is considered wizard-like because, being data-driven, it doesn't take much to adapt the same interface for new reports.)

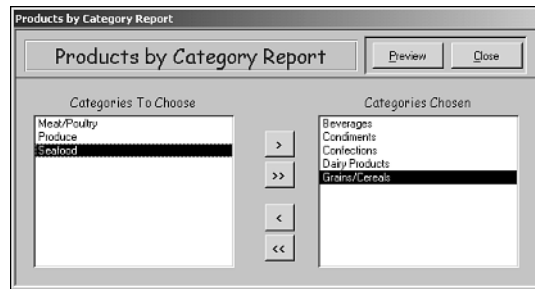


FIGURE 11.21

This interface can be used for an unlimited number of single-level grouping reports.

Here are the main features of this example:

- The interface used for calling the report uses a wizard-like look and feel, giving users something they're used to working with.
- The routines are written so that for you to use the main form created with a new report, you only have to make an entry in a table and have the record source for the report use the table specified here.
- Not only can the individual group elements in this category be selected, but also the order in which the groups are printed can be chosen.

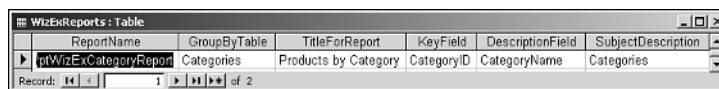
You'll be looking at a lot of code shortly, but the great news is that the example is pretty much data-driven. As a result, all you have to do is make an entry in a table and use another table in the query that the report is based on. This is covered later in the section "Using the Group Element Wizard with a New Report." For now, let's go through the code and objects for the current example.

To work this example, I want to start with the two tables that are the core for making it work. I'll move on to the form that's used, and then work my way to the other objects as they're mentioned.

The Core Tables: WizExReports and WizExElements

The two tables used for this example are for two very different purposes. The first table, `WizExReports`, is maintained for adding new reports. Figure 11.22 shows the entries for the first report. Here's a breakdown of the fields used:

- `ReportName` stores the name of the report—in this case, the `rptWizExCategoryReport` report.
- `GroupByTable` specifies the table that supplies the key values and/or description of the Grouping field. The current example uses the `Categories` table.



ReportName	GroupByTable	TitleForReport	KeyField	DescriptionField	SubjectDescription
rptWizExCategoryReport	Categories	Products by Category	CategoryID	CategoryName	Categories

Record: 1 of 2

FIGURE 11.22

Here are the entries for the report `rptWizExCategoryReport`.

- `TitleForReport` is just what it says: the caption and title of the `frmWizExReports` form.
- `KeyField` is the `KeyField` for the record source mentioned in `GroupByTable`. `CategoryID` is used here.
- `DescriptionField` describes the elements. `CategoryName` is used here.

NOTE

In some cases, when `DescriptionField` and `KeyField` are the same, only `DescriptionField` is filled in. For example, if you were looking at the video store application by using the `Ratings` table, `KeyField` and `DescriptionField` might be the same in that they're both G, PG, or R. Then `DescriptionField` would say `Rating` and `KeyField` would be `Null`. You see this addressed in the `Form_Open` routine of the `frmWizExReports` form shortly in Listing 11.8.

- `SubjectDescription` describes what the elements are—in this case, categories. This field is used in the titles for the list boxes on the `frmWizExReports` form.

I'll go over adding another report to this table later in the chapter.

The second table, `WizExElements`, drives the form and reports used for the wizard-like interface by controlling the grouping elements—in this case, categories. It also specifies the order, and whether the grouping elements are to be included.

TIP

Rather than add elements to and delete elements from a table, this technique toggles a flag field to indicate whether to include an element. This can result in better performance, although it generally takes a bit more work.

Figure 11.23 shows the sample data supplied by using the `rptWizExCategoryReport` example in the `WizExElements` table. Again, let's walk through each field used in this table:

- `FieldID` tracks the original order in which the elements were loaded into the table. The `lboGetFields` list box (on the `frmWizExReports` form) sorts on this `AutoNumber` field and is discussed in more detail in the next section.
- `ElementKey` stores the information from the field supplied in the `WizExReports` table's `KeyField` field.

FieldID	ElementKey	ElementDescription	Include	OrderNo
760	1	Beverages	-1	0
761	2	Condiments	-1	1
762	3	Confections	-1	2
763	4	Dairy Products	-1	3
764	5	Grains/Cereals	-1	4
765	6	Meat/Poultry	0	0
766	7	Produce	0	0
767	8	Seafood	0	0
(AutoNumber)	0		0	0

FIGURE 11.23

Don't panic—users don't deal directly with this data.

- `ElementDescription` stores the information from the field supplied in the `WizExReports` table's `DescriptionField` field.
- `Include` toggles between true and false.
- `OrderNo` is set when an element is to be included in a report. It keeps the order in which the group is printed in the report. It's set to zero if the element isn't included.

Working with the frmWizExReports Form

The frmWizExReports form (in the Chap11.mdb file) has six controls to set up a table with the elements to include for the grouping (see Figure 11.24).

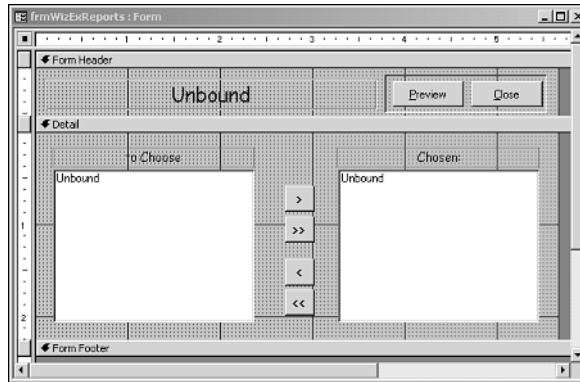


FIGURE 11.24

The frmWizExReports form uses two list boxes to maintain whether to include an element for a group.

The two list boxes shown, lboGetFields and lboFieldsChosen, are actually based on the same row source, with a Boolean field specifying whether the element is included. Here's the SQL string used for the lboGetFields:

```
Select ElementDescription From WizExElements Where
Not Include Order By FieldID;
```

The row source for lboFieldsChosen looks like this:

```
Select ElementDescription From WizExElements Where
Include Order By OrderNo;
```

The differences in these two SQL statements are

- The flag showing whether to Include or Not Include the element.
- The FieldID found in the row source for lboGetFields, which refers to an AutoNumber field created when the Append query is run for populating the WizExElements table. This is for maintaining the order of the original elements used.
- The OrderNo field found in the row source for lboFieldsChosen, used to track the order of choices made for including group elements in the report. This is later referred to directly for the report's group header.

The four command buttons—`cmdSelectCurrent`, `cmdSelectAll`, `cmdUnselectCurrent`, and `cmdUnselectAll`—select and deselect current choices or all choices. I'll discuss the code behind these buttons in a bit, but first I want to take you logically through the form code as the form opens and is used. Let's start by looking at the declarations section:

```
Option Compare Database
Option Explicit
```

```
Dim rstReportToUse As New ADODB.Recordset
Dim cmdUpdateInclude As New ADODB.Command
```

`rstReportToUse` is the variable reference to the `WizExReports` table, specifically to the current report. You see this assigned in the `Form_Open` event.

`cmdUpdateInclude` is a `Command` object used to update an element as to whether to update the `Include` field in the `WizExElements` table. You see this assigned in Listing 11.8 for the `Form_Open` event.

LISTING 11.8 Chap11.mdb: Setting Up `frmWizExReports` for Use

```
Private Sub Form_Open(Cancel As Integer)
    Dim catCurr As New ADOX.Catalog

    DoCmd.SetWarnings False

    catCurr.ActiveConnection = CurrentProject.Connection

    '-- Initialize the Order number
    intCurrOrder = 0

    '-- Getting the information for the current Group by subject
    With rstReportToUse
        .Open "Select * from WizExReports Where ReportName = '" & _
            Me.OpenArgs & "'", CurrentProject.Connection
        DoCmd.Echo True, "Loading " & !SubjectDescription & ", Please wait..."
        '-- Set the form caption and title to reflect the current subject
        Me.Caption = !TitleForReport & " Report"
        Me!lblTitle = !TitleForReport & " Report"
        Me!lblToChoose.Caption = !SubjectDescription & " To Choose"
        Me!lblChosen.Caption = !SubjectDescription & " Chosen"
        '-- Set references up for including and unincluding elements
        '-- in the temp table
        Set cmdUpdateInclude = _
```


LISTING 11.8 Continued

```

        catCurr.Procedures("qryWizExUpdateElementToInclude").Command
'-- Clear the temp table
DoCmd.RunSQL "DELETE * FROM WizExElements;"
'-- Copy the current group by table into the elements table
If IsNull(!KeyField) Then
    DoCmd.RunSQL "INSERT INTO WizExElements (ElementDescription) " & _
        "SELECT " & !DescriptionField & " FROM " & !GroupByTable & ";"
Else
    DoCmd.RunSQL _
        "INSERT INTO WizExElements (ElementKey, ElementDescription) " & _
        "SELECT " & !KeyField & ", " & !DescriptionField & " FROM " & _
        !GroupByTable & ";"
End If
End With

'-- Requery the two listboxes
Me!lboGetFields.Requery
Me!lboFieldsChosen.Requery

'-- Set the initially selected element
Me!lboGetFields = Me!lboGetFields.ItemData(0)

DoCmd.Echo True

End Sub

```

Before you panic, look at the code in sections so it won't be quite so overwhelming. The first piece of code initializes the `intCurrOrder` variable used to set the order of the included elements:

```

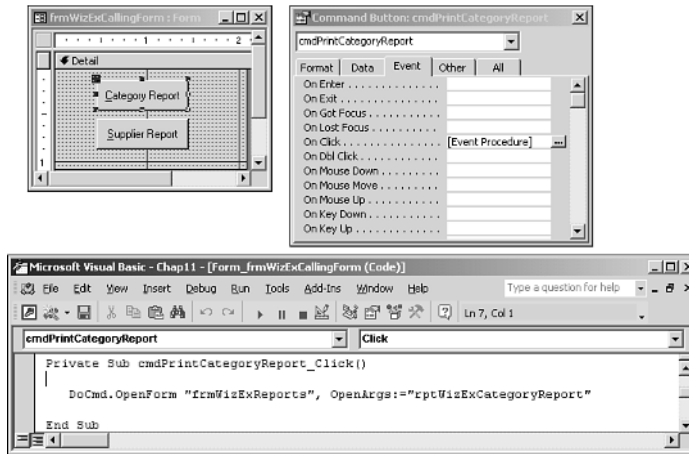
'-- Initialize the Order number
intCurrOrder = 0

'-- Getting the information for the current Group by subject
With rstReportToUse
    .Open "Select * from WizExReports Where ReportName = '" & _
        Me.OpenArgs & "'", CurrentProject.Connection
    DoCmd.Echo True, "Loading " & !SubjectDescription & ", Please wait..."

```

Next, the recordset variable `rstReportToUse` is set to return the record in the `WizExReports` table with the same Report Name (`ReportName` field) as passed by using the form's `OpenArgs` property. You can see this call on the `WizExCallingForm` form.

Figure 11.25 shows the event procedure assigned to the `OnClick` event with `WizExCallingForm` in the background. Notice that named arguments are used.

**FIGURE 11.25**

This is all it takes to call the wizard-like report interface after the tables and queries are set up.

The last line uses the SubjectDescription field in the recordset to create a status message; you just don't know how long it could take to load the WizExElements table initially.

CAUTION

If performance is an issue, you definitely want to test the new report with client data before giving the application to users. This example does an append query in the section after the next piece of code, and could take a few seconds to run, depending on the amount of data.

Generally, users understand that reports can take some time to set up, but this routine might not be suitable for the amount of data. If it's not a lot of data, you have no problem. Your users have to be the judges.

In the next code section, the TitleForReport and SubjectDescription fields are used to populate various labels on the form and the form's Caption:

```
'-- Set the form caption and title to reflect the current subject
Me.Caption = !TitleForReport & " Report"
Me!lblTitle = !TitleForReport & " Report"
Me!lblToChoose.Caption = !SubjectDescription & " To Choose"
Me!lblChosen.Caption = !SubjectDescription & " Chosen"
```

PART II

```
'-- Set references up for including and unincluding elements
'-- in the temp table
Set cmdUpdateInclude = _
    catCurr.Procedures("qryWizExUpdateElementToInclude").Command
```

Next, a reference is set up with the cmdUpdateInclude variable. This query is run when an element is selected or unselected.

The next section of code first clears the WizExElements temp table, and then loads the table with the specified data:

```
'-- Clear the temp table
DoCmd.RunSQL "DELETE * FROM WizExElements;"

'-- Copy the current group by table into the elements table
If IsNull(!KeyField) Then
    DoCmd.RunSQL "INSERT INTO WizExElements (ElementDescription) " & _
        "SELECT " & !DescriptionField & " FROM " & !GroupByTable & ";"
Else
    DoCmd.RunSQL _
        "INSERT INTO WizExElements (ElementKey, ElementDescription) " & _
        "SELECT " & !KeyField & ", " & !DescriptionField & " FROM " & _
        !GroupByTable & ";"
End If
```

The If statements check to see whether there's a value in the KeyField field. If it's null, only the DescriptionField is loaded; otherwise, both fields are loaded. (As mentioned earlier, this is for the cases when the description field is also the key field.)

Finally, the last code section for the Form_Open routine requeries the list boxes because both look at the temporary table for their data, and then set the first element in lboGetFields as selected:

```
'-- Requery the two listboxes
Me!lboGetFields.Requery
Me!lboFieldsChosen.Requery

'-- Set the initially selected element
Me!lboGetFields = Me!lboGetFields.ItemData(0)
```

That's it for the Form_Open routine. Now, I have good news and bad news. The bad news is that there's one more kind of hairy routine that needs to be looked at. The good news is that four of the controls on the form use it, so it can be discussed only once. The routine I'm talking about, the MoveCurrentField() function, is stored with the general functions behind the WizExReports form. Listing 11.9 shows this function.

LISTING 11.9 Chap11.mdb: Toggling an Element from Selected to Unselected Based on the blnInclude Argument

```

Function MoveCurrentField(blnInclude)
    Dim intCurrIndex As Integer, strCurrField As String
    Dim strFromlbo As String, strTolbo As String

    '-- Depending on who called this routine set up the source and destination
    '-- listboxes.
    If blnInclude Then
        strFromlbo = "lboGetFields"
        strTolbo = "lboFieldsChosen"
    Else
        strFromlbo = "lboFieldsChosen"
        strTolbo = "lboGetFields"
    End If

    '-- If an item is chosen, perform task.
    If Not IsNull(Me(strFromlbo)) Then
        '-- Store the current string and index.
        intCurrIndex = Me(strFromlbo).ListIndex
        strCurrField = Me(strFromlbo)
        '-- Update the current elements in the temp table.
        cmdUpdateInclude.Parameters("CurrField") = Me(strFromlbo)
        cmdUpdateInclude.Parameters("CurrInclude") = blnInclude
        '-- Update the order number if the element is to be included.
        If blnInclude Then
            cmdUpdateInclude.Parameters("CurrOrder") = intCurrOrder
            intCurrOrder = intCurrOrder + 1
        Else
            cmdUpdateInclude.Parameters("CurrOrder") = 0
        End If
        cmdUpdateInclude.Execute
        '-- Requery both list boxes.
        Me(strFromlbo).Requery
        Me(strTolbo).Requery
        '-- Set the new selected source element.
        If IsNull(Me(strFromlbo).ItemData(intCurrIndex)) Then
            Me(strFromlbo) = Me(strFromlbo).ItemData(intCurrIndex - 1)
        Else
            Me(strFromlbo) = Me(strFromlbo).ItemData(intCurrIndex)
        End If
        '-- Set the new selected destination element.
        Me(strTolbo) = strCurrField
    End If

End Function

```

End Function

This function, used in the `OnClick` event of the `cmdSelectCurrent` and `cmdUnselectCurrent` command buttons (see Figure 11.26), is also called from both list boxes' `OnDbClick` event.

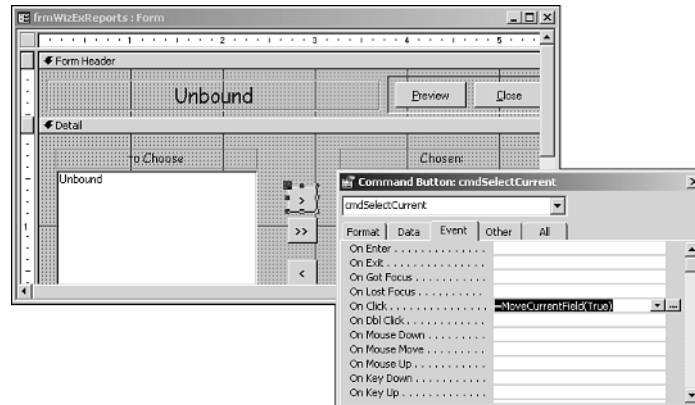


FIGURE 11.26

Centralizing code such as the `MoveCurrentField()` function saves a lot of recoding.

The code is commented quite heavily, so rather than go line by line, I'll lay out the steps that this function performs:

1. The function sets up the list boxes according to whether you will select or deselect an element (a “from” and “to” type of setup).
2. The function stores the current string and index in variables.
3. It sets the parameters of `cmdUpdateInclude` to the element to update. It also specifies whether to change the Include field to true or false, depending on the value passed in with `bInInclude`. Another parameter set is `OrderNo`, which is incremented if an element is being selected, and set to 0 if an element isn't selected. The query is then executed.
4. Both list boxes are requeried.
5. The new source element is selected. If the last source element was the last in the list, the previous choice is selected; otherwise, whichever one comes next is selected.
6. The new selected destination is set to the element just selected or unselected.

Remember that this heavy-duty function does the work of four controls. The last three controls are the command buttons that select and deselect all the elements, and the button that calls the report itself. Listing 11.10 shows the code, which is attached to the `OnClick` events of the select/deselect buttons.

LISTING 11.10 Chap11.mdb: Selecting and Deselecting All Elements

```

Private Sub cmdSelectAll_Click()
    '-- Flag the last elements that aren't included as being included.
    CurrentDb.Execute "qryWizExIncludeAll"
    '-- Requery the two list boxes
    Me!lboGetFields.Requery
    Me!lboFieldsChosen.Requery
    '-- Set the last item as the selected element
    Me!lboFieldsChosen = _
        Me!lboFieldsChosen.ItemData(Me!lboFieldsChosen.ListCount - 1)
End Sub

Private Sub cmdUnselectAll_Click()
    '-- Flag all the elements as not included.
    DoCmd.RunSQL "UPDATE WizExElements SET Include = False,OrderNo = 0;"
    '-- Requery the two list boxes
    Me!lboGetFields.Requery
    Me!lboFieldsChosen.Requery
    '-- Set the first item as the selected element
    Me!lboGetFields = Me!lboGetFields.ItemData(0)
End Sub

```

The last routine calls the report requested, using the WizExReport table's ReportName field. It's attached to the cmdPrintPreview button, located at the top of the form with the caption Print:

```

Private Sub cmdPrintReview_Click()
    DoCmd.OpenReport rstReportToUse!ReportName, acPreview
End Sub

```

That's it for the creation of the form. I know it's pretty extensive, and if you had to deal with modifying it for every report to be printed, you probably wouldn't use it. (I know I wouldn't.) Fortunately, there are only a few steps to add a new report to use the wizard-type interface.

Using the Group Element Wizard with a New Report

When you add reports with this interface, you can do it in about 10 minutes, not including the time it takes to create the report itself. I held off showing the report and record source needed for the report earlier, so I wouldn't be duplicating steps. This way, if you don't want to get into the guts of the form, you don't need to.

Follow these steps to add a report that's grouped by supplier:

1. Create the query first. Look at the query used for the Product by Supplier report, rstWizExSupplierReport. Figure 11.27 shows this query, qryWizExSupplierReport.

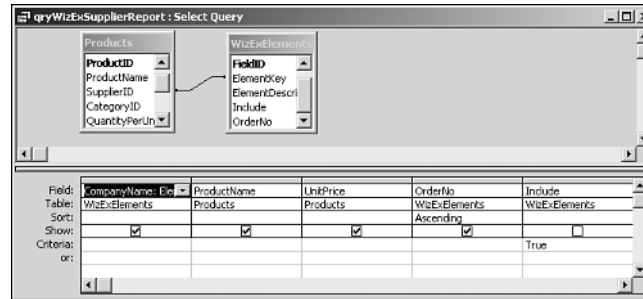


FIGURE 11.27

Notice that the `WizExElements` table is central to this query.

In Figure 11.27, notice how

- SupplierID is joined to the ElementKey field in the WizExElements table.
- OrderNo and ElementDescription are included in the grid.
- The Include field's criteria is set to True. This field isn't shown in the query because it's used only for criteria. (You can see that it won't be in the recordset because the Show check box is unmarked.)

Any detail information you want to include is up to you for the specific report.

2. Set up the report. You can do this with the Report Wizard, as long as you set the OrderNo to be the value grouped by (see Figure 11.28).

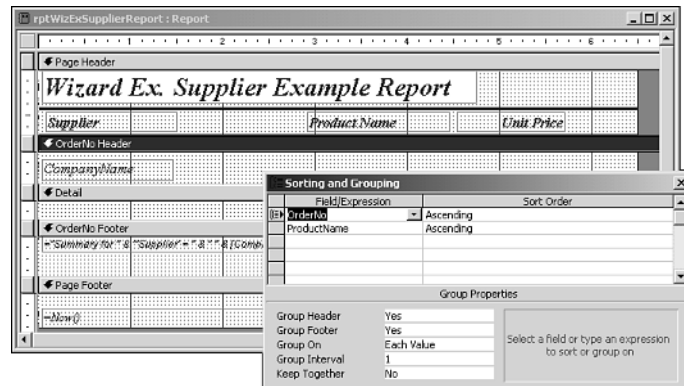


FIGURE 11.28

Even though grouping is set as OrderNo, the data displayed is the element description you want—in this case, CompanyName.

As far as setting up the report goes, that's about it. You can have additional levels under the OrderNo grouping if you want. Be sure to set the record source to the query created in step 1.

3. Add an entry into the WizExReports table to include the new report. Use the following entries for the new report:

<i>Field</i>	<i>Setting</i>
ReportName	WizExSupplierReport
GroupByTable	Suppliers
TitleForReport	Products for Suppliers
KeyField	SupplierID
DescriptionField	CompanyName
SubjectDescription	Suppliers

4. Add the call for the frmWizExReport form, with the new report specified. You can find the example of this for both reports mentioned in this chapter on the frmWizExCallingForm form. Open this form in Design view, and look at the code behind the OnClick event of the cmdPrintSupplierReport command button:

```
Private Sub cmdPrintSupplierReport_Click()  
    DoCmd.OpenForm "frmWizExReports", OpenArgs:="rptWizExSupplierReport"  
End Sub
```

That's all it takes to use this technique. After you do it a couple of times, it shouldn't take too long to add reports and expand on the concept of the grouping wizard.

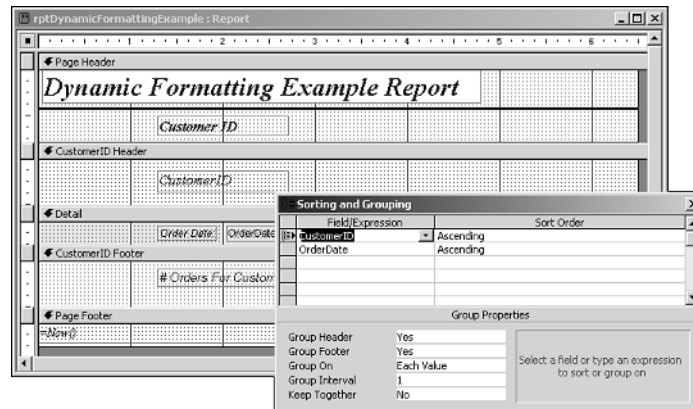
Formatting Reports Dynamically

Sometimes you will want to apply formatting, show or hide controls, or perform calculations at runtime section by section or control by control in a report. Fortunately, Access gives you a great deal of flexibility for adding these features to your applications.

In the first example, colors are alternated for the rows in a report's Detail section. This feature can make it easier for users to read across long rows, and it's a quick way to add a professional touch to your reports.

Looking at the rptDynamicFormattingExample Report

The simple rptDynamicFormattingExample report, in the Chap11.mdb database on this book's Web page at www.sampublishing.com, lists and counts orders by customers. Figure 11.29 shows the report in Design view.

**FIGURE 11.29**

This report displays and counts orders by customer.

The report is based solely on the Orders table. The order records are spaced rather close together and use a small font size to maximize the amount of data visible on each page. This can have the drawback of making it difficult to read across rows. A common technique to alleviate this problem is to alternate the row colors.

A short piece of code in the Detail section's Format event handler takes care of this (see Listing 11.11). The Format event fires after Access retrieves the data belonging in a section, but before that section is formatted for printing or previewing.

LISTING 11.11 Chap11.mdb: Alternating Colors in a Report Section

```
Private Sub Detail_Format(Cancel As Integer, FormatCount As Integer)
    Static intTimesCalled As Integer
    If intTimesCalled Mod 2 = 0 Then
        Me.Detail.BackColor = 65280
    Else
        Me.Detail.BackColor = 8454016
    End If
    intTimesCalled = intTimesCalled + 1
End Sub
```

This code uses a Static variable to track the number of times this routine is called. On even-numbered calls, the BackColor of the Detail section is set to a darker green, whereas on odd-numbered calls, light green is used. Figure 11.30 shows the end result.

Dynamic Formatting Example Report

Customer ID

ALFKI

Order Date:	25-Aug-02	Required Date:	22-Sep-02	Shipped Date:	02-Sep-02
Order Date:	03-Oct-02	Required Date:	31-Oct-02	Shipped Date:	13-Oct-02
Order Date:	13-Oct-02	Required Date:	24-Nov-02	Shipped Date:	21-Oct-02
Order Date:	15-Jan-03	Required Date:	12-Feb-03	Shipped Date:	21-Jan-03
Order Date:	16-Mar-03	Required Date:	27-Apr-03	Shipped Date:	24-Mar-03
Order Date:	09-Apr-03	Required Date:	07-May-03	Shipped Date:	13-Apr-03

Orders For Customer: 6

ANATR

Order Date:	18-Sep-01	Required Date:	15-Oct-01	Shipped Date:	24-Sep-01
Order Date:	06-Aug-02	Required Date:	05-Sep-02	Shipped Date:	14-Aug-02
Order Date:	28-Nov-02	Required Date:	25-Dec-02	Shipped Date:	12-Dec-02
Order Date:	04-Mar-03	Required Date:	01-Apr-03	Shipped Date:	11-Mar-03

Orders For Customer: 4

Page: 1 of 1

FIGURE 11.30

Alternating section colors can jazz up reports while making them easier to read.

Using Conditional Formatting in Reports

The conditional formatting feature is available in reports as well as forms. In this way, you can add powerful data-driven formatting to your text boxes and combo boxes without having to write VBA code. It's quite similar to the conditional formatting feature that first appeared for worksheets in Excel 97.

The first example is a simple inventory report based on the Products table. It uses conditional formatting to highlight products that fall below the reorder level and, more importantly, products with too few units on order.

Looking at the rptConditionalFormattingExample Report

The rptConditionalFormattingExample report can be found in the Chap11.mdb database on this book's Web site at www.sampublishing.com. Figure 11.31 shows the report in Design view.

The report itself is straightforward, except that the UnitsInStock field has formatting conditions. To view, edit, or add these formatting conditions, open the form in Design view, select the UnitsInStock text box, and choose Conditional Formatting from the Format menu. Figure 11.32 shows the resulting dialog.

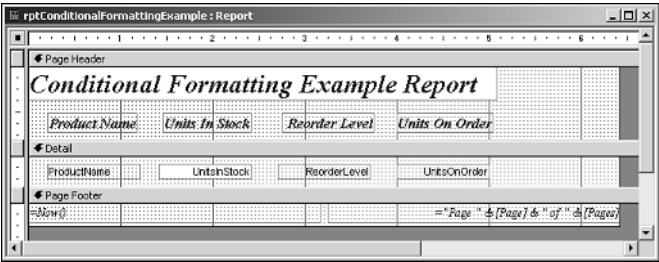


FIGURE 11.31
This report lists products and their inventory information.

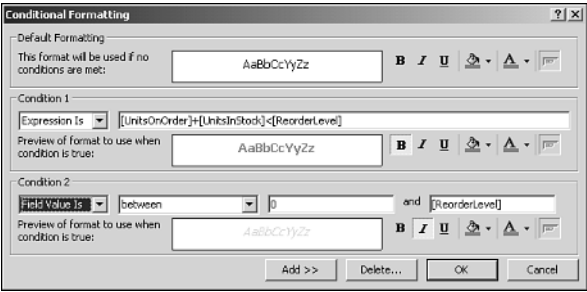


FIGURE 11.32
You can set format conditions for text boxes and combo boxes via the Conditional Formatting dialog.

You can add up to three conditions per control. There are two types of conditions for reports:

- The Field Value Is condition allows you to format based on standard comparisons involving the field's value.
- For extra flexibility, the Expression Is condition can perform formatting based on valid Access expressions that evaluate to True or False.

NOTE

Forms allow you to use a third type of condition, Field Has Focus, which is evaluated when the input focus enters the field in question.

This example uses both types of report formatting conditions. The first condition is an expression that determines whether more units need to be ordered. Select Expression Is from the Condition 1 drop-down list, and then type the following:

[UnitsOnOrder]+[UnitsInStock]<[ReorderLevel]

Values meeting this condition are formatted in bold red.

The second condition simply checks to see whether the current inventory falls below the reorder level. Select Field Value Is from the Condition 2 drop-down list, select Between from the second drop-down list, and then put 0 in the first text box and **[ReorderLevel]** in the last text box. These values are formatted in italic gold. Figure 11.33 shows the resulting report in Print Preview.

NOTE

In this case, the conditions aren't mutually exclusive. In fact, all values meeting the first condition also meet the second. Thus, the order of the conditions is important. Access uses the format of the first met condition.

Product Name	Units In Stock	Reorder Level	Units On Order
Alice Mutton	0	0	0
Aniseed Syrup	13	25	5
Boston Crab Meat	123	30	0
Camembert Pierrot	19	0	0
Carnarvon Tigers	42	0	0
Chai	39	10	0
Chang	17	25	40
Chartreuse verte	69	5	0
Chef Anton's Cajun	53	0	0
Chef Anton's Gumbo	0	0	0
Chocolade	15	25	8
Côte de Blaye	17	15	0

FIGURE 11.33

Here's the report, complete with formatting conditions.

You also can fully manipulate format conditions through code. The next example shows how you can add a format condition on-the-fly in code in response to user input.

Looking at the rptConditionalFormattingInCodeExample Report

You can find the rptConditionalFormattingInCodeExample report in the Chap11.mdb database on this book's Web page at www.samspublishing.com. The basic form design is identical to that of the preceding example (refer to Figure 11.31). The only difference is that the UnitsInStock text box doesn't have any saved format conditions.

This report highlights values in bold red where the current number of units in stock lies within a certain delta of the reorder level. Report users can choose that delta, which means that the format condition must be dynamically manipulated. The necessary code contained completely in the report's Open event handler is shown in Listing 11.12.

LISTING 11.12 Chap11.mdb: Adding a Format Condition in Code

```
Private Sub Report_Open(Cancel As Integer)
    Dim intDifference As Integer
    Dim strUserInput As String
    Dim fmtCondition As FormatCondition

    '-- Get user input
    strUserInput = _
        InputBox("Highlight values within how many of reorder level?", , 0)

    '-- Make sure it's numeric
    If Not IsNumeric(strUserInput) Then
        MsgBox "Value must be numeric."
        Cancel = True
        Exit Sub
    End If

    '-- Make sure it's positive
    intDifference = CInt(strUserInput)

    If intDifference < 0 Then
        MsgBox "Value must be greater than zero."
        Cancel = True
        Exit Sub
    End If

    '-- Format UnitsInStock in bold red if it's within that
    '-- much of the ReorderLevel
    Set fmtCondition = Me.UnitsInStock.FormatConditions.Add(acExpression, _
        , "[UnitsInStock] - [ReorderLevel] <= " & intDifference)
```

LISTING 11.12 Continued

```
fmtCondition.ForeColor = RGB(255, 0, 0)
    fmtCondition.FontBold = True
```

```
End Sub
```

After asking users for a value and performing input validation, the Add method of the text box's FormatConditions collection creates the condition. The desired formatting is then added to the condition.

To see this report in action, preview it and enter an integer in the input box. Figure 11.34 shows a typical result.

Product Name	Units In Stock	Reorder Level	Units On Order
Alice Mutton	0	0	0
Aniseed Syrup	13	25	5
Boston Crab Meat	123	30	0
Camembert Pierrot	19	0	0
Carnarvon Tigers	42	0	0
Chai	39	10	0
Chang	17	25	40
Chartreuse verte	69	5	0
Chef Anton's Cajun	53	0	0
Chef Anton's Gumb	0	0	0
Chocolate	15	25	8
Côte de Blaye	17	15	0

Stocked units lie within a certain delta of the reorder level.

FIGURE 11.34

This report shows a dynamically added format condition.

Summary

The Access report writer is one of the most complete on the market. Between the report writer, VBA, and Access's queries, there aren't many tasks you can't do. If Access can't handle something, you can simply pass the task to Word or Excel by using Automation.

- Chapter 4, "Working with Access Collections and Objects," discusses in greater detail how to work with the various collections, including Forms, Reports, and Controls.
- Chapter 9, "Creating Powerful Forms," does for forms what this chapter does for reports—gives creative ideas for using code behind your forms to get the most out of them.
- Chapter 13, "Driving Office Applications with Automation," covers how to drive other applications from Access and to drive Access from other applications.

Working with Data Access Pages

CHAPTER

12

IN THIS CHAPTER

- **Why Data Access Pages? 356**
- **Saving Time with the Data Access Page Wizards 364**
- **Creating and Enhancing Simple Data Access Pages 369**
- **Grouping Data Access Pages: Reports for the Web 379**
- **Finding Additional Resources 384**

Probably one of the most new exciting and powerful features in Access 2000—and one of the most confusing—were Data Access Pages (DAPs). Data Access Pages are Microsoft's closest attempt yet at melding Access with the Web. DAPs are HTML documents that you create through Access by using Jet or SQL Server for the data source.

NOTE

You will see the terms *Data Access Page*, *DAP*, *data page*, and *page* used for the same item throughout this chapter.

DAPs can be used

- From within Access, mixed with standard forms and reports.
- On a pure Web site on the Internet or an intranet.
- As a combination of both. For example, the input side and maintenance can be performed in Access, with information retrieval and comparison performed on the Web.

This chapter starts off slowly by looking at creating basic DAPs, including the samples that come in the standard Northwind.mdb database. Then you cruise into working with the sample database (Chap12.mdb) included on this book's Web page at www.sampublishing.com.

NOTE

This chapter is located here to flow with the other UI objects (forms and reports). If you're unfamiliar with Internet terminology, you can jump ahead and read Chapter 19, "Using Access with the Internet," before tackling this chapter.

Why Data Access Pages?

One of the first questions asked when Access developers look at DAPs is, "Why do I need to use Data Access Pages when I have forms and reports, which I am so comfortable with?" That's a good question, and one that this chapter should answer for you.

Understanding How Data Access Pages Are Structured

Data Access Pages are primarily separate HTML files with a shortcut in the Access database window. When not using the simple page version, which is a single record displayed and edited at a time, DAPs are built based on the "banded" theory, in which each "band" has grouping and

sorting. This said, DAPs are neither forms nor reports, but a hybrid of the two, mixed with HTML features.

NOTE

In Access 2002, you now can save a form or report as a Data Access Page. Microsoft has gone to a great deal of work not only to save you development time, but also to give you the power of “live” Web reports.

You can see an example of grouping by looking at a simple example of a DAP in Northwind.mdb (see Figure 12.1).

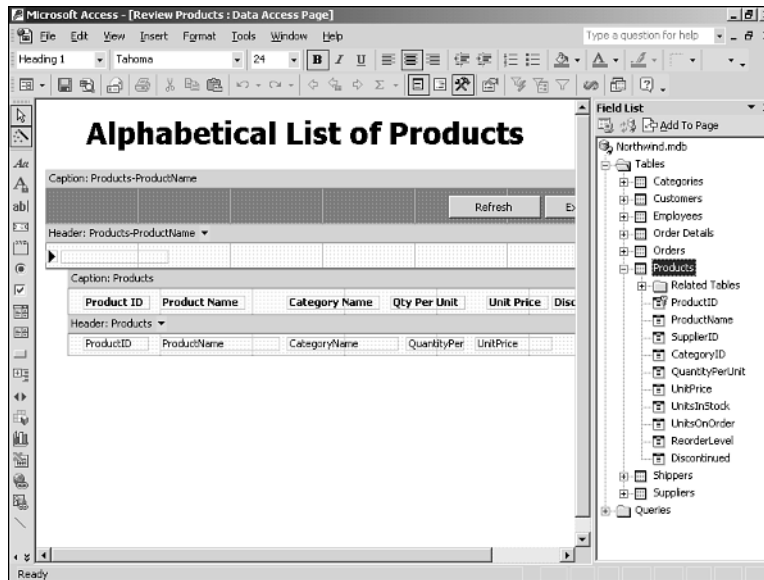


FIGURE 12.1

This Data Access Page, Review Products, is located in the Northwind.mdb sample database that ships with Access.

You can tell which sections can be included for each group, including a Caption section, Header section, Footer section (not shown in Figure 12.1), and Navigation section.

Access 2002 has seen many revisions to the way you design DAPs, one of them being the way you deal with grouping sections. You now have Group-Level property sheets, which you can

get to by right-clicking the section you are interested in, and choosing Group Level Properties (see Figure 12.2). Unlike reports, even your most basic DAP (using data) will include one record source grouping on a page.

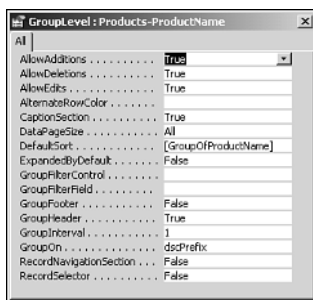


FIGURE 12.2

Sorting and grouping is now handled through a traditional-looking property sheet.

NOTE

In the preceding paragraph, I used the phrase *using data* because you can create static pages that are simply like any other Web page. This might be for creating a home page that doesn't require any data.

However, you *never* want to do this because even that kind of page will attempt a database connection. If it can't find the database, your users will get annoying dialogs. You should use Word or FrontPage in that scenario.

You can set the following properties for each record source:

- **AllowAdditions, AllowDeletes, and AllowEdits.** In Access 2002, you can specify whether to let users have these capabilities for each grouping level. In Access 2000, if you specified that a DAP had more than one grouping level, the DAP data was read-only. This is no longer the case.
- **Alternate Color.** This property lets you specify an alternate color that will be displayed for every other group.
- **Caption Section.** This property displays a Caption section for the group when the group above the current group is expanded. You can't bind any controls in this section. This is closest to the Report Header section of a report.

- **Data Page Size.** This property specifies how many items at a time you want displayed for the group. To display all records, set this property to All. The smaller the Data Page Size value is set, the faster those records will display. Use All only when you know you have a limited number of records in the data page.

NOTE

If you're using the page for editing records one at a time, you can specify to include only one group on the page. The Data Page Size property is then set to 1. (For an example of this, see the Employees DAP in Northwind.mdb in Figure 12.3.) Any action you take that causes this number to become greater than 1 (for example, adding grouping or banding or manually changing it) in Access 2000 would cause your page to cease to be updateable. This is not the case in Access 2002.

12

WORKING WITH
DATA ACCESS
PAGES



FIGURE 12.3

When editing records, you can display one at a time on a page as desired.

- **Default Sort.** This specifies the field in the group to sort on by default.
- **Expanded By Default.** This affects whether to expand the lower group level items by default. You will get better performance when viewing the page by setting all levels of the page's Expanded By Default property to No.
- **Group Filter Control.** Use this combo box or listbox control to filter the group's records on.

- **Group Filter Field.** This field is used to compare against the Group Filter Control to filter the records in the group on.
- **Group Footer.** This property displays a Footer section for the group. As with the Header section, this works primarily as it does with reports, for displaying footer information and summarizing data, if desired.
- **Group Header.** This property displays a Header section for the group. Actually, this section is closest to the form/report's detail section.
- **Group Interval.** This indicates the number of characters or interval of specific data type to group on.
- **Group On.** This specifies the field you want to group on.
- **Record Navigation Section.** As it says, this displays the record's navigation control for the group. It will show after the Footer section, if there is one; otherwise, it will show in the Header section. The Data Page Size property affects the number of records displayed at a time, on the page and in this control.

You can place other controls in the navigation section—they just can't be bound. However, putting unbound controls there is frequently useful. For example, you can replace the barely customizable—and rather slow-performing—Navigation control with your own navigation buttons.

- **Record Selector.** This property allows you to have a record selector display, similar to a form's record selector. A big difference is when you have Record Selector set to True on a group within a group, you will get two record selectors displayed.

Understanding the Navigation Control

In addition to the ActiveX controls and bound HTML controls, one major control is the Navigation control. This control looks similar to the navigation buttons found on the standard form, but contains a few more features in the additional commands it can contain (see Table 12.1).

TABLE 12.1 Properties to Use on the Navigation Control

<i>Property</i>	<i>Description</i>
ShowDelButton	Deletes a record
ShowFilterBySelectionButton	Filters for records that match the selected field
ShowFirstButton	Moves to the first record
ShowHelpButton	Displays Help
ShowLabel	Displays the information specified in the RecordsetLabel property

TABLE 12.1 Continued

Property	Description
ShowLastButton	Moves to the last record
ShowNewButton	Moves to a new record
ShowNextButton	Moves to the next record
ShowPrevButton	Moves to the previous record
ShowSaveButton	Saves the current record
ShowSortAscendingButton	Sorts records in ascending order
ShowSortDescendingButton	Sorts records in descending order
ShowToggleFilterButton	Applies or removes a filter
ShowUndoButton	Undoes the last record

TIP

For the most professional-looking pages, hide any buttons that don't apply to your page. For example, if you have a nonupdateable page (see the note in the preceding section), hide the Delete, Save, Undo, and New buttons. Access won't do this for you.

A really cool feature of Access 2002 is that although you can work with the properties in Table 12.1 in code, you can now have controls appear and disappear using the right-click menu (see Figure 12.4).

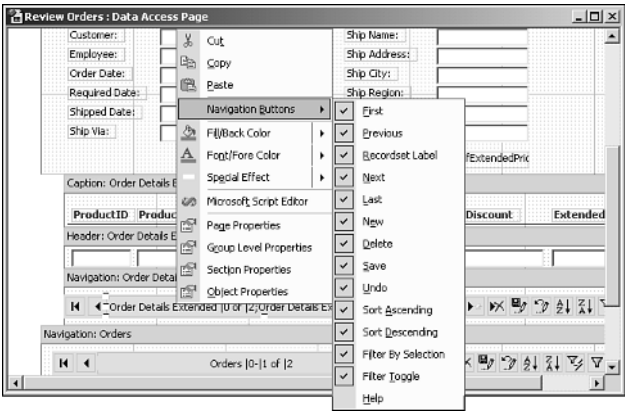


FIGURE 12.4
Choosing which buttons go on your Data Access Page gets easier in Access 2002.

Another Navigation control property to discuss is `RecordsetLabel`, which is a string value. In Figure 12.4, the `RecordsetLabel` property for the Products group level is set to "Orders |0 of |2;Orders |0-|1 of |2" (default). Here is how the string breaks down:

- The first part of the string—Orders |0 of |2;—is the mask for what's shown when the page displays a single record.
- The second part of the string (after the semicolon) represents the label that's shown when the page displays two or more records.
- |0 shows the current record number on a page used for data entry or the number of the first visible record in the group on a grouped page.
- |1 shows the number of the last visible record in the group on a grouped page.
- |2 shows the number of records in the recordset.

NOTE

You might be wondering why I'm going into such detail about the Navigation control so early in the chapter. Getting a handle on this control and the Group Level property sheet was key to helping me understand and get a handle on creating DAPs.

Comparing Data Access Pages to Forms and Reports

For ideas on how to really take advantage of Data Access Pages, look at how they fit with standard Access reports and forms. Traditional Access forms and reports are separated into two purposes: data input (forms) and presentation (reports). Data Access Pages can be used for either purpose. I have even heard DAPs referred to as "interactive reports."

You can accomplish the following with DAPs and their form or report counterparts. Although the lines can blend when it comes to defining which object is used for handling one task or another, these items are the generally accepted choices for solutions.

<i>Task</i>	<i>Counterpart</i>
Analyzing information	Form for parameters, and then calls report
Inputting data	Form
Making projections	Form for parameters, and then calls report
Reviewing data	Form for parameters, and then calls report

Another big difference between DAPs and the other two objects is that DAPs aren't stored in the Access database. They are HTML files to which Access creates a shortcut in the Pages tab of the database window. You can see the DAPs (with .htm extensions) in Chap12.mdb listed in the Explorer in Figure 12.5. You can find Chap12.mdb on this book's Web page at www.sampublishing.com.

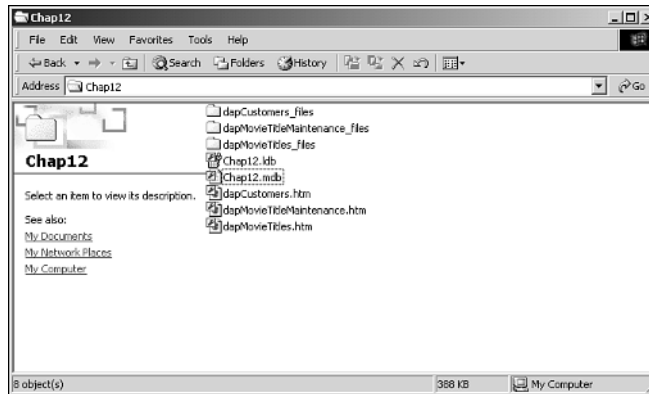


FIGURE 12.5

The .htm files corresponding to the names used for the DAPs in Chap12.mdb can be incorporated directly into a Web page.

Because the connection information is stored with the DAP, using it with Access and the Web is pretty easy.

NOTE

The biggest issue is what happens when you move the data source containing the .mdb/.adp. None of your pages know how to connect anymore. Hence, the ugly `FixDataAccessPageLinks` routine found in the `StartUp` module in `Northwind.mdb`.

Understanding What Users Need for Data Access Pages

For users to take advantage of DAPs with live data outside Access 2002 (using an intranet or the Internet), they need to have Internet Explorer 5.0 or higher and a Microsoft Office XP license.

This generally doesn't make DAPs useful for displaying data on the Internet—because you can't rely on the whole world having a Office XP license (although Microsoft wouldn't mind)—but on your company's intranet, providing you have a site license for Office XP.

Saving Time with the Data Access Page Wizards

As when you create other Access objects, the Access team has created a wizard to help you get going quickly and easily with a new DAP. One way to start off quickly and easily is by using the AutoPage: Columnar feature.

Using AutoPage: Columnar

While in Chap12.mdb's Database window, follow these steps:

1. Choose Page from the Insert menu. In the New Data Access Page dialog, you have the following options: Design View, Existing Web page, Page Wizard, and AutoPage: Columnar.
2. Select AutoPage: Columnar.
3. Select the table or query to base the page on—in this case, tblCustomers (see Figure 12.6)—and then click OK.

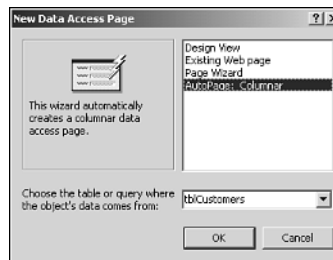


FIGURE 12.6

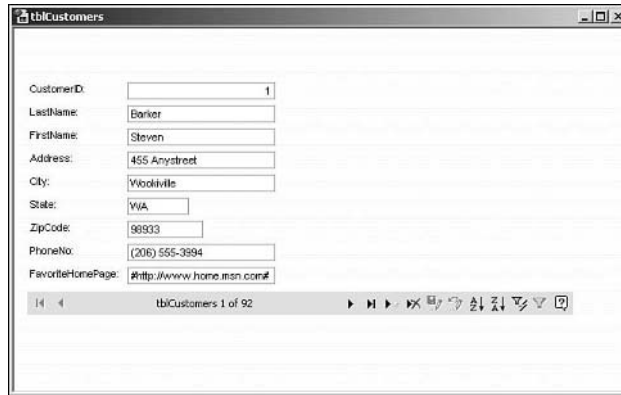
Notice the choices displayed as listed in step 1.

4. In the Save As Data Access Page dialog, save the page as Page1.htm.

NOTE

Because the Save As Data Access Page dialog is the same as the Office Save As dialog, it will default to saving your documents to the \My Documents folder, unless you change the default folder in Explorer or on the Options dialog's General page (accessed from the Tools menu).

You're now presented with a data-editable, if somewhat boring, DAP (see Figure 12.7).

**FIGURE 12.7**

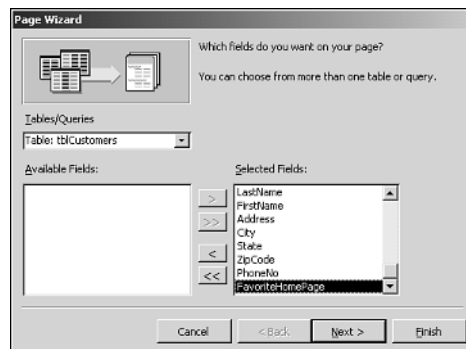
This Data Access Page was created by using the AutoPage: Columnar feature.

Taking Off with the Page Wizard

One way to create a quick DAP that gives you more control over the look of the DAP is using the Data Access Page Wizard. To see how to use the Page Wizard, you will create a page that lets you modify customer information. Use the Chap12.mdb database located on this book's Web page at www.sampublishing.com.

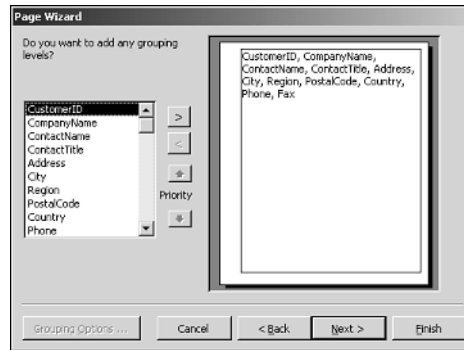
In Chap12.mdb, in the Database window using the Pages tab, follow these steps:

1. Choose Create Data Access Page by Using Wizard from the object list displayed.
2. From the Tables/Queries choices, pick Table:tblCustomers. Click the >> button, moving all the fields over to the Selected Fields column (see Figure 12.8).

**FIGURE 12.8**

Select Table:tblCustomers on the first page of the Data Access Page Wizard.

- Click Next. The next dialog lets you choose some grouping levels (see Figure 12.9). For the purpose of this example, don't add any groups.

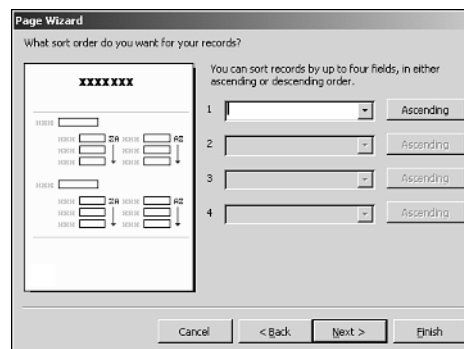
**FIGURE 12.9**

Unlike Access 2000, Access 2002 lets you pick groups for Data Access Pages, and still have them Read/Write.

NOTE

Other than creating a base for an editable page, using this wizard for displaying other views of the data is, in my experience, frustrating. Unlike other wizards, which leave you with something close to the completed object, the Page Wizard leaves you with an object that needs quite a bit of tweaking to be user friendly.

- Click Next. The next dialog asks which sort order you want for the default on the page (see Figure 12.10). Leave this blank.

**FIGURE 12.10**

You can set the default sort order on the page.

5. Click Next. The wizard's last dialog is pretty standard in that it lets you specify a name for the page and whether you want to open the page in Form or Design view. Besides asking whether you want help modifying the page, the wizard also asks whether you want to use a theme with your DAP. To make this interesting, select the option Do You Want to Apply a Theme to Your Page? (see Figure 12.11) and click Finish.

**FIGURE 12.11**

Here we are, finishing up with the Page Wizard.

6. The Theme dialog appears, allowing you to choose a FrontPage theme to apply to the page. For this example, pick Refined (see Figure 12.12). Click OK.

NOTE

If you plan to select a theme, have your Office XP CD handy. Some themes require you to install them even if you want to look at them.

TIP

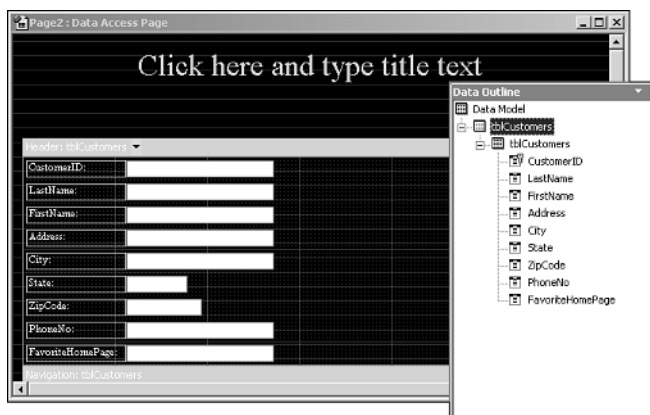
As a rule, pick a theme and stick with it for all your applications. I can't tell you how many times I've gone into an application someone has sent me and found that various themes and colors are used on different forms and reports.

Use the Set Default button in the Theme dialog to set the default for all pages from there on out.

**FIGURE 12.12**

Pick a theme for your Data Access Page.

You then see the final page created. Figure 12.13 shows the finished page in Design view. Save it as **dapCustomers**; you will use it in the next section.

**FIGURE 12.13**

The theme is applied to the final product of the Page Wizard.

NOTE

Although you specified that you wanted the page title to be **Customers**, you will still get a label that says **Click Here and Type Title Text**. Type **Customers** in the (Heading Text) label at the top of the page.

Notice the Data Outline window (new to Access 2002) in Figure 12.13. This window varies from the Field List that was available in Access 2000. The Field List shows you all the available data objects in the project, whereas the Data Outline shows you the data objects for a given Data Access Page. The nice thing about the Data Outline is that you can use it to view and set properties for the various objects if you have the property sheet open.

Creating and Enhancing Simple Data Access Pages

Now you get to see how to enhance the data-entry–style page discussed in the preceding section and in the section “Understanding the Navigation Control.”

Before enhancing a page, however, you need a closer look at the Field List.

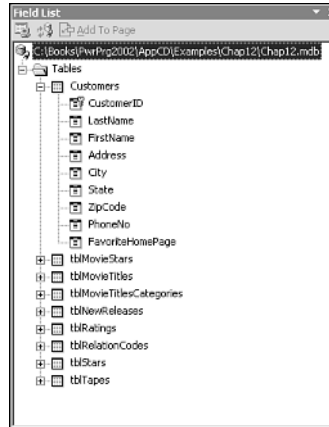
Looking at the Data Access Page Field List

As with standard Access forms and reports, DAPs have a field list from which you can pull fields and place them on your page. Unlike the other objects just mentioned, which list only the fields of the record source specified in the `RecordSource` property, in Access 2002 the Data Access Page Field List displays all the fields and record sources available in the entire database, whether or not they’re being used (see Figure 12.14).

The Data Access Page Field List in Access 2000 had two tabbed pages: **Database**, which is all the Field List shows in Access 2002, and **Page**, which contained the record sources, calculated fields, and data fields specified on the Data Access Page. In Access 2002, the Data Outline takes the place of the Page tab.

Notice that the tables are arranged hierarchically in the Field List, with one level of related tables below each table node. That’s one reason defining system relationships on your database is more important than ever.

More will be discussed about the Field List as it’s used in various areas of creating pages. Let’s continue now by showing how to add hyperlinks to the page.

**FIGURE 12.14**

To pull up the Field List, click its button on the toolbar.

Adding Hyperlinks

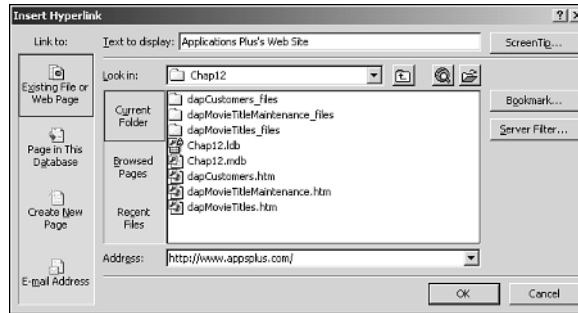
A common task with HTML pages of any kind is to be able to use hyperlinks on them. You can think of *hyperlinks* as shortcuts to locations on the Internet or other files on an intranet. A thorough explanation of what a hyperlink is and the hyperlink dialogs can be found in Chapter 19, which also explains how to add hyperlinks to labels, image controls, and command buttons through code.

Adding an Unbound Hyperlink

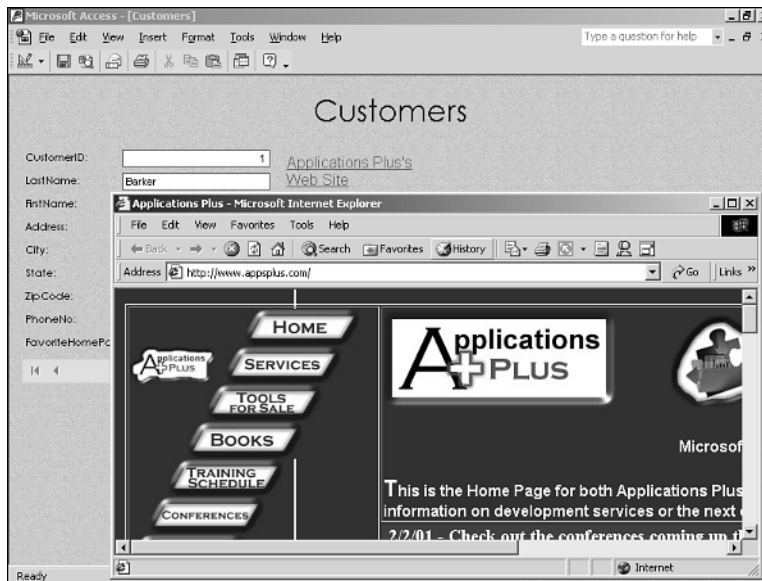
Adding an unbound hyperlink is pretty straightforward. To add a hyperlink to my Web site, www.AppsPlus.com, follow these steps:

1. Open the `dapCustomers` Data Access Page (created earlier) in Design view.
2. Choose Hyperlink from the Insert menu.
3. In the Insert Hyperlink dialog, type **`http://www.appsplus.com/`** in the Address text box (see Figure 12.15).
4. Click OK.

Now when you open the page in View mode, you see the hyperlink displayed on the page at all times. Then, when you click the hyperlink, you are taken to my Web site (see Figure 12.16).

**FIGURE 12.15**


Because this hyperlink is unbound, it remains the same no matter which record you happen to be on.

**FIGURE 12.16**

Okay, a little flagrant advertising here!

Adding a Bound Hyperlink

Creating a bound hyperlink isn't much tougher than using the unbound hyperlink. Notice the FavoriteHomePage hyperlink field in Figure 12.17. The field can't be used to follow the link, however, because it's simply a text box. To follow the link, you need to bind a Hyperlink control to the field.

**FIGURE 12.17**

The field at the bottom of this page is a text box with a Web address for data.

NOTE

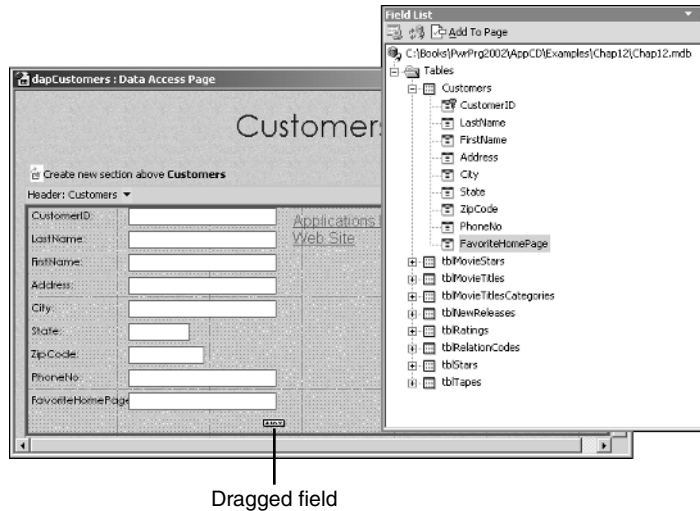
When using hyperlinks on Access forms, the Hyperlink data type causes a hyperlink to be created as the form's control by default. If you use the Hyperlink data type and create a DAP with one of the wizards, it won't work the same way.

In Access 2002, you have to drop a Hyperlink type field onto the DAP from the Field List.

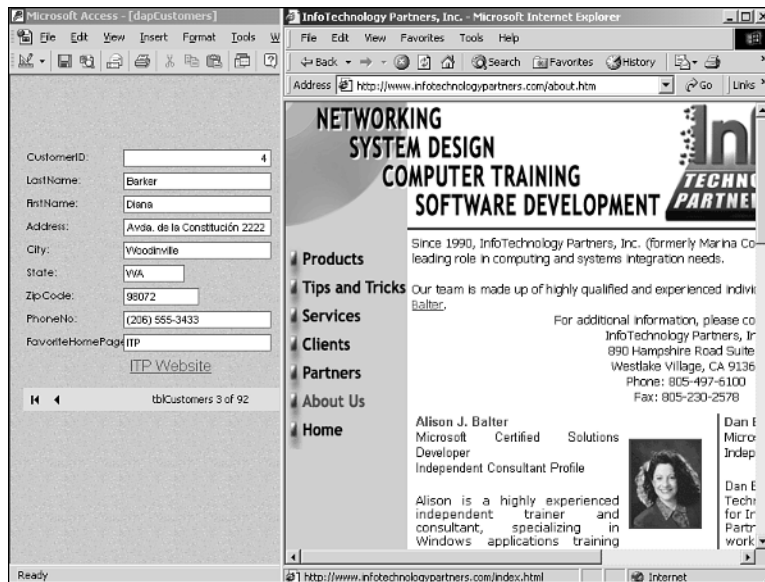
To add a Hyperlink type field onto your DAP, follow these steps:

1. Open the `dapCustomers` page in Design view.
2. Open the Field List by clicking the Field List toolbar button.
3. Click the `FavoriteHomePage` field, and drag and drop it onto the `dapCustomers` page (see Figure 12.18). You will then see a hyperlink with the name `Expr1`. This is because a control is already on the DAP with the name `FavoriteHomePage`.

You might want to expand the control's width. When you go to the same record as in Figure 12.18, you see a hyperlink displayed in the new bound hyperlink. Clicking the new bound Hyperlink control takes you to the specified URL—the Web site for Info Technology Partners (see Figure 12.19).

**FIGURE 12.18**

The field being dragged onto the form is now a bound hyperlink.

**FIGURE 12.19**

Here is the Web site for my knowledgeable colleagues, Alison and Dan Balter.

NOTE

Unlike Access 2000, DAPs in Access 2002 treat HyperLink-type fields like they should in using the display text and address. This is why you see the text ITP Web site, when the hyperlink is

ITP Website#<http://www.infotechnologypartners.com/>

Using Expressions on Data Access Pages

As with standard forms and reports, you can use expressions (or calculated values) on your Data Access Pages. There are a few differences in how you use them in DAPs versus in the standard forms and reports:

- When typing a calculated value on standard forms and reports, precede the expression with an equal sign (=). Although you can do this with Data Access Pages, it's not necessary. Instead—as with a query—you can type a label, a :, and then the expression. Here is an example:

TotalCost: RetailPrice*CopiesPurchased

- When you create a control on forms and reports, you have a unique Name property to set. With a page, you set the ID field to a unique name.

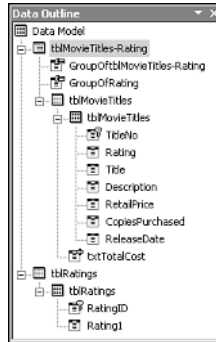
CAUTION

Make sure that you fill in the ID and the alias for the expression you're using. If you don't fill in the ID, and then try to use the control from another with the label name, you will get an error.

If you just leave the alias Expr?, that's what will be displayed in the Data Outline. You can see the calculated fields used on the `dapMovieTitles` page, shown in the Data Outline window in Figure 12.20.

Using Bound Combo and List Boxes

Using the combo and list boxes on a Data Access Page isn't too painful—if you're using them as you would on a page or report for updating a field from another table of values. In fact, in Access 2002 it's not much more work to use a combo box to control your grouping. (Using a combo box in this manner is discussed in the section “Using a Combo for a Group Filter Control.”)

**FIGURE 12.20**

Fill in the alias of the expression and ID property to be thorough.

NOTE

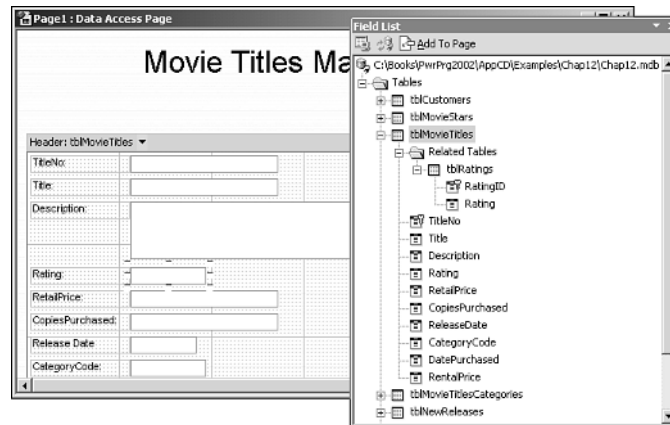
The combo box controls are actually referred to as *drop-down lists* on DAPs. If you place the cursor over what Access developers know as the combo box control in the toolbox, the ScreenTip displays `Dropdown list`. Yet, when a wizard is used, it's the Combo Box Wizard. So for simplicity's sake, when the control is being referred to, it will be called a *combo box*.

NOTE

This section will refer mainly to the combo box control, but everything discussed also applies to the list box control.

In the meantime, let's stick to the more straightforward purpose: adding a combo or list box control on your Data Access Page. To see how to perform this task, create the base page by using `AutoPage: Columnar` with `tblMovieTitles` (in `Chap12.mdb`) and creating a simple page as described earlier in the section "Using `AutoPage: Columnar`." After you create the page based on `tblMovieTitles`, it should resemble the page in Figure 12.21.

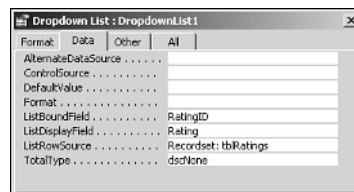
Notice in Figure 12.21 that the control next to the Rating label is selected. Go ahead and delete that control. The label will also be deleted.

**FIGURE 12.21**

This page was created by using AutoPage: Columnar.

As with forms, when you create a combo box, you can use the Combo Box Wizard, with one exception. As with a form, make sure that the wizards are enabled at the top of the toolbox. Then click the combo box control in the toolbox. When creating a form, you would click the field that you want in the field list, and then drag it onto the form. On the form, the Combo Box Wizard would then be invoked. However, with a Data Access Page, the wizard isn't invoked if you click and drag the field onto the page.

To use the wizard, you need to go directly to the form with the new control after you click the combo box control in the toolbox. When you place the control on the page, the Combo Box Wizard is then invoked. Follow the instructions as the wizard takes you through creating a combo box control by using the tblRatings table for the ListRowSource (analogous to a RowSource on the form) of the combo box. After you create your combo box, all the required properties needed to use the combo box will be filled in, except ControlSource (see Figure 12.22).

**FIGURE 12.22**

Here you can see the properties of a combo box (drop-down list) on a Data Access Page.

The following properties are set on the DAP’s combo box:

<i>DAP Property</i>	<i>Form Property Explanation</i>
ListBoundField	Bound column specifies the field’s position.
ListDisplayField	Which field will be displayed.
ListRowSource	RowSource.

ControlSource is used on DAPs and forms. Set the ControlSource of the combo box control to the Rating field.

NOTE

You can display only one field in the combo and list boxes on a DAP. To display more than one field, you must create and display a query expression. Even then, it still won’t be neatly formatted with a dividing line, as on a form/report.

NOTE

As with list boxes on forms, you can set the list box to allow multiple selections. To do this on a DAP, set the Multi property to True and use VBScript to manipulate the control.

Formatting with Themes

As you saw earlier in the Theme dialog in Figure 12.12, you can also apply a theme to a page after it’s created. To do this while designing a page, follow these steps:

1. Choose Theme from the Format menu.
2. Select the theme, or use the default that was set earlier in the section “Taking Off with the Page Wizard.”
3. Click OK.

The theme will now be applied to the Data Access Page.

Using Additional Controls on Data Access Pages

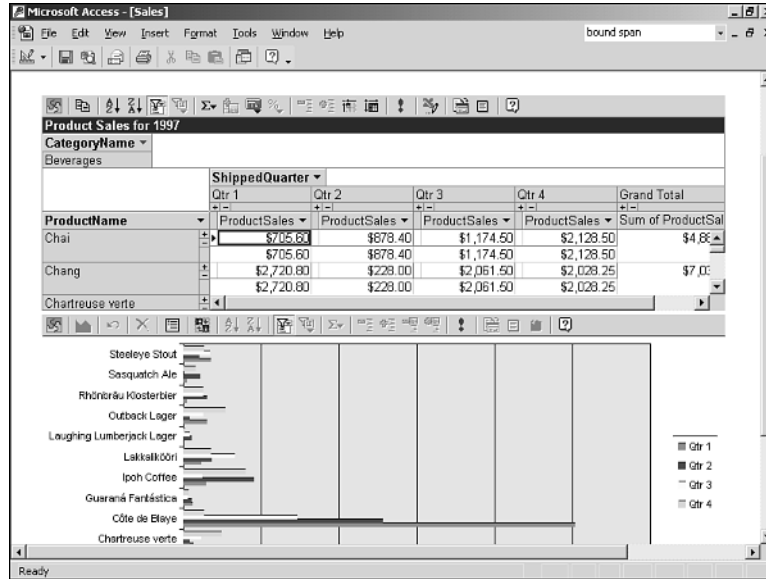
In addition to the standard controls found on forms and reports, you can use a number of other controls on Data Access Pages. Some controls are as follows:

- You can bind the Bound Span control to HTML to perform tasks and calculations. The HTML can also be text in a table, which you can bind to a field.

TIP

If you don't have an updateable page, use Bound Spans to display your data so users don't get the mistaken impression that they can actually change the data. Choose Select Page from the Edit menu. Then, choose Properties from the View menu, and click the Data tab. You can then change the Default Control Type to make the field list create these instead of text boxes. As a result, your pages will look more like reports and less like forms that don't allow you to change data.

- The Scrolling Text control adds an Internet Explorer–specific <MARQUEE> tag to the page. Set the ControlSource to a string and watch it go. It can also be data bound.
- The Image control is the same as on forms and reports. It can be used to display pictures but can't be data bound.
- The Movie control lets you place a movie clip right on your page.
- The Expand control expands and collapses a group level or banding level.
- The Record Navigation control is used to control the record navigation of a group of records.
- Hyperlink controls consist of Unbound and Image. The Image control allows you to place an image on the control that's "hot" with a hyperlink. When you click it, you navigate to the URL specified.
- The Pivot Table Office Web Component allows you to create pivot tables on your Data Access Page.
- The Chart Office Web Component helps you create charts that you can update dynamically. You can see an example of the Pivot Table and Chart Web Components on the Sales page in Northwind.mdb (see Figure 12.23).
- The Spreadsheet Office Web Component allows your users to work with numbers in a spreadsheet within your page.
- Additional ActiveX controls are being created for the Web that can be used with your pages.

**FIGURE 12.23**

The new Pivot Table and Chart ActiveX controls in action.

Grouping Data Access Pages: Reports for the Web

So far, you've created pretty simple pages. Besides the standard columnar page, the really cool feature with Data Access Pages shows up when you create them by using groupings. You can create pages that contain groups with the Page Wizard. The problem is that it still creates the pages using only a columnar-type layout (see Figure 12.24). I want to show how to create a grouped report with a filtered view from scratch.

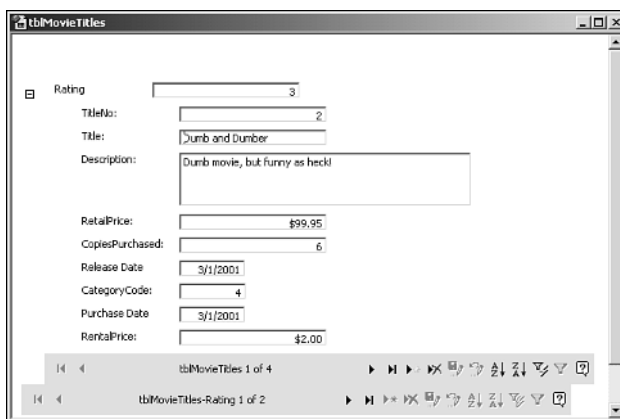
TIP

The following sections compose one big example of creating a multiple grouping page. I have purposefully titled the sections so that when you're creating your own pages, you can turn directly to the step and feature needed.

Creating the Base Page

To begin creating a grouped page, you need to start with a base. While in the Database window of Chap12.mdb, follow these steps:

1. Choose Page from the Insert menu. The New Data Access Page dialog appears.
2. Fill in Choose the Table or Query Where the Object's Data Comes From with **tblMovieTitles**. Unlike with forms, filling in the data source in the New *Object* dialog with a DAP doesn't buy you anything other than opening the field list to the correct table. Your page is still completely unbound when it initially opens in Design view.
3. Because Design view is the default for designing pages in the New Data Access Page dialog, click OK. You're presented with a blank page.
4. Open the Field List via the View menu.
5. Highlight the Title field of the **tblMovieTitles** table, and then drag and drop it onto the page.

**FIGURE 12.24**

This page was created by using the Page Wizard, with a grouping specified on Rating.

You've created the first grouping level and will see a Navigation control appear at the bottom of the page.

NOTE

Before continuing with the current example, it might help if you can see your pages set up as multiple queries, including joins. You will see this with the next step because you will create a relationship on the page between the **tblMovieTitles** and **tblRatings** tables, where **tblMovieTitles.Rating** will be joined to **tblRatings.RatingID**. This will provide multiple benefits to you on the page. This is analogous to joining the two tables in a query.

As with joining tables in a query, you then can display other fields in the page. The creation of the relationship will take place when you drag `tblRatings.Rating` onto the page from the field list.

Creating a Relationship on a Data Access Page

To create a relationship and add the ratings description to the page, follow these steps:

1. Expand the `tblRatings` tree in the Field List.
2. Drag the `tblRatings.Rating` field onto the page. The Layout Wizard will appear, giving you the choice of laying out the fields individually or using a pivot list.
3. Click OK to accept the default, Individual Controls.
4. In the Relationship Wizard dialog, choose `RatingID` field in the `tblRatings` table (see Figure 12.25). Click OK.

12

WORKING WITH
DATA ACCESS
PAGES

NOTE

If you had a system relationship between `tblMovieTitles` and `tblRatings` in the database, Access would figure out what kind of join you wanted (unless it's ambiguous, which it wouldn't be in this case) and you would not be presented with the Relationship Wizard dialog.

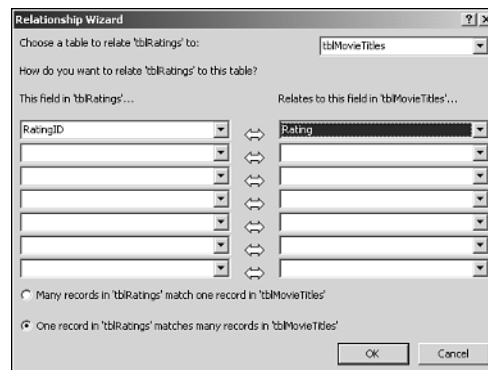


FIGURE 12.25

Create a temporary relationship for the current Data Access Page.

This is still basically a single group. You need to add another level of grouping.

Creating Group Levels with Promote

Now the fun begins. As confusing as pages can be to work with, they have some cool features, including the ability to take a field from one grouping level and promote it into a new grouping level. That's what will happen next with the Rating field.

To promote the Rating field, highlight it and either click the Promote toolbar button or right-click and choose Promote (see Figure 12.26).

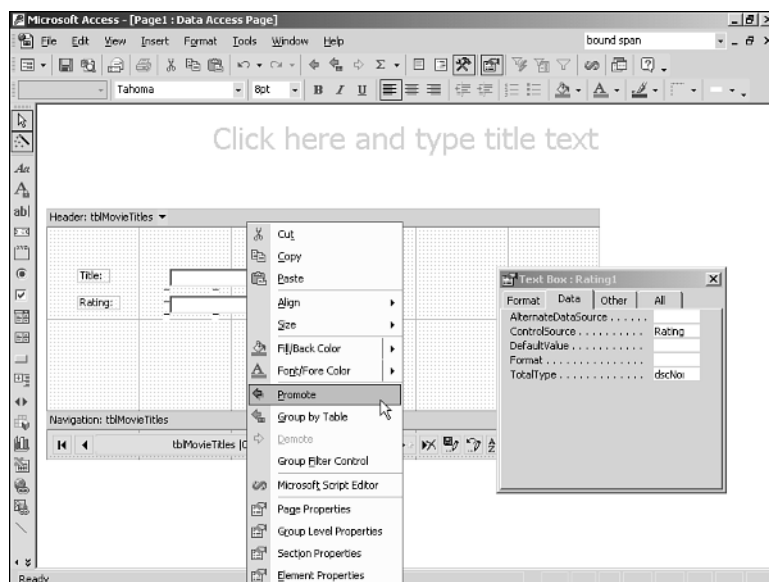


FIGURE 12.26

Promoting a field will create a higher-level grouping.

Promoting the Rating field creates a new grouping called `tblMovieTitles-Rating1`. Before looking at the page in View mode, shrink the height of the Header: `tblRatings` section. Now if you view the page, you will see the two ratings, PG-13 and R, with Expand controls beside them. You can then expand the sections to see the movie titles under each rating.

TIP

To expand the `tblMovieTitles` grouping level by default, open the Group Level property sheet, and then set the Expanded By Default property to True for the `tblMovieTitles-Rating1` group.

Adding a Caption Section

This page is still a columnar report, similar to the Continuous Form view for standard forms. To make this page a true list-type report, you need to add column headings and line up the columns. To add columns, select the section you want to add column titles to—in this case, Header: tblMovieTitles—and select Caption from the right-click menu. Another band will appear on the page.

Highlight the label for the Title field, and then choose Cut from the Edit menu. Now highlight the Caption: tblMovieTitles section and choose Paste from the Edit menu.

You can now add some additional columns on the page from the tblMovieTitles table and line them up as you did with the Title column.

Viewing Your Data Hierarchically with Banded Data Access Pages

This page shows the banded concept pretty well. To simply display records hierarchically, traverse down your view of the data by going through the group levels. Creating more than one grouping level makes the “banded” characteristics of Data Access Pages stand out that much more.

You can now add some additional columns on the page from the tblMovieTitles table and line them up as you did with the Title column.

Using a Combo for a Group Filter Control

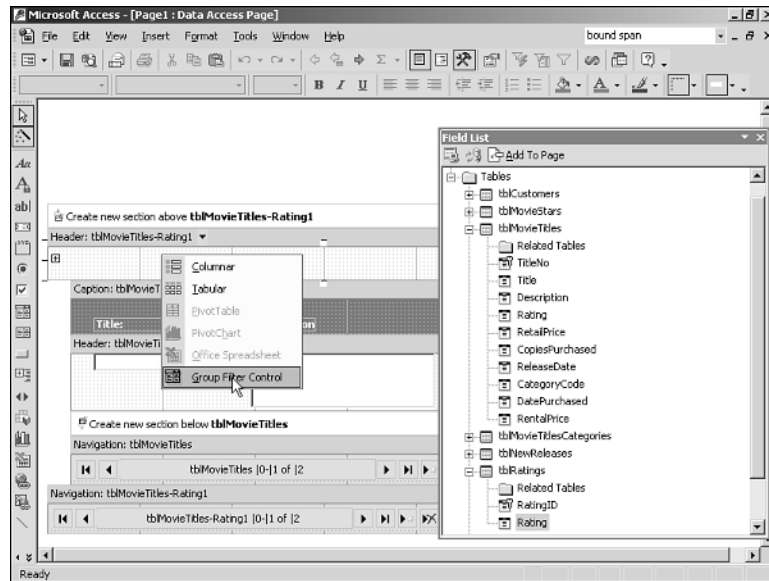
Now you’ll see how to create a combo box control that will allow you to display one higher-level value at a time and filter the lower-level grouping based on the combo box value. Follow these steps for creating a Group Filter control:

1. Delete the Rating field in the Header: tblMovieTitles-Ratings1 section.
2. With the tblRatings table expanded in the Field List, highlight the Rating field.
3. Click the Combo Box toolbox button.
4. With the right (not left) mouse button down, drag and drop the Rating field into the Header: tblMovieTitles-Rating1 section.

CAUTION

If you don’t do step 3 or hold the left mouse button down in step 4, you’ll have no end of grief. You won’t be presented with the menu asking to make the control into the Group Filter control.

- As soon as you drop the field, a pop-up menu will appear, giving you the choice of making the field a regular control or a Group Filter control (see Figure 12.27). Choose Group Filter Control.

**FIGURE 12.27**

Create a Group Filter control on the page.

Now if you pick a value from the GroupOfRating combo box (you probably will want to change that label's caption), you will see all the movie titles with that rating.

NOTE

Notice that when you click the combo box, the list is limited to those actually used by the movie titles. This is another feature reminiscent of queries in that only those values found in the `tblRatings` and `tblMovieTitles` tables will appear.

Finding Additional Resources

Normally, I don't point to outside sources for information on a subject in this book because I try to cover everything myself (of course), but because Data Access Pages span so many

environments and applications (such as Office, intranet/Internet, VBScript, and FrontPage), it's hard to cover all of them satisfactorily. Knowing this, I will point to other resources that cover DAPs in the various areas.

Although DAPs are a fairly new technology, Microsoft has taken great pains to come up with some documentation for using them. These take the form of sample applications and white papers.

At the time of this writing, white papers and sample applications are in the works and are supposed to be stored on Microsoft's Web site at <http://www.Microsoft.com/OfficeDev>.

Some intended papers are as follows:

- *Programming Data Access Pages* is a pretty extensive paper showing everything you want to know about using VBScript with Data Access Pages.
- *Deploying Data Access Pages on the Internet* discusses the issues with using your DAPs with the Web, including working with DAPs in FrontPage and using frames on your Web site.
- *Creating Secure Data Access Pages* looks at the various issues with using the various back ends with DAPs: Jet, MSDE, and SQL Server.

Some or all white pages listed here might change or not even be included. Get busy with the information you receive here and create your own pages!

Summary

Data Access Pages are definitely one of the hottest features of Access. Although working with them can be somewhat confusing, the power they add to applications that need to work with the Web is worth the effort.

For more information on the Internet features available in Access 2002, see Chapter 19, "Using Access with the Internet."

Extending Access with Interoperability

PART



IN THIS PART

- 13 Driving Office Applications with Automation 389
- 14 Programming for Power with ActiveX Controls 423
- 15 Extending the Power of Access with API Calls 461
- 16 Extending Your VBA Library Power with Class Modules and Collections 491
- 17 Creating Your Own Wizards and Add-ins 511
- 18 Manipulating the Registry with VBA 543
- 19 Using Access with the Internet 573

Driving Office Applications with Automation

CHAPTER

13

IN THIS CHAPTER

- Working with Automation 392
- Running Other Applications from Access with Automation 397
- Driving Access from Another Application with Automation 418

Interoperability is a major feature of the Office products. One kind of interoperability is working in one application, such as Access, and manipulating an object in another application, such as a Word document. For example, interoperability lets you print an Access report to a Rich Text Format file, open a session of Word from Access with the report as its document, format the report in any style desired, and print the report in Word. This might not seem like a big task, but trying to perform it before Automation (OLE prior to Office 97) existed would have been pretty difficult, especially before *DDE (Dynamic Data Exchange)*. Automation performs this feat almost seamlessly.

From Access, Automation allows you, the developer, to link to objects (such as documents, spreadsheets, and so on) from other applications, or embed objects from other applications in forms, reports, or tables, thereby allowing the native application to perform the tasks needed. In this case, Access is the client (also called *controller*); the other application being driven, such as Word, is the server.

NOTE

There are few tasks that you can't perform by using Automation. Unfortunately, not all Windows applications can use Automation. In general, those that use DDE can't use Automation.

You can take advantage of Automation in Access in two ways:

- **Through the user interface.** You can have an object embedded or linked in a table, and then use that table as a record source for a bound form. To get to the server application, double-click the embedded or linked object to initiate *in-place activation*. This allows you to edit, for example, a Word document by using Word menus while still in Access.
- **Through VBA.** You can use the Automation method discussed in this chapter to create an object-type variable and manipulate it by using VBA.

Automation with VBA gives you about as much control as you need in programming objects in other applications. By creating the `Object` type variable, the client can use the server's native commands (methods) to perform the necessary functions. Figure 13.1 diagrams a generic Automation session.

Through Automation, you can manipulate the following Office products and objects. The following classes are used by Automation functions, which are discussed in the next section.

<i>Application</i>	<i>Object</i>	<i>Class</i>
Access 10.0	Application	Access.Application
Excel 10.0	Application	Excel.Application
	Chart	Excel.Chart
	Worksheet	Excel.Worksheet
Graph 10.0	Application	Graph.Application
	Chart	Graph.Chart
Project 9.0	Application	MSPProject.Project
PowerPoint 10.0	Application	PowerPoint.Application
	Presentations	PowerPoint.Presentations
Word 10.0	Application	Word.Application
	Document	Word.Document
Outlook 10.0	Application	Outlook.Application
	Journal Item	Outlook.JournalItem

Generic Automation Model

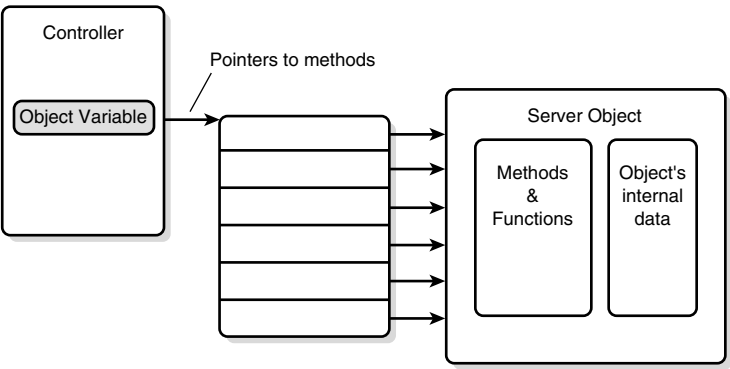


FIGURE 13.1
An Automation client can set up a reference to a server application.

NOTE

If you are using Access 2002 with previous versions (7, 8, and 9) of other Office products, you can still automate them as long as you point to the right library. Watch out, however, because the Office applications' object models probably vary from the current versions.

Working with Automation

Before actually using Automation, you need to set up references to the various application object libraries, also known as *TypeLibs*. By setting up references, you can strongly type variables (so that you can use data types such as *Sheet* and *Chart* instead of *Object*), get early binding, and use constants defined in the *TypeLib*. (Declaring Automation variables is described in more detail in the next section.)

Suppose that you want to set a reference to the Microsoft Word 10.0 *TypeLib*. In the database where you want to set up a reference, open the module editor with any module, and then follow these steps:

1. Choose References from the Tools menu.
2. Select Microsoft Word 10.0 Object Library (see Figure 13.2) and click OK.

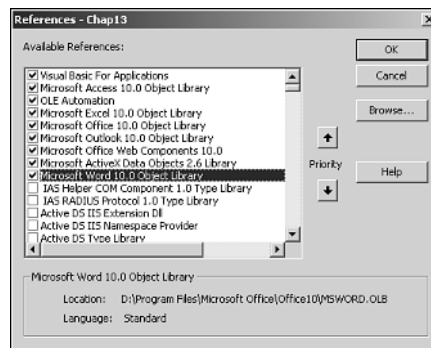


FIGURE 13.2

References are specific to individual databases and must be set up for each .mdb or .adp file.

TIP

If you choose not to set references in your database (or forget to), you can still use Automation to access the server application. Use the *Object* type when declaring variables to work with objects located in the server application.

NOTE

If you don't have an object library that a reference is set to in a database, you will get a compile error with that database. One of first things I do when I get a compile error for no apparent reason is open a module and choose References from the Tools menu. I look for a reference with the word MISSING beside it. If I know it's a reference that isn't there, I unmark the entry. This means, of course, I can't run that particular routine, but my database will then compile. The Missing Reference error will occur if Microsoft Project isn't installed on your system and you open the Chap13.mdb; your database also won't compile.

After the reference is created to the specific object library, you can pull up the library and its objects in the Object Browser, located in the VBA modules. The Object Browser allows you to examine the object models for any application you've referenced. For more information about the Object Browser, see Chapter 2, "Coding in Access 2002 with VBA."

To bring up the Object Browser, open any module and then press F2. Figure 13.3 shows the Object Browser available in Access.

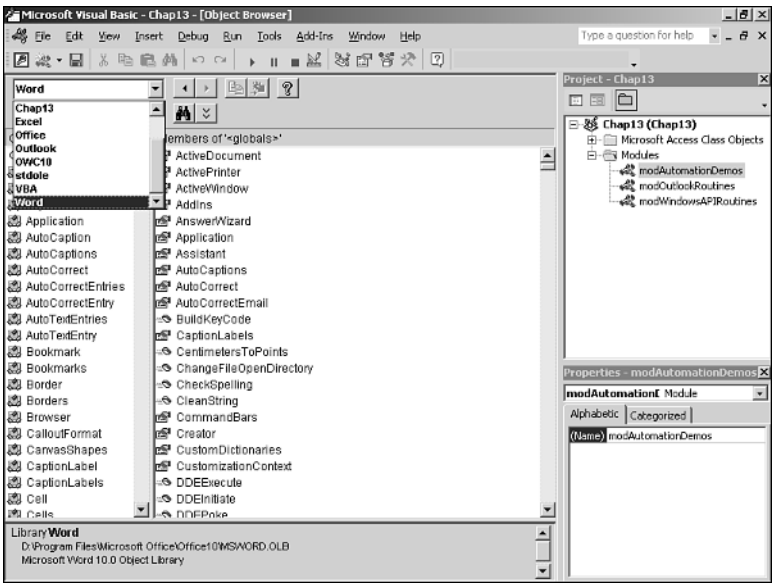


FIGURE 13.3
The Object Browser is available in applications that use VBA.

NOTE

Setting up references also works for ActiveX controls. After they're set up, you can bring up their objects in the Object Browser.

After setting up the references you need, you're ready to start coding for Automation. To do this, you must first declare any object variables needed.

Declaring Object Variables in VBA

Declaring object variables in VBA is the same as declaring other variables used in VBA—by using the `Dim` statement. (For more information about declaring variables using VBA, see Chapter 2.) The following is the declaration statement for a variable used as a Word document:

```
Dim docWord as Word.Document
```

You use this variable type for objects in referenced servers. When the Excel object library is referenced, you can declare a `Worksheet` variable as such:

```
Dim shtCustomers as Excel.Worksheet
```

If you declare a variable of an Access object, such as a database, from another application as the client, you can specify a `Data Access Object (DAO)` variable type:

```
Dim dbOLEDemo as DAO.Database
```

NOTE

When automating Access from another application—that is, when you use Access as the Automation server—don't forget that you need to set references to the Access objects. To do so, enter the VBA module of whatever product you're using as the client. Then choose `References` from the `Tools` menu and select the `Microsoft Access 10.0 Object Library`.

For the purpose of the examples in this chapter, the following code shows the various object variables defined globally as they appear in the declarations section of the `modAutomationDemos` module. You can find this module in the `Chap13.mdb` database on this book's Web page at www.sampublishing.com.

```
Option Compare Database  
Option Explicit
```

```
Public appWord as Word.Application
Public appProject As MSPProject.Application
Public appExcel as Excel.Application
```

NOTE

To have the server application stay open after the routine ends, declare the Object variable as Public.

After you define the variables, have the client create an Automation instance of the variable class you need to use. To do so, you have two different functions: `CreateObject()` for new objects and `GetObject()` for existing objects.

Using the `CreateObject()` Function

The `CreateObject()` function takes one parameter—the class of object to create. An example of the syntax for the `CreateObject()` function, creating a `Word.Application` object, follows:

```
Dim appWord as Word.Application
'Create the object variable
Set appWord = CreateObject("Word.Application")
```

Sometimes you might want to include a version number when you're creating an object. Some classes can't have the version number included. When the version isn't included, the following can occur:

- If a copy of the server application is running, regardless of the version, that copy is used. For example, if you have Word 2000 running when you execute the preceding code, that version of the application is run.
- If no version of Word is running and Microsoft Word 2002 has been installed on your system more recently than Word 2000, Word 2002 is run.

For example, if you're using a Microsoft Excel object, you would use the following line of code:

```
Set objProject = CreateObject("Excel.Application.10")
```

You see more examples of using the `CreateObject()` function later in this chapter.

TIP

Rather than use the `CreateObject()` function, you can use the `New` keyword in a declaration statement to establish the reference when the variable is first used:

```
Dim appExcel as New Excel.Application
appExcel.Visible = True
```

You don't have to use the `Set` statement after declaring the variable and before using the variable.

However, the recommended way is to declare the variable, and then use the `Set` statement with the `New` keyword:

```
Dim docWord As Word.Document
Set appWord = New Word.Application
```

This is recommended so that you can declare the variable in a module declaration, but then not have instantiated until it's needed.

Using the `GetObject()` Function

Use the `CreateObject()` function when you want to create a new instance of the server class. Use the `GetObject()` function to create a reference to an already existing instance. In a Word document, the server class is the full path and filename—for example, `Set objDocument = GetObject("c:\temp\Example.doc")`

The actual syntax for `GetObject()` is

```
Set objVariable = GetObject(strPathName, strClass)
```

In this syntax,

- *objVariable* is the object variable.
- *strPathName* is the full path and filename of the object in which you're creating a reference.
- *strClass* is the Automation class type. This value is the same as that used for the `CreateObject()` variable.

Based on how you use these parameters, different results occur:

- If a *strPathName* is given, that object is referenced.
- If *strPathName* is set to an empty string (""), a new instance is created of *strClass*.

- If `strPathName` is omitted but `strClass` is specified, a running copy of the application specified in `strClass` starts. If no `strClass` application (the application represented) is running, an error is returned.

The `GetObject()` function also lets you specify part of a file, such as a range in a spreadsheet.

Cleaning Up When Done with an Object

After you're through with an Automation instance, you want to set the object variable to `Nothing`. The syntax for this is as follows:

```
Set objDocument = Nothing
```

The best way to learn Automation is to examine code samples. To help you learn the code, I included several examples in the `Chap13.mdb` sample database, available on this book's Web page at www.sampublishing.com. When you open it, the `frmAutomationDemoCalls` form opens (see Figure 13.4).

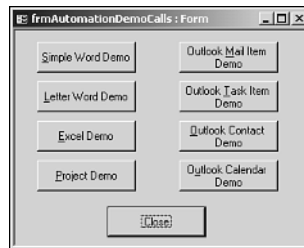


FIGURE 13.4

The eight demos included in this form show how to control other Office applications from Access.

Running Other Applications from Access with Automation

The number of applications you can drive from Access grows with every version of Access. You can automate all Office 2000 applications, including Access itself.

TIP

When using Word or any other application that has a set reference to it, you can receive help while programming for that application, even while in Access. You just have to make sure that you've included VBA Help for that application when you installed Office 2000.

Driving Word from Access

This example for automating Word from Access is fairly straightforward. It performs these functions:

1. The routine opens a new Word document.
2. It inserts some text into the document.
3. It formats all the text as bulleted.

Listing 13.1 shows the code that performs these steps. It can be found in the modAutomationDemos module in the Chap13.mdb database.

LISTING 13.1 Chap13.mdb: A Simple Automation Example

```
Function AccessToWordAutomation()  
    Dim docWord As Word.Document  
    Dim rngCurrent As Word.Range  
    On Error GoTo Error_AccessToWord  
  
    Set appWord = New Word.Application  
    Set docWord = appWord.Documents.Add() '-- Create the document variable  
    appWord.Visible = True  
    Set rngCurrent = docWord.Content  
  
    With rngCurrent                                '-- Create some text  
        .InsertAfter "This is a simple example for Word!"  
        .InsertAfter vbCrLf  
        .InsertAfter "Another Line"  
        .ListFormat.ApplyBulletDefault  
    End With  
  
    Exit Function  
  
Error_AccessToWord:  
    AppActivate "Microsoft Access"  
    Beep  
    MsgBox "The Following Automation Error has occurred:" & vbCrLf & _  
        Err.Description, vbCritical, "Automation Error!"  
    Exit Function  
  
End Function
```

In some cases, you want to fill in data in a Word document while still in Access. Normally, you need to create a mail merge document that users can implement. The problem with mail merge,

however, is that users tend to change their minds about where they want things. For example, in one case they want a name at the beginning of the document; but the next time, they want it at the end. They can change the document to suit their needs. If you are using mail merge functions, you now have to change the data's layout to match user changes. This problem is demonstrated in the next example in which a form letter is created to send to customers with overdue video tapes (see Figure 13.5).

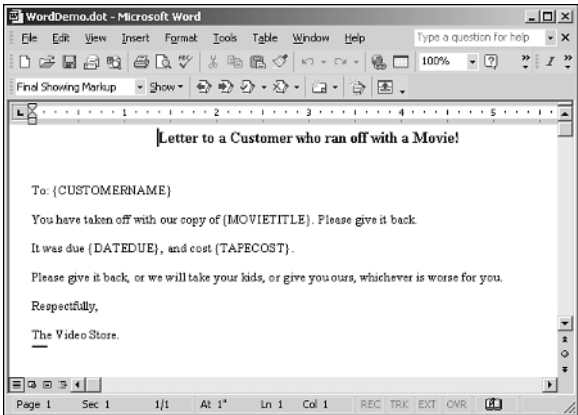


FIGURE 13.5
This document template, WordDemo.dot, is copied to another file to protect it.

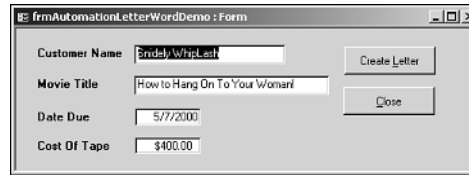
Following are items needed to create the finished document:

- You need a document created in Word that has keywords enclosed with braces ({}). The braces are optional because the keywords can be anything you decide. Braces are used here because it's unlikely that you'll have actual words in the document text with braces around them. You can find the template for this document, WordDemo.dot, on this book's Web page at www.sampublishing.com.
- A table of keywords in Access, named `tblAutomationWordReplaceCodes`, is set up with all the fields used on the form (see Figure 13.6).

CodeToReplace	ReplaceWithFieldName
{CUSTOMERNAME}	Forms!frmAutomationLetterWordDemo!txtCustomerName
{DATEDUE}	Format(Forms!frmAutomationLetterWordDemo!txtDateDue,"dddd, mmmm d, yyyy")
{MOVIE TITLE}	Forms!frmAutomationLetterWordDemo!txtMovieTitle
{TAPECOST}	Format(Forms!frmAutomationLetterWordDemo!txtTapeCost,"\$###,###.00")

FIGURE 13.6
This table contains the keywords (left column) and the replacement values for the keywords (right column).

- All fields that supply the replacement values for the keywords are located on the form `tblAutomationLetterWordDemo` (see Figure 13.7). This form also contains the code that performs the operation.

**FIGURE 13.7**

This form contains all the fields for which the searches and replaces are performed.

Now that you've seen all the pieces, the code follows these steps:

1. It gets the application's path and establishes the final filename.
2. Before copying the template to the final filename, the routine deletes the current final file, if it already exists; then the routine copies the template.
3. The routine creates the Automation instance of Word.
4. The code adds a document to the Documents collection and opens the document, also creating a reference to the document.
5. The routine opens the table of replace codes and cycles through them.
6. It activates Word by making it visible.
7. For each replace code, the routine does the following:
 - It gets the actual replacement value and then uses the Word Find object off the Content object.
 - For the replace code `{MOVIEITITLE}`, it sets the replacing value to boldface and italic. (This step is performed only once—for the `{MOVIEITITLE}` keyword, not for the other keywords.)
 - It executes the replace method.

To retrieve the actual value desired, use the `Eval()` function to evaluate strings passed to it, returning whatever result is retrieved. (For more information on `Eval()`, search for it in the Answer Wizard.)

Listing 13.2 shows the actual code to perform these actions. You can find the code for the `cmdCreateLetter_Click` event procedure on the `OnClick` event for the `cmdCreateLetter` command buttons.

LISTING 13.2 Chap13.mdb: Creating a Custom Letter

```

Private Sub cmdCreateLetter_Click()
    Dim rstReplaceCodes As New ADODB.Recordset
    Dim strCurrAppDir As String
    Dim strFinalDoc As String
    Dim varReplaceWith As Variant
    Dim docWord As Word.Document

    On Error GoTo Error_cmdCreateLetter_Click

    '-- Get the application's path and establish the final filename
    strCurrAppDir = CurrentProject.Path
    strFinalDoc = strCurrAppDir & "\DemoTest.doc"

    '-- If the final file is already there, delete it.
    On Error Resume Next
    Kill strFinalDoc
    On Error GoTo Error_cmdCreateLetter_Click

    '-- Copy the template so it doesn't get written over.
    FileCopy strCurrAppDir & "\WordDemo.DOT", strFinalDoc

    '-- Create the OLE instance of Word, then activate it.
    Set appWord = New Word.Application

    '-- Create the object variable
    Set docWord = appWord.Documents.Add(strFinalDoc)

    appWord.Visible = True

    '-- Open the table of replace codes then cycle through them.
    rstReplaceCodes.Open "tblAutomationWordReplaceCodes", _
        CurrentProject.Connection

    Do While Not rstReplaceCodes.EOF
        '-- Get the actual value to replace with, then use the Word replace.
        varReplaceWith = Eval(rstReplaceCodes!ReplaceWithFieldName)
        varReplaceWith = IIf(IsNull(varReplaceWith), " ", CStr(varReplaceWith))
        With docWord.Content.Find
            If rstReplaceCodes!CodeToReplace = "{MOVIEITITLE}" Then
                With .Replacement
                    .ClearFormatting
                    .Font.Bold = True
                    .Font.Italic = True
                End With
            End If
        End With
    Loop

```

LISTING 13.2 Continued

```

End If
.Execute FindText:=rstReplaceCodes!CodeToReplace, _
        ReplaceWith:=varReplaceWith, Format:=True, Replace:=wdReplaceAll
End With
rstReplaceCodes.MoveNext
Loop

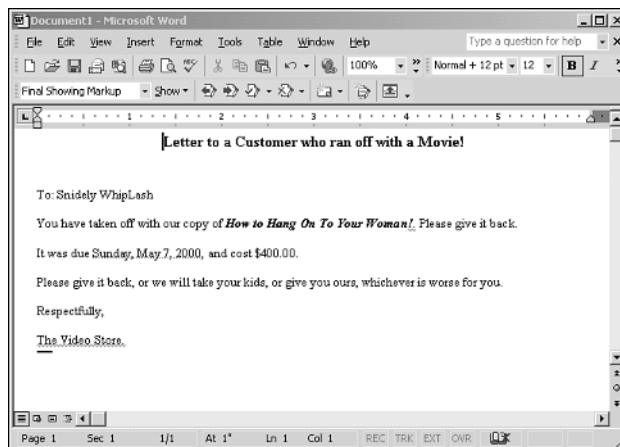
Exit Sub

Error_cmdCreateLetter_Click:
Beep
MsgBox "The Following Error has occurred:" & vbCrLf & _
        Err.Description, vbCritical, "OLE Error!"
Exit Sub

End Sub

```

The order in which the fields are used doesn't matter for this method. If keywords are deleted from the document, Word just doesn't find them during the search-and-replace operation. Figure 13.8 shows the final document for this example.

**FIGURE 13.8**

Here I had Word perform special text selection from Access by selecting and boldfacing the video's title.

Driving Excel from Access

Manipulating Excel isn't much tougher than manipulating Word. The hard part is knowing which Excel properties and methods to use. In this example, a spreadsheet is populated with

data from an Access table named `tblProjects`. The data represents tasks that are actually used by Microsoft Project in another example. The duration for each task is stored and totaled after the last row.

This example performs the following steps to load an Excel spreadsheet from Access by using Automation:

1. It opens the `tblProjects` table.
2. It creates a reference to a new Workbook, using the Excel instance declared with the `New` statement.
3. The routine creates a reference to a new worksheet, and then activates Excel by setting the `Visible` property to `true`.
4. It creates the column headings.
5. It loops through the `tblProjects` recordset, adding the cells to the spreadsheet.
6. In the row following the last value, the routine creates a calculated cell.

Listing 13.3 shows the code, which is in the `modAutomationDemos` module.

LISTING 13.3 Chap13.mdb: Writing Data to an Excel Spreadsheet from Access

```
Function AccessToExcelAutomation()
    Dim rstProjects As New ADODB.Recordset
    Dim intCurrTask As Integer
    Dim wbkNew As Excel.Workbook, wksNew As Excel.Worksheet
    Dim rngCurr As Excel.Range

    On Error GoTo Error_OLEAccessToExcel

    '-- Open the current database and projects table
    rstProjects.Open _
        "Select Tasks, Resources, CInt(Duration) from tblProjects", _
        CurrentProject.Connection, adOpenKeyset

    Set appExcel = New Excel.Application
    Set wbkNew = appExcel.Workbooks.Add
    Set wksNew = wbkNew.Worksheets.Add
    appExcel.Visible = True

    With wksNew
        '-- Create the Column Headings
        .Cells(1, 1).Value = "Task"
        .Cells(1, 2).Value = "Resource"
        .Cells(1, 3).Value = "Hours"
    End With
```


LISTING 13.3 Continued

```

rstProjects.MoveLast
rstProjects.MoveFirst

Set rngCurr = wksNew.Range(wksNew.Cells(2, 1), _
    wksNew.Cells(2 + rstProjects.RecordCount, 3))

rngCurr.CopyFromRecordset rstProjects

'-- Create the calculation that sums up the Duration Column
wksNew.Cells(2 + rstProjects.RecordCount, 3).Value = _
    "=SUM(C2:C" & LTrim(Str(rstProjects.RecordCount) + 1) & ")"

wksNew.Columns("A:C").AutoFit

rstProjects.Close
Set rstProjects = Nothing

Exit Function

Error_OLEAccessToExcel:
Beep
MsgBox "The Following OLE Error has occurred:" & vbCrLf & _
    Err.Description, vbCritical, "OLE Error!"
Set appExcel = Nothing
Exit Function

End Function

```

In this code, the Excel application object has a `Visible` property. For the Excel application to be seen, this property is explicitly set by this line of code:

```
appExcel.Visible = True
```

The most exciting part of the code is how you can combine an element of Access with an element of Excel. For example, in the following lines of code, values from an Access table are being assigned to an Excel spreadsheet, using a method off the Excel Range object that is designed especially to use with recordsets:

```
Set rngCurr = wksNew.Range(wksNew.Cells(2, 1), _
    wksNew.Cells(2 + rstProjects.RecordCount, 3))
```

The last bit to look at is the code that assigns a formula to a spreadsheet cell and then best fits the columns by using the `AutoFit` method:

```
'-- Create the calculation that sums up the Duration Column
wksNew.Cells(2 + rstProjects.RecordCount, 3).Value = _
    "=SUM(C2:C" & LTrim(Str(rstProjects.RecordCount) + 1) & ")"

wksNew.Columns("A:C").AutoFit
```

You can call the `AccessToExcelAutomation()` function from the immediate window or by clicking the Excel Demo button on the `frmAutomationDemoCalls` form (refer to Figure 13.4). Figure 13.9 shows the final spreadsheet.

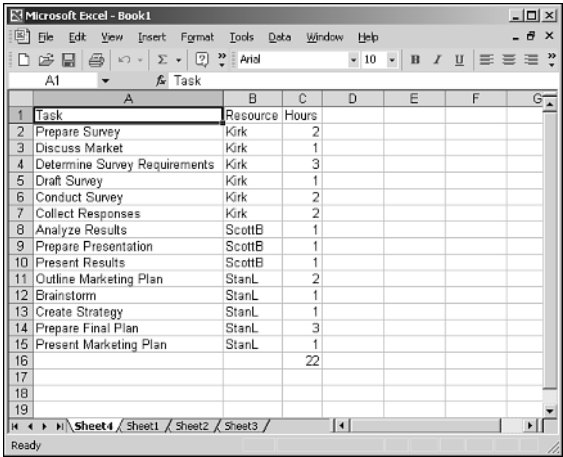


FIGURE 13.9
Not only can you populate Excel spreadsheet cells programmatically, but you also can create formulas.

Driving Microsoft Project from Access

Driving Project from Access with VBA is as easy as driving the other applications; you just have to know how the Project object model works. The next example creates a project from an Access table, which includes creating the task relations. Here are the steps performed:

1. The routine creates a reference to the current database.
2. It opens the `tblProjects` table.
3. It creates a reference to a Project instance.
4. The routine turns off alerts and opens a new project.
5. The routine writes the data from the Access record to Project.
6. It sets the project to be visible within Microsoft Project.

You can fire off this demo from the frmAutomationDemoCalls form by clicking the Project Demo button. Listing 13.4 shows the code, which is found in the modAutomationDemos module.

LISTING 13.4 Chap13.mdb: Creating a Project from an Access Table

```
Function AccessToProjectAutomation()
    Dim rstTasks As New ADODB.Recordset
    Dim intCurrTask As Integer

    '-- You must have project 9.0 installed; otherwise, an error occurs.
    On Error GoTo Error_OLEAccessToProject

    rstTasks.Open "tblProjects", CurrentProject.Connection

    '-- Create an instance of Project
    Set appProject = CreateObject("MSProject.Application.4_1")
    ' New MSProject.Application

    With appProject
        .DisplayAlerts = False
        .FileNew
    End With

    intCurrTask = 0    '-- Initialize the counter for the Unique Task ID

    Do Until rstTasks.EOF

        intCurrTask = intCurrTask + 1    '-- Increment the counter for the Task ID
        '-- Use the SetTaskField method to write data from Access record to Project
        appProject.SetTaskField Field:="Name", _
            Value:=rstTasks!Tasks, TaskID:=intCurrTask
        appProject.SetTaskField Field:="Start", _
            Value:=rstTasks!Start, TaskID:=intCurrTask
        appProject.SetTaskField Field:="Duration", _
            Value:=rstTasks!Duration, TaskID:=intCurrTask
        appProject.SetTaskField Field:="Predecessors", _
            Value:=IIf(IsNull(rstTasks!Predecessors), "", _
                rstTasks!Predecessors), TaskID:=intCurrTask
        appProject.SetTaskField Field:="Resource Names", _
            Value:=rstTasks!Resources, TaskID:=intCurrTask
        rstTasks.MoveNext
    Loop
```

LISTING 13.4 Continued

```
With appProject '-- Make project visible
    .Visible = True
    .AppMaximize
End With

rstTasks.Close
Set rstTasks = Nothing

Exit Function

Error_OLEAccessToProject:

    Beep
    MsgBox "The Following OLE Error has occurred:" & vbCrLf & _
        Err.Description, vbCritical, "OLE Error!"
    Exit Function

End Function
```

An interesting code segment assigns values from Access into Project. One code line that does this is

```
appProject.SetTaskField Field:="Name", _
    Value:=rstTasks!Tasks, TaskID:=intCurrTask
```

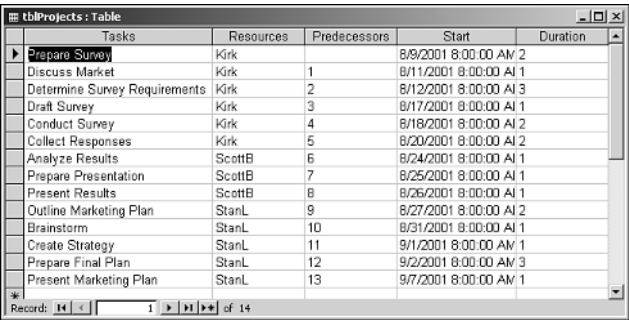
By using () the With statement, the Microsoft Project application is made visible by assigning the Visible property of the application object. Then the application is maximized:

```
With appProject
    .Visible = True
    .AppMaximize
End With
```

Figure 13.10 shows the original table used to create the project, and Figure 13.11 shows the project as seen in Microsoft Project.

Driving Outlook from Access

Driving Outlook from Access is actually easier, in my opinion, than driving from some of the other Office products. Even though Outlook uses VBScript for its language, from Access it codes just like any other Office app but with a simpler object model.



Tasks	Resources	Predecessors	Start	Duration
Prepare Survey	Kirk		8/9/2001 8:00:00 AM	2
Discuss Market	Kirk	1	8/11/2001 8:00:00 AM	1
Determine Survey Requirements	Kirk	2	8/12/2001 8:00:00 AM	3
Draft Survey	Kirk	3	8/17/2001 8:00:00 AM	1
Conduct Survey	Kirk	4	8/18/2001 8:00:00 AM	2
Collect Responses	Kirk	5	8/20/2001 8:00:00 AM	2
Analyze Results	ScottB	6	8/24/2001 8:00:00 AM	1
Prepare Presentation	ScottB	7	8/25/2001 8:00:00 AM	1
Present Results	ScottB	8	8/26/2001 8:00:00 AM	1
Outline Marketing Plan	StanL	9	8/27/2001 8:00:00 AM	2
Brainstorm	StanL	10	8/31/2001 8:00:00 AM	1
Create Strategy	StanL	11	9/1/2001 8:00:00 AM	1
Prepare Final Plan	StanL	12	9/2/2001 8:00:00 AM	3
Present Marketing Plan	StanL	13	9/7/2001 8:00:00 AM	1

FIGURE 13.10

The *tblProjects* table contains all the fields necessary to create a small project.

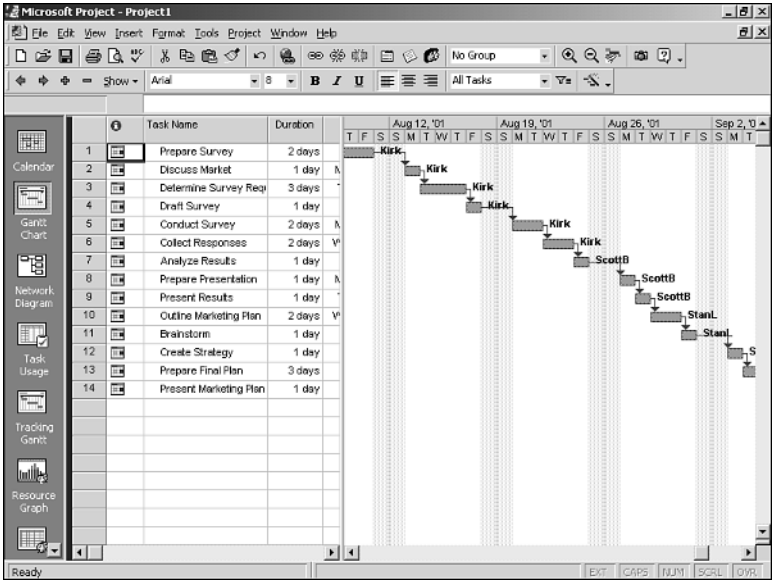


FIGURE 13.11

This project was created by using Access and Automation.

Most objects you will create will be different types of one object, called an *item*. Here are the possible items, with their Outlook.olItemType enum equivalents used with the CreateItem method:

<i>Item</i>	<i>Enum</i>
Mail	olMailItem
Appointment	olAppointmentItem
Contact	olContactItem
Task	olTaskItem
Journal	olJournalItem
Note	olNoteItem
Post	olPostItem

When you specify the type of item you will use, the properties particular to that type can then be accessed. The following sections look at four of the six types of items:

- Mail item
- Task item
- Contact item
- Appointment item (calendar)

All examples here are called from the frmAutomationDemoCalls form on their respective command buttons.

Creating a Outlook Mail Item

Let's start with the basics by creating a simple mail item. This procedure is useful when you're designing any number of applications. A good example of mail in an Access application is an e-mail to yourself about errors that occur for other users at different times during the application. Another example is an e-mail to the sales manager when an invoice is prepared for amounts of \$1 million or more. Figure 13.12 shows a mail message created from Access.

Listing 13.5 shows the code for this example, in the ap_CreateOLMailItem routine, which you can find in modOutlookRoutines. This module is located on the book's Web page at www.sampublishing.com in the Chap13.mdb database.

LISTING 13.5 Chap13.mdb: Creating an Outlook Mail Item from Access

```
Sub ap_CreateOLMailItem(strRecipient As String, strSubject As String)
    Dim objMailItem As Outlook.MailItem

    Set olkApp = New Outlook.Application
    Set olkNameSpace = olkApp.GetNamespace("MAPI")
    Set objMailItem = olkApp.CreateItem(olMailItem)

    With objMailItem
        .To = strRecipient
        .Recipients.ResolveAll
```

PART III**LISTING 13.5** Continued

```

.Subject = "This is the subject line"
.Body = "Here is the body"
.Display
End With

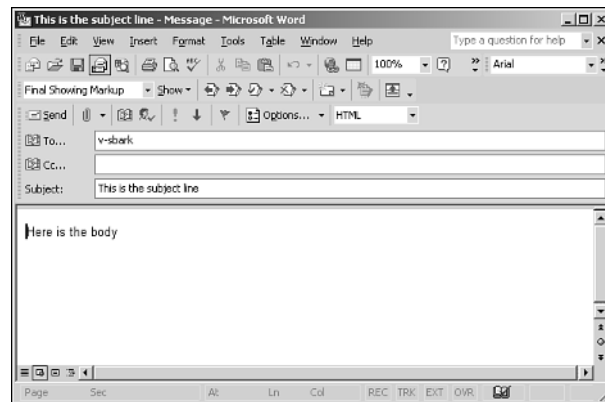
```

```

Set objMailItem = Nothing
Set olkNameSpace = Nothing
Set olkApp = Nothing

```

```
End Sub
```

**FIGURE 13.12**

This mail message was created programmatically from Access.

Notice the new object declared with the line

```
Dim objMailItem As Outlook.MailItem
```

This is part of the Outlook object model mentioned earlier and is the object you will use the most.

The next line sets a reference to Outlook.Application. The olkApp variable and NameSpace object variable were declared in the modOutlookRoutines module, shown here with the constants already mentioned:

```
Option Compare Database
Option Explicit
```

```
Public olkApp As Outlook.Application
Public olkNameSpace As Outlook.NameSpace
```

The `Namespace` object is used to reference *MAPI (Mail Application Programming Interface)* formatted data, which Outlook uses. Notice that I use the `GetNamespace` method in Listing 13.5 to point to MAPI, but then reference directly to the application for any objects or methods we might use.

After the `Namespace` object is referenced, I then use the `CreateItem` method to create the object I want—in this case, the mail item. Next, the necessary properties are set, and then the `Display` method is evoked to show the mail item.

Creating an Outlook Task Item from Access

Creating an Outlook task item is about the same as a mail item. The only real difference is the properties that will be set for the item. Figure 13.13 shows the finished product; Listing 13.6 shows the code from `modOutlookRoutines`.

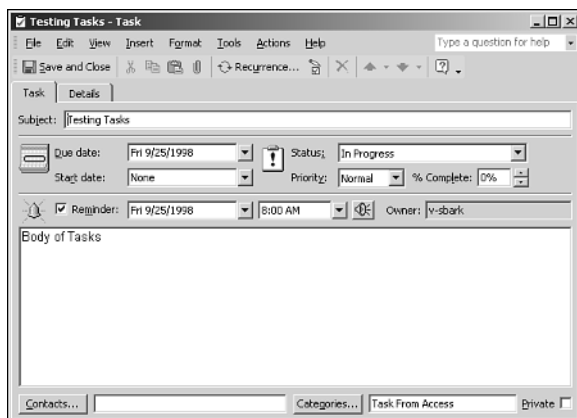


FIGURE 13.13

Make sure that Outlook reflects tasks created by your application.

LISTING 13.6 Chap13.mdb: Creating Outlook Task Items from Access

```
Sub ap_CreateOLTask(strSubject As String, strBody As String, _
    strDueDate As String, strOwner As String)

    Dim objTaskItem As TaskItem

    Set olkApp = New Outlook.Application
    Set olkNamespace = olkApp.GetNamespace("MAPI")
    Set objTaskItem = olkApp.CreateItem(olTaskItem)

    With objTaskItem
        .Subject = strSubject
        .DueDate = strDueDate
```


LISTING 13.6 Continued

```

.Status = olTaskInProgress
.ReminderSet = True
.ReminderTime = CDate(strDueDate) & " " & CDate(#8:00:00 AM#)
.Owner = strOwner
.Categories = "Task From Access"
.Body = strBody
.Display
End With

Set objTaskItem = Nothing
Set olkNameSpace = Nothing
Set olkApp = Nothing

```

End Sub

As mentioned earlier, the only differences between this and the preceding example are the properties set for the item. The next example gets a little trickier in that you're now going to create Outlook contacts.

Putting Contacts into Outlook from Access

When you create an Outlook contact, all you're doing is specifying a different item type and different properties. This example performs a bit more in that it takes all the customer records in the table `tblCustomers` (see Figure 13.14) and loads them into Outlook.



CustomerID	LastName	FirstName	Address	City	State	ZipCode
1	Baker	Steven	455 Anystreet	Woodville	WA	98933
3	Anders	Maria	Obere Str. 57	Woodinville	WA	98072
4	Trujillo	Ana	Avda. de la Con	Woodinville	WA	98072
5	Moreno	Antonio	Mataderos 231	Woodinville	WA	98072
6	Hardy	Thomas	120 Hanover Sq	Woodinville	WA	98072
7	Berglund	Christina	Berguvsvägen 6	Woodinville	WA	98072
8	Moos	Hanna	Forsterstr. 57	Woodinville	WA	98072
9	Citeaux	Frédérique	24, place Kléber	Woodinville	WA	98072
10	Sommer	Martin	C/ Araquil, 67	Woodinville	WA	98072
11	Lebihan	Laurence	12, rue des Bou	Woodinville	WA	98072
12	Lincoln	Elizabeth	23 Tsvassen E	Woodinville	WA	98072
13	Ashworth	Victoria	Fauntleroy Cir	Woodinville	WA	98072
14	Simpson	Patricio	Cerro 333	Woodinville	WA	98072
15	Chang	Francisco	Sierras de Gran	Woodinville	WA	98072
16	Wang	Yang	Hauptstr. 29	Woodinville	WA	98072
17	Afonso	Pedro	Av. dos Lusíada	Woodinville	WA	98072
18	Brown	Elizabeth	Berkeley Garde	Woodinville	WA	98072
19	Ottlieb	Sven	Walserweg 21	Woodinville	WA	98072
20	Labruno	Janine	67, rue des Cinc	Woodinville	WA	98072
21	Devon	Ann	35 King George	Woodinville	WA	98072
22	Mendel	Roland	Kirchgasse 6	Woodinville	WA	98072

FIGURE 13.14

The original Access table will soon reside in Outlook Contacts.

This routine not only puts the customers in Outlook Contacts, but also displays a nice status bar while doing so, using the SysCmd function in Access.

Before seeing the contacts in Outlook itself, let's look at the code in Listing 13.7.

LISTING 13.7 Chap13.mdb: Copying Access Customers into Outlook Contacts

```
Function ap_CreateOLContacts()
    Dim objContactItem As ContactItem
    Dim rstContacts As New ADODB.Recordset
    Dim intCurrRec As Integer, intRecCount As Integer

    Application.Echo True, _
        "Initializing to create Outlook contacts. Please wait..."

    rstContacts.Open "tblCustomers", CurrentProject.Connection, adOpenKeyset

    '-- Get the record count for the progress meter
    rstContacts.MoveLast
    intRecCount = rstContacts.RecordCount
    rstContacts.MoveFirst

    SysCmd acSysCmdInitMeter, "Creating Outlook contacts...", intRecCount
    intCurrRec = 1

    Set olkApp = CreateObject("Outlook.Application")
    Set olkNameSpace = olkApp.GetNamespace("MAPI")

    '-- Create an Outlook contact entry for each Calypso contact record
    Do Until rstContacts.EOF
        SysCmd acSysCmdUpdateMeter, intCurrRec
        Set objContactItem = olkApp.CreateItem(olContactItem)
        With objContactItem
            .FirstName = rstContacts!FirstName
            .LastName = rstContacts!LastName
            .BusinessAddress = rstContacts!Address
            .BusinessAddressCity = rstContacts!City
            .BusinessAddressState = rstContacts!State
            .BusinessAddressPostalCode = rstContacts!ZipCode
            .BusinessTelephoneNumber = rstContacts!PhoneNo
            '-- This helps to know this Outlook contact came from Access
            .Categories = "Access Contact"
            .Save
        End With
        rstContacts.MoveNext
        intCurrRec = intCurrRec + 1
    Loop
```

LISTING 13.7 Continued

```
rstContacts.Close

Set objContactItem = Nothing
Set olkNameSpace = Nothing
Set olkApp = Nothing
Set rstContacts = Nothing

SysCmd acSysCmdClearStatus
```

End Function

This routine has a few differences from the previous Outlook routines presented. Rather than just add a single item, it goes through a recordset and adds quite a few items. Another difference is the use of the SysCmd function, mentioned not long ago. This command for progress bars is summed up in the three lines of code used in this routine:

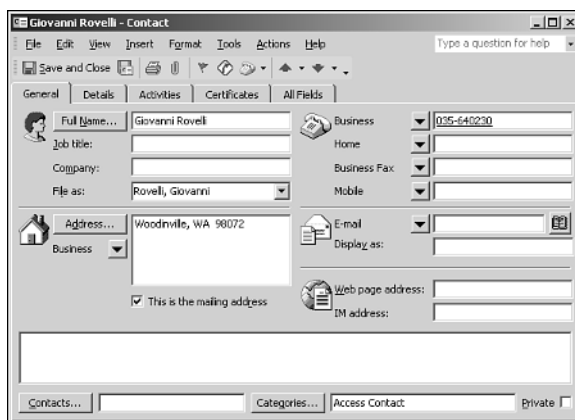
- The first line of code initializes the progress bar:
`SysCmd acSysCmdInitMeter, "Creating Outlook contacts...", intRecCount`
- The next code line updates the progress bar:
`SysCmd acSysCmdUpdateMeter, intCurrRec`
- The final line removes the progress meter when the process is complete:
`SysCmd acSysCmdClearStatus`

To see more of how to create progress bars, check out Chapter 14, “Programming for Power with ActiveX Controls,” which examines SysCmd and the ActiveX Progress Bar.

The last two differences between the current code and the previous Outlook routines are the most important:

- Rather than use the Display property to see the items, you use the Save method to store the item in the contacts folder.
- You set the Categories property on this code line:
`.Categories = "Access Contact"`

Use this property to specify what area this contact is used for. Besides the built-in categories that Outlook supplies, you can add your own, as shown in the last half of this example, where you see how to delete the Access contacts from Outlook’s Contacts folder. For now, see in Figure 13.15 how the contact information looks in Outlook coming from Access.

**FIGURE 13.15**

Notice that Categories at the bottom of this window is set to Access Contact.

Deleting Contacts in Outlook from Access

Sometimes you will need to remove contacts from Outlook, particularly if you refresh your contacts periodically. The code for performing this action varies from other tasks done in this chapter with the Outlook object model. To delete or “remove” items from Outlook, you need to reference the folder in which the item exists—in this case, the Contacts folder. Listing 13.8 shows this.

LISTING 13.8 Chap13.mdb: Removing Outlook Contacts from Access

```
Sub ap_ClearOLContacts()
    Dim objOLFolder As Outlook.MAPIFolder
    Dim olContactItem As ContactItem

    Application.Echo True, "Deleting Access Contacts in Outlook..."

    Set olkApp = New Outlook.Application
    Set olkNameSpace = olkApp.GetNamespace("MAPI")

    Dim intCurrContact As Integer

    Set objOLFolder = olkNameSpace.GetDefaultFolder(olFolderContacts)

    '-- Delete starting from the last of the list
    For intCurrContact = objOLFolder.Items.Count To 1 Step -1
        '-- Delete the entry if it came from Access
```

LISTING 13.8 Continued

```

    If objOLFolder.Items(intCurrContact).Categories = "Access Contact" Then
        objOLFolder.Items.Remove (intCurrContact)
    End If
Next

Application.Echo True

End Sub

```

Notice that the code uses the `Categories` property to note which items to remove. Also notice that you use the `GetDefaultFolder` method from the `NameSpace` object with the intrinsic constant `olFolderContacts` to open the desired folder.

Creating Outlook Calendar Entries from Access

In this example, you again specify a folder. Before doing so, however, look at an event procedure that supplies the dates that need to be added to the Outlook calendar. Listing 13.9 shows the code behind the `OnClick` event of the Outlook Calendar Demo button on the `frmAutomationDemoCalls` form.

LISTING 13.9 Chap13.mdb: Moving Through the Table That Supplies the Calendar Entries

```

Private Sub cmdOLCalendarItemsDemo_Click()
    Dim rstProjects As New ADODB.Recordset

    rstProjects.Open "tblProjects", CurrentProject.Connection

    With rstProjects
        Do Until .EOF
            ap_AddOLAppointment !Tasks, !Start, DateAdd("d", !Duration, !Start)
            .MoveNext
        Loop
        .Close
    End With

    Set rstProjects = Nothing

End Sub

```

For this example, again use the `tblProjects` table (see Figure 13.16).

Tasks	Resources	Predecessors	Start	Duration
Prepare Survey	Kirk		8/9/2001 8:00:00 AM	2
Discuss Market	Kirk	1	8/11/2001 8:00:00 AM	1
Determine Survey Requirements	Kirk	2	8/12/2001 8:00:00 AM	3
Draft Survey	Kirk	3	8/17/2001 8:00:00 AM	1
Conduct Survey	Kirk	4	8/18/2001 8:00:00 AM	2
Collect Responses	Kirk	5	8/20/2001 8:00:00 AM	2
Analyze Results	ScottB	6	8/24/2001 8:00:00 AM	1
Prepare Presentation	ScottB	7	8/25/2001 8:00:00 AM	1
Present Results	ScottB	8	8/26/2001 8:00:00 AM	1
Outline Marketing Plan	StanL	9	8/27/2001 8:00:00 AM	2
Brainstorm	StanL	10	8/31/2001 8:00:00 AM	1
Create Strategy	StanL	11	9/1/2001 8:00:00 AM	1
Prepare Final Plan	StanL	12	9/2/2001 8:00:00 AM	3
Present Marketing Plan	StanL	13	9/7/2001 8:00:00 AM	1

FIGURE 13.16

These tasks will soon be displayed in the Outlook calendar.

The real power behind this example is in the `ap_AddOLAppointment` function, located in the `modOutlookRoutines` module. Listing 13.10 shows this function.

LISTING 13.10 Chap13.mdb: Adding an Individual Calendar Entry

```
Function ap_AddOLAppointment(strSubject As String, varStart As Variant, _
    varEnd As Variant)

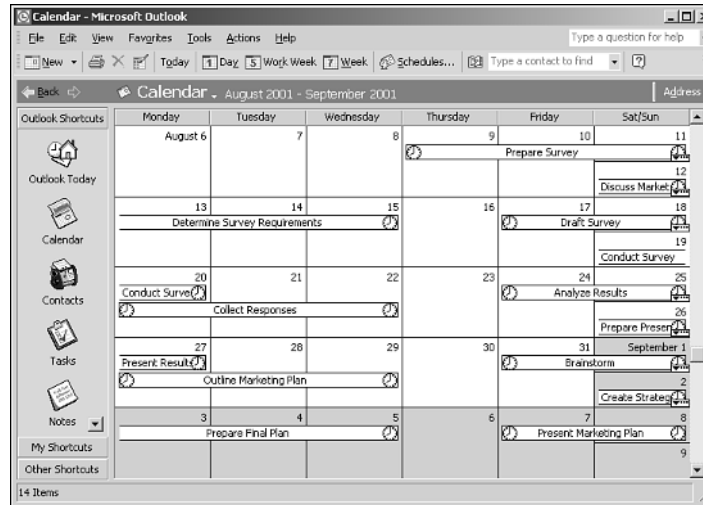
    Dim olkApp As New Outlook.Application
    Dim olkCalendar As Outlook.MAPIFolder
    Dim olkNameSpace As Outlook.NameSpace

    Set olkNameSpace = olkApp.GetNamespace("MAPI")
    Set olkCalendar = olkNameSpace.GetDefaultFolder(olFolderCalendar)

    With olkCalendar.Items.Add(olAppointmentItem)
        .AllDayEvent = True
        .Subject = strSubject
        .Start = CDate(varStart)
        .End = CDate(varEnd)
        .ReminderSet = False
        .Save
    End With

End Function
```

As in Listing 13.9, the code in Listing 13.10 sets a reference to the desired folder (`olCalendarFolder`), and then takes actions by using the `Items` collection in the folder. Again, it sets the properties necessary, and then uses the `Save` method to record the new item. Figure 13.17 shows the final calendar.

**FIGURE 13.17**

After appointments are moved to Outlook, they can easily be printed.

Driving Access from Another Application with Automation

Going from other Office products to Access is as easy as going from Access to others. It's just a matter of knowing which objects and methods to use with VBA. When working with Access objects such as reports and forms, you need to reference the Microsoft Access 10.0 Objects Library. When you work with tables and queries, depending on whether you use DAO or ADO, you must reference either the Microsoft DAO 3.6 Object Library for DAO or one or more of the three libraries for ADO. For a list of the ADO libraries, see Chapter 5, "Introducing ActiveX Data Objects."

For this example, you see how to have Project create an Access table containing key elements of a project. The `AutomateToAccess` routine, in the `Access.mpp` Microsoft Project project, is on this book's Web page at www.sampublishing.com. This routine performs the following steps:

1. It opens the database (for this example, `Chap13.mdb`) in which you want to create the table. This table, `tblProjects`, will hold information about a project.
2. `AutomateToAccess` calls the `CreateProjectTable` routine, which creates the `Projects` table; then `AutomateToAccess` appends the `Projects` table to the database's `TableDefs` collection.

3. The AutomateToAccess routine opens the Projects table in Access and cycles through each task, assigning the Project-specific fields.
4. The routine closes the recordsets.

Listing 13.11 shows the actual code that performs these tasks.

LISTING 13.11 Access.mpp: Creating an Access Table While in Microsoft Project

```
Sub AutomateToAccess()
    '-- Be sure to set your references
    Dim dbProjDemo As Database
    Dim dynProjects As Recordset
    Dim tskTasks As Task
    Dim ResResources As Resource

    On Error GoTo Error_AutomateToAccess

    '-- Open the sample mdb
    Set dbProjDemo = OpenDatabase(ActiveProject.Path & "\Chap13.mdb")

    On Error Resume Next
    dbProjDemo.TableDefs.Delete "tblProjects"
    On Error GoTo Error_AutomateToAccess

    CreateProjectTable dbProjDemo

    '-- Open the Projects table
    Set dynProjects = dbProjDemo.OpenRecordset("tblProjects")

    '-- Write resource and task info to the appropriate fields
    '-- Note that when dealing with Access object the bang is supported.
    For Each tskTasks In ActiveProject.Tasks
        With dynProjects
            .AddNew
            !Tasks = tskTasks.Name
            !Predecessors = tskTasks.Predecessors
            !Start = tskTasks.Start
            !Duration = tskTasks.Duration / 480 'proj calcs in minutes
            !Resources = tskTasks.ResourceNames
            .Update dbUpdateRegular, False
        End With
    Next tskTasks

    dynProjects.Close
    dbProjDemo.Close
    MsgBox "The table 'tblProjects' has been created " & _
        "in the database 'Chap13.mdb'."
Exit Sub
```


LISTING 13.11 Continued

```
Error_AutomateToAccess:
    MsgBox Err.Description
    Exit Sub

End Sub
```

NOTE

The ! (bang) character can be used with the Access objects. Not all Office applications use the bang character for referencing objects.

The following code line uses the Access `OpenDatabase` method combined with the `ActiveProject` object's `Path` property in `Project` to open an Access database from within `Project`:

```
Set dbProjDemo = OpenDatabase(ActiveProject.Path & "\Chap13.mdb")
```

To run this code in the project itself, open `Access.mpp` in Microsoft Project. You see the message shown in Figure 13.18.

To examine the code for the `AutomateToAccess` routine, choose `Macros` from the `Tools` menu. You can then select the `AutomateToAccess` macro and click `Edit`.

One last thing to look at before moving on is the `CreateProjectTable` routine. Just by looking at Listing 13.12, you can't tell whether it was written in Access or Project (for the record, it was written in Project). That's one of the beauties of working with VBA.

LISTING 13.12 Access.mpp: Creating the Structure of an Access Table

```
Sub CreateProjectTable(dbProjDemo As Database)
    Dim tdfProjects As TableDef
    Dim fldNewField As Field

    '-- Create a table
    Set tdfProjects = dbProjDemo.CreateTableDef("tblProjects")

    '-- Create the tasks field
    Set fldNewField = tdfProjects.CreateField("Tasks", dbText, 50)
    tdfProjects.Fields.Append fldNewField

    '-- Create the resources field
    Set fldNewField = tdfProjects.CreateField("Resources", dbText, 30)
    fldNewField.AllowZeroLength = True
    tdfProjects.Fields.Append fldNewField
```

LISTING 13.12 Continued

```
'-- Create the predecessors field
Set fldNewField = tdfProjects.CreateField("Predecessors", dbText, 30)
fldNewField.AllowZeroLength = True
tdfProjects.Fields.Append fldNewField

'-- Create the start field
Set fldNewField = tdfProjects.CreateField("Start", dbText, 30)
tdfProjects.Fields.Append fldNewField

'-- Create the Duration field
Set fldNewField = tdfProjects.CreateField("Duration", dbText, 30)
fldNewField.AllowZeroLength = True
tdfProjects.Fields.Append fldNewField

'-- Append the new table
dbProjDemo.Tabledefs.Append tdfProjects

End Sub
```

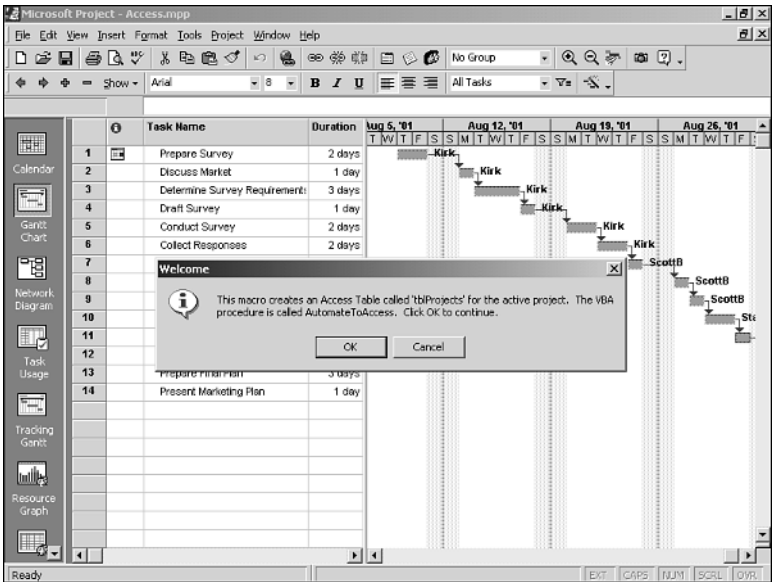


FIGURE 13.18
The VBA code in Access.mpp creates a table from a project, storing tasks and resources.

The Access.mpp routines create the tblProjects table shown earlier in Figure 13.10.

Summary

With Automation, you're no longer limited to working in one type of programming environment to present a solution for a client. By using Automation, you can give users seamless applications that take advantage of the data maintenance of a database application, the number-crunching capability of a spreadsheet, and the text formatting of a word processor.

- Chapter 2, "Coding in Access 2002 with VBA," gives the basics of coding VBA in Access, including using methods and objects.
- Chapter 4, "Working with Access Collections and Objects," explains the various collections that you can use with an application's objects and Data Access Objects, along with the methods used to manipulate them.
- Chapter 5, "Introducing ActiveX Data Objects," describes the object model for ADO and how to take advantage of it.

Programming for Power with ActiveX Controls

CHAPTER

14

IN THIS CHAPTER

- Understanding the ActiveX Common Controls 424
- Using the TabStrip Control 425
- Taking a Closer Look at the ImageList Control 428
- Emulating the Windows Explorer with the ListView Control 434
- Displaying a Task's Progress with the ProgressBar Control 439
- Sizing Text Boxes at Runtime with the Slider Control 443
- Telling It Like It Is with the Rich Textbox Control 445
- Creating Status Bars for Individual Forms with the StatusBar Control 450
- Docking Toolbars on Forms Using the ToolBar Control 453
- Viewing Data File Manager Style with the TreeView Control 456

You can use ActiveX controls, generally created with Visual Basic or C++, in your Access application. They are created to be used within various development environments such as Visual Basic, Access, and even HTML.

Because ActiveX controls are a common part of developing applications in Windows, it's understood that most Access developers have had a chance to work with ActiveX controls, or at least the basics of adding the control to a form and using it without coding. If this isn't the case for you, Appendix B, "Getting Started with ActiveX Controls," introduces you to working with ActiveX controls: how to register controls manually and programmatically, how to place them on forms, how to manipulate properties at design time and runtime, and so on. Appendix B also discusses the Calendar control, included in the Access retail box. (You can find this appendix on this book's Web page at www.sampublishing.com.)

Understanding the ActiveX Common Controls

Windows has a number of controls that you can access through ActiveX. Nine ActiveX controls are available with Comctl32.ocx, which is automatically registered when you install the Microsoft Office Developer (MOD). The MOD includes a number of other ActiveX controls. Table 14.1 lists the ones described in this chapter.

TABLE 14.1 ActiveX Common Controls Included in the MOD

<i>OCX Name</i>	<i>Description</i>
ImageList	Holds bitmaps and icons that the other controls can use
ListView	Displays images in the same four views available in Explorer
Progress	Displays the progress of a task anywhere on any form
Rich Textbox	Allows you to use different fonts, colors, sizes, italics, and so forth for each text box character, if desired
Slider	Displays a vertical or horizontal slider control
StatusBar	Can be used on the main Access screen, as well as on each individual form
TabStrip	Allows you to arrange tabs in multiple rows and to display images on tabs
ToolBar	Gives you more control over toolbars
TreeView	Allows you to view data hierarchically

Using the TabStrip Control

The native Access Tab control performs the same tasks as the ActiveX TabStrip control. However, because this chapter is on ActiveX controls, we should discuss the TabStrip control.

With the TabStrip control, you can create a tabbed form with multiple rows. All you have to do is set MultiRow to Yes. Figure 14.1 shows an example of multiple tab rows created with the TabStrip control.

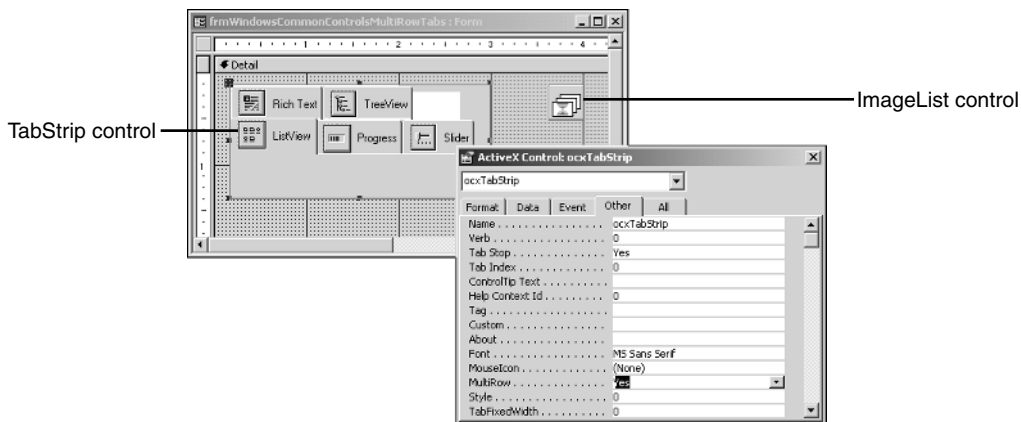


FIGURE 14.1

The TabStrip control can show multiple tab rows with the MultiRow property.

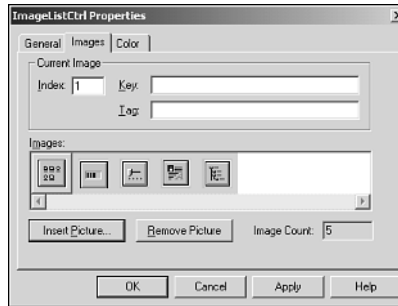
Using an ImageList Control with the TabStrip Control

The TabStrip control also lets you display images on the tabs. Figure 14.1 includes an ImageList control that specifies the list of images to use.

To specify a list of images, insert an ImageList control onto a form in which you have a TabStrip control. Then open the ImageList control-specific property sheet. On the Images page, click Insert Picture and add a list of images. Figure 14.2 shows a list of selected images.

Again, after you create the image list, do the following to place the images on each tab:

1. Open the TabStrip control-specific property sheet.
2. Set the TabStrip control's ImageList property to the name of the ImageList control.
3. On the property sheet's Tabs page, set each tab's Image property to the corresponding image index value. The ListView image is 1, the Slider image is 2, the Progress image is 3, and so forth.

**FIGURE 14.2**

These bitmaps make up an image list used with the TabStrip control.

You can also use the ImageList control with the ListView, ToolBar, and TreeView controls. More information on using the ImageList control appears later in the section “Taking a Closer Look at the ImageList Control.”

Programming the Standard Access Tab Versus the ActiveX TabStrip Control

Using the Access Tab control and ActiveX TabStrip control in VBA is basically the same, except that different properties are used and that the routines are attached to different events. When using the Tab control, the default Value property is used; the TabStrip control, on the other hand, uses the SelectedItem.Index property.

Listing 14.1 shows the code used for the Access Tab control and attached to the Change event.

NOTE

The code in Listing 14.1 isn't in the sample database because the Access Tab control is usually used. I present the code here simply to show how to use the Tab control with VBA.

LISTING 14.1 Programming the Access Tab Control

```
Private Sub Tabs_Change()
    On Error GoTo Error_Tabs_Change

    Me.Painting = False
    ' The Tab value property is zero based
    Select Case Me!Tabs.Object.Value
```

LISTING 14.1 Continued

```

    Case 0
        Me!CustomerSub1.Visible = True
        Me!CustomerSub2.Visible = False
        Me!CustomerSub3.Visible = False
    Case 1
        Me!CustomerSub1.Visible = False
        Me!CustomerSub2.Visible = True
        Me!CustomerSub3.Visible = False
    Case 2
        Me!CustomerSub1.Visible = False
        Me!CustomerSub2.Visible = False
        Me!CustomerSub3.Visible = True
End Select
Me.Painting = True

Exit_Tabs_Change:
Exit Sub

Error_Tabs_Change:
MsgBox Err.Description
Resume Exit_Tabs_Change:

End Sub

```

Listing 14.2 shows the same code used with the TabStrip control. Notice that you attach it to the Click event. Also notice that instead of Value, you look at SelectedItem.Index.

LISTING 14.2 Programming the ActiveX TabStrip Control

```

Private Sub ocxTabStrip_Click()
    On Error GoTo Error_ocxTabStrip_Click

    Me.Painting = False
    ' The TabStrip Selected.Index property is zero-based
    Select Case Me!ocxTa3bStrip.SelectedItem.Index
        Case 0
            Me!CustomerSub1.Visible = True
            Me!CustomerSub2.Visible = False
            Me!CustomerSub3.Visible = False
        Case 1
            Me!CustomerSub1.Visible = False
            Me!CustomerSub2.Visible = True
            Me!CustomerSub3.Visible = False
    End Select

```


LISTING 14.2 Continued

```
Case 2
    Me!CustomerSub1.Visible = False
    Me!CustomerSub2.Visible = False
    Me!CustomerSub3.Visible = True
End Select
Me.Painting = True

Exit ocxTabStrip Click:
Exit Sub

Error ocxTabStrip Click:
MsgBox Error$
Resume Exit ocxTabStrip Click

End Sub
```

In the end, with Access, you will want to use the standard Access Tab control.

For more information on paging techniques with multiple subforms, see Chapter 9, “Creating Powerful Forms.”

Taking a Closer Look at the ImageList Control

The ImageList control, mentioned briefly earlier in the TabStrip control discussion, is also used with the ListView, ToolBar, and TreeView controls. These controls are examined individually and examples are given for use with the ImageList control, but some specific syntax for programming the ImageList control is called for at this time.

By itself, the ImageList control is pretty useless. However, when you use it with other controls, you can display bitmaps and icons, thereby giving your interface that professional Windows graphical look.

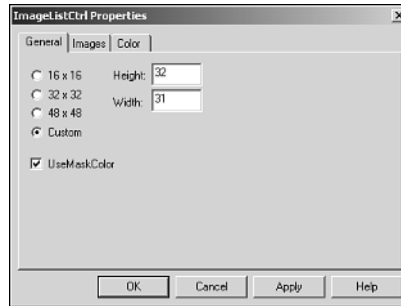
You can load the images used in the ImageList control at design time or during runtime.

Adding Images During Design Time

To place an ImageList control on a form and load it with images at design time, follow these steps:

1. Open the form on which you plan to use the ImageList control in Design mode.
2. From the Insert menu, choose Custom Controls.
3. In the Insert ActiveX Controls dialog, select the Microsoft ImageList control from the list of controls and click OK. You now have an ImageList control on the form.

4. Double-click the middle of the control to open the ImageList's property sheet. On the General page, you can choose the size of the images you're storing (see Figure 14.3). Leave the default, Custom.

**FIGURE 14.3**

You can choose various sizes to display from the ImageList control. Here, the width and height have been filled in.

NOTE

When determining the bitmaps/icons to use in the ImageList control, remember that they all have to be the same size. Otherwise, an irritating error message keeps appearing.

5. On the Images page, click the Insert Picture button. In the Select Picture dialog that opens, choose a graphic to go into the ImageList based on the graphic size you want to display, and then click OK.
6. Repeat step 5 for each graphic you want to include. Click OK in the Insert ActiveX Controls dialog when finished.

NOTE

ImageList controls store images in memory when a form is open. The more images you have, the more memory that's used, and the longer the form's load time.

The Current Image section of the Images page has three properties, two of which are filled in automatically for you:

- The Index property is the index value of the current image. This value corresponds with the Image property of the control that this ImageList is used with. This property is filled in automatically.

For example, on the `frmWindowsCommonControls` form, the `TabStrip` control has pictures on each tab. Open this form, double-click the `TabStrip`, and then select the `Tabs` tab. (You can find the `frmWindowsCommonControls` form in the `Chap14.mdb` database, on this book's Web page at www.sampublishing.com.) If you look at the `Image` property at the bottom of the property sheet, it's a 1 for the first choice, `ListView` (see Figure 14.4).

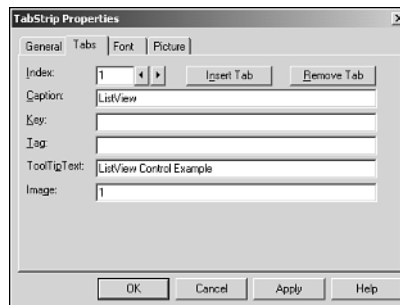


FIGURE 14.4

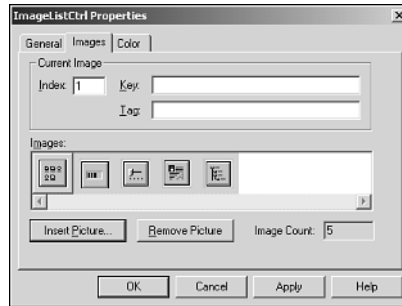
This TabStrip uses an ImageList with the current tab pointing to the first image.

Close the `TabStrip Properties` sheet and locate on the form the `ImageList` control named `ocxImageList`. (Select `ocxImageList` from the Object pick list on the Format toolbar. If it hasn't been moved, the `ocxImageList` control should be on the `RichText` tab on the `TabStrip` control.) Double-click `ocxImageList` to bring up the control-specific property sheet. Then on the `Images` page, you can see all the pictures used on this form's `TabStrip` control. Notice that `Index 1` points to the `ListView` picture, as in `Image 1` of the `TabStrip` control (see Figure 14.5).

- Use the `Key` property to tie the image to the other control. Rather than specify a number, give a text key value that can then be used in the other control. Notice that the `Key` property is used in the `TabStrip` and `ImageList` controls in Figures 14.4 and 14.5. For this example, it's left blank.

TIP

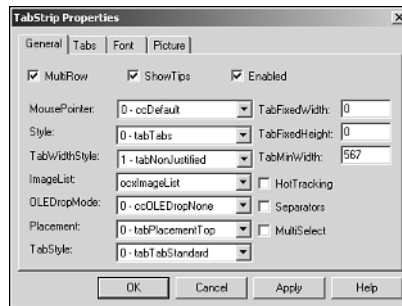
The `Key` property is useful when you're linking the images programmatically. As with using constants, they make the code easier to understand when maintaining it because you can give them meaningful names instead of numbers.

**FIGURE 14.5**

The *Index* property for the *ImageList* control corresponds with the *Image* property in other controls, including *TabStrip*, *ListView*, *TreeView*, and *ToolBar*.

- The *Image Count* property displays the current number of images in the *ImageList*. This number increments automatically as you add images.

When the *ImageList* is created, it's simply a matter of specifying that the list is to be used in the control. Looking again at the *ocxTabStrip* found on the *frmWindowsCommonControls* form, bring up the control-specific property sheet. You see the *ImageList* property that specifies the *ImageList* control to use with the *TabStrip* control. As shown again in Figure 14.6, *ocxImageList* is selected.

**FIGURE 14.6**

With the *ImageList* property, you connect an *ImageList* control to the *TabStrip* control.

Adding Images to the ImageList Control at Runtime

Sometimes, you might prefer to load images at runtime rather than at design time—for example, when the bitmaps need to be different based on the task.

As with objects in Access, the ImageList control has a collection made up of other objects. In this case, the ImageList's collection is made up of ListImages. Each ListImage has its own set of properties and certain methods that can be used.

To add a ListImage to the ListImages collection, use the Add method. Again, using this method with the ListImages collection is similar to the other Access object collections. (For more information on Access collections and manipulating them, see Chapter 4, "Working with Access Collections and Objects.")

To see how to add items to ImageList controls, look at Listing 14.3, which was taken from the OnLoad event on the frmWindowsCommonControlsListView form. (For the complete listing, check out Listing 14.4.)

LISTING 14.3 Chap14.mdb: Loading an ImageList with VBA

```
Dim strAppPath As String
'-- Use a dummy object variable that will be thrown away.
Dim objDummy As Object
'-- ListView Variable
Dim ocxListV As Object

strAppPath = CurrentProject.Path

'-- Create a reference to the control
Set ocxListV = Me!ocxListView

'-- Load the images into the image list controls
Set objDummy = Me!ocxImageList1.ListImages.Add(, , LoadPicture(strAppPath _
    & "\Graphics\bigvid.ico"))
Set objDummy = Me!ocxImageList2.ListImages.Add(, , LoadPicture(strAppPath _
    & "\Graphics\1tl1vid.bmp"))

'-- Store the image lists into the ListView control's various picture types
ocxListV.Icons = Me!ocxImageList1.Object
ocxListV.SmallIcons = Me!ocxImageList2.Object
```

Listing 14.3 uses object variables. ObjDummy doesn't store anything needed in this case, thus the name. A reference to the new ListImages item is created, but it's not needed in this example. The other object variable, objListV, creates a reference to the ListView object being manipulated.

These code lines add single images to two different ImageList controls—one a bitmap and the other an icon:

```
'-- Load the images into the image list controls
Set objDummy = Me!ocxImageList1.ListImages.Add( , LoadPicture(strAppPath _
& "\Graphics\bigvid.ico"))
Set objDummy = Me!ocxImageList2.ListImages.Add( , LoadPicture(strAppPath _
& "\Graphics\l1t1vid.bmp"))
```

You need to use two different ImageList controls. A single ImageList control can contain only images of the same size. The ListView control, however, might use different-sized controls, based on the view chosen. (For more information on the ListView control, see the next section.)

After the images are added to the ImageList controls, the code connects the two ImageLists to the properties in the ListView control:

```
ocxListV.LargeIcons = Me!ocxImageList1.Object
ocxListV.SmallIcons = Me!ocxImageList2.Object
```

The ListView can now use the images. Figure 14.7 shows the large bitmaps; Figure 14.8 shows the same form displaying a detail view.

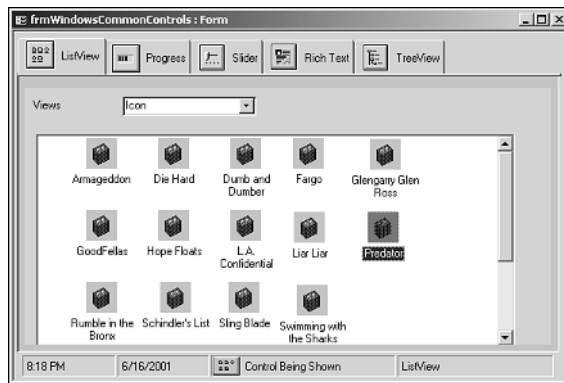
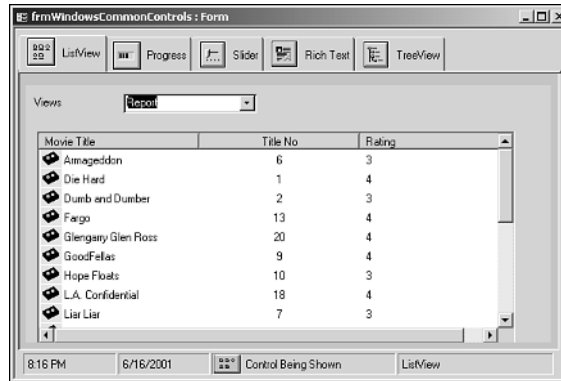


FIGURE 14.7

Here, the ListView control is using ocxImageList1 to display images in icon mode.

NOTE

The images used in the preceding example could easily have been assigned at design time. This example was strictly for demonstrative purposes only.

**FIGURE 14.8**

Here, the ListView control uses `ocxImageList2` to display images in report view.

You can use the ImageList control with other controls in myriad ways. Besides reading about them in the following sections, you can see the ImageList control used in the forms `frmWindowsCommonControls`, `frmWindowsCommonControlsListView`, `frmWindowsCommonControlsMultiRowTabs`, `frmWindowsCommonControlsToolBar`, and `frmWindowsCommonControlsTreeView`.

Emulating the Windows Explorer with the ListView Control

The ListView control allows you to create a dialog that looks similar to the views found in the Windows Explorer. With this control, you can add a view that's different from the everyday laundry list of items.

The following sections show how to fill the ListView control programmatically. But first, some explanation about the control and its properties is needed.

Looking at the Different Views of the ListView Control

The following views can be found in the ListView control:

- **Large Icons.** This view displays large icons with the label underneath (refer to Figure 14.7). Only the specified label is displayed.
- **Small Icons.** Items are lined up across the screen with small icons displayed beside the labels. Only the specified label is displayed.
- **List.** Items are listed down the screen with small icons displayed beside the labels. Only the specified label is displayed.

- **Report.** Items are listed down the screen with small icons displayed beside the labels (refer to Figure 14.8). Other item details are also displayed.

Although the names of the first three view choices match Explorer's view names, the Report view corresponds to Explorer's Detail view.

Seeing the Major Groupings of the ListView Control Properties

The ListView control has properties not only for the main control, but also for each possible column (see Figure 14.9). Table 14.2 describes each page on this property sheet.

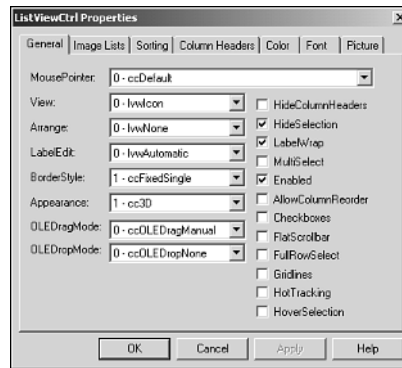


FIGURE 14.9

The General properties for the ListView control affect the control's overall appearance.

TABLE 14.2 The Pages of the ListView Control's Property Sheet

Page	Description
General	Contains properties that affect the ListView control's overall display.
Image Lists	Requires that you specify two ImageList controls: one for the normal icon size and one for the small icon size.
Sorting	Contains three properties. The first specifies whether you want to sort a column of the ListView control. The second allows you to set which column you want to use for sorting (0 – for the label text, > 0 for the other columns). The third specifies whether to sort ascending or descending.
Column Headers	Adds the individual columns. You can specify the text, alignment, and width, as well as other properties pertaining to columns.

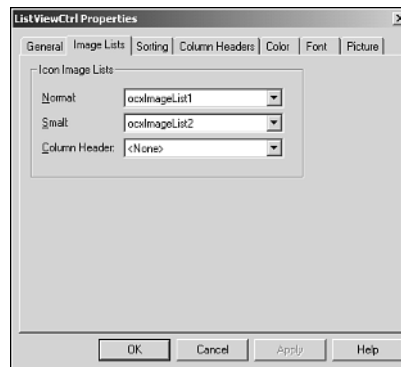
TABLE 14.2 Continued

<i>Page</i>	<i>Description</i>
Color	Lets you specify colors for the standard Windows settings.
Font	Allows you to select the fonts for the text displayed.
Picture	Lets you set a custom mouse cursor.

Setting Up a ListView Control Manually

To set up a ListView control manually, follow these steps:

1. Create two ImageLists as described earlier in the section “Adding Images During Design Time”: one for normal icons and one for small icons. The ImageLists have only one picture apiece.
2. Specify the two ImageList controls in the appropriate property on the Image Lists page, Normal and Small (see Figure 14.10).

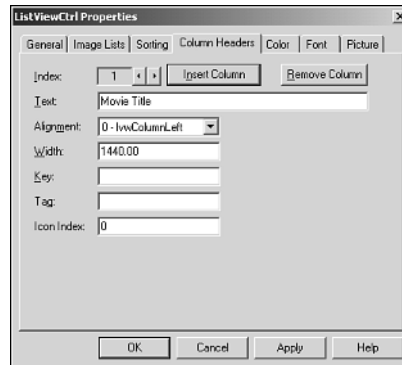
**FIGURE 14.10**

The Image Lists page lets you set up which ImageLists to use in the ListView control.

3. On the Column Headers page, create the columns you want to include in your view list. Figure 14.11 shows the filled-out page for the Movie Title column heading.

NOTE

The Width value for columns is set in *twips*, the unit of measure when assigning values to screen measurements from VBA. A twip is 1,440th of an inch.

**FIGURE 14.11**

You can set up column headings for the ListView control in Design view.

Although you can set up the ListView control's columns and images at design time, the actual data that's displayed must be loaded in code by using VBA. Unlike Access controls such as combo boxes and list boxes, the ListView control isn't bound.

Creating and Filling a ListView Control Using VBA

How a ListView control is created is easier to see through code instead of with steps through the property sheet. The following example shows how to load ImageList controls, create column headings for the ListView control, and then fill the control with data. Listing 14.4 shows the code that's run in the OnLoad event for the frmWindowsCommonControlsListView form.

LISTING 14.4 Chap14.mdb: Filling a ListView Control

```
Private Sub Form_Load()
    Dim strAppPath As String

    '-- Use a dummy object variable that will be thrown away.
    Dim objDummy As Object

    '-- ListView Variable
    Dim ocxListV As Object

    strAppPath = CurrentProject.Path

    '-- Create a reference to the control
    Set ocxListV = Me!ocxListView
```

LISTING 14.4 Continued

```
'-- Load the images into the image list controls
Set objDummy = Me!ocxImageList1.ListImages.Add(, , LoadPicture(strAppPath _
    & "\Graphics\bigvid.ico"))
Set objDummy = Me!ocxImageList2.ListImages.Add(, , LoadPicture(strAppPath _
    & "\Graphics\ltlvid.bmp"))

'-- Store the image lists into the ListView control's various picture types
ocxListV.Icons = Me!ocxImageList1.Object
ocxListV.SmallIcons = Me!ocxImageList2.Object

'-- Create the Headers
Set objDummy = ocxListV.ColumnHeaders.Add(, , "Movie Title", _
    ocxListV.Width / 3)
Set objDummy = ocxListV.ColumnHeaders.Add(, , "Title No", _
    ocxListV.Width / 3, 2)
Set objDummy = ocxListV.ColumnHeaders.Add(, , "Rating", _
    ocxListV.Width / 3)

'-- Load the data
Dim rstMovies As New ADODB.Recordset
Dim objListItem As Object

rstMovies.Open "Select * From tblMovieTitles Order By Title", _
    CurrentProject.Connection

Do Until rstMovies.EOF
    '-- Load the fields
    Set objListItem = ocxListV.ListItems.Add(, , rstMovies!Title, 1)
    objListItem.Icon = 1
    objListItem.SmallIcon = 1
    objListItem.SubItems(1) = CStr(rstMovies!TitleNo)
    objListItem.SubItems(2) = rstMovies!Rating
    rstMovies.MoveNext
Loop

rstMovies.Close
Set rstMovies = Nothing
Set objDummy = Nothing
Set ocxListV = Nothing
End Sub
```

The code in Listing 14.4 performs these steps:

1. It declares two object variables: `objDummy` and `ocxListV` (`objDummy` isn't actually used for anything in this code).
2. It sets `ocxListV` as a reference to the `ListView` control on the form.
3. The code uses the `ListImages` collection's `Add` method for two `ImageList` controls and loads the two different icon sizes. (This code section is covered earlier in Listing 14.3 and in the section "Adding Images to the `ImageList` Control at Runtime.")
4. It assigns the two `ListImage` controls to the `Icon` (referred to as `Normal` in the UI) and `SmallIcon` (`Small`) properties of the `ListView` control. (This is the same as step 2 in the preceding section.)
5. The code uses the `ColumnHeaders` collection's `Add` method to create the columns of the `ListView` control.
6. It opens the recordset that supplies the actual data.
7. It loads the data into the `ListItems` collection via the `Add` method. The `SubItems` collection off the `ListItems` object is loaded with the other information to be displayed—in this case, the `TitleNo` and `Rating` columns.

With this code, you can start displaying data with the `ListView` control. You can also create different views for flexibility.

Displaying a Task's Progress with the `ProgressBar` Control

The `ProgressBar` ActiveX control provides you with another way to inform users of the progress of processes that might take some time to perform. Using the `ProgressBar` control to show a task's progress is a great addition to a professional user interface.

Displaying the Access Progress Bar with `SysCmd()`

In the past, you could use the `SysCmd()` function to display a progress bar only on the Access status bar. Using `SysCmd()` to create a status bar is pretty straightforward. The code in Listing 14.5 is attached to the `OnClick` event of the `frmWindowsCommonControlsProgressBar` form's `cmdShowOldProgress` command button.

LISTING 14.5 Chap14.mdb: Displaying the Old Access Progress Bar

```
Private Sub cmdShowOldProgress_Click()  
    Dim lngCounter As Long  
    Dim varDummy As Variant  
    Dim intWait As Integer
```

LISTING 14.5 Continued

```
'-- Initialize progress bar
varDummy = SysCmd(acSysCmdInitMeter, "Old Progress", Me!txtMaximum)

For lngCounter = Me!txtMinimum To Me!txtMaximum
    '-- Because of fast machines
    For intWait = 1 to 5
        DoEvents
    Next
    '-- Increment progress bar
    varDummy = SysCmd(acSysCmdUpdateMeter, lngCounter)
Next

Beep
MsgBox "Progress bar complete"

'-- Clear progress bar
varDummy = SysCmd(acSysCmdClearStatus)

End Sub
```

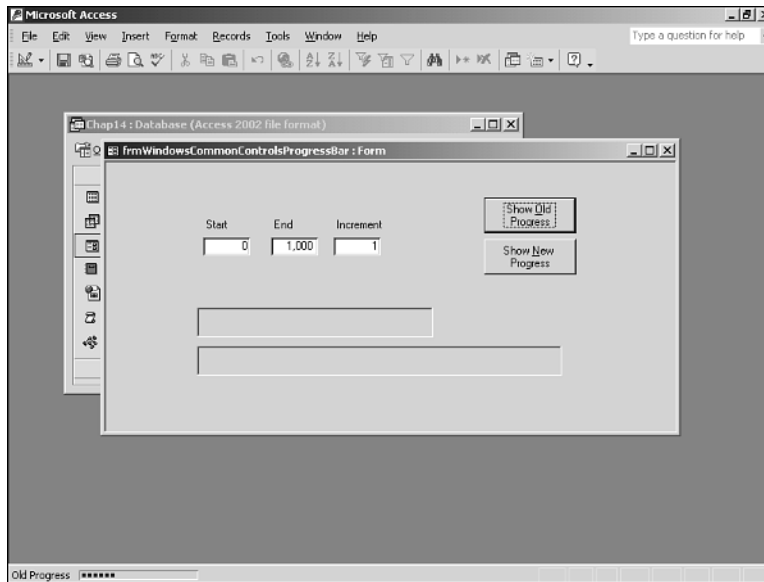
NOTE

The constants used in this routine—`acSysCmdInitMeter`, `acSysCmdUpdateMeter`, and `acSysCmdClearStatus`—are all Access intrinsic constants.

The following steps show how the `SysCmd()` function is used in Listing 14.5's routine, `cmdShowOldProgress_Click`:

1. Calling `SysCmd()` initializes the progress bar. `acSysCmdInitMeter` and the text to display is passed to `SysCmd()`.
2. Calling `SysCmd()`, passing `acSysCmdUpdateMeter`, and passing the counter value increments the progress bar.
3. Calling `SysCmd()` and passing `acSysCmdClearStatus` clears the progress bar.

Figure 14.12 shows the Access progress bar in action. The main issue with using `SysCmd()` is that the progress bar can be displayed only at the bottom of the Access application window. In some cases, you might want to display a progress bar on individual forms.

**FIGURE 14.12**

The Access progress bar using `SysCmd()` appears at the bottom of the screen.

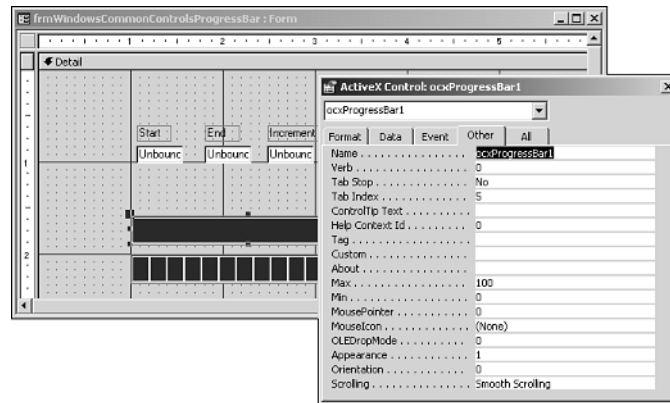
Using the ActiveX ProgressBar Control

With the ActiveX ProgressBar control, you can place a progress bar on any form you want. On `frmWindowsCommonControlsProgressBar`, the same form that contains the sample code for `SysCmd()`, are two ProgressBar controls. Figure 14.13 shows the two controls, with the property sheet for the `ocxProgressBar1` ProgressBar control.

Although you can preset the Min and Max properties and then use code to increment the control's Value property to increase the progress bar, it's more pragmatic to set the Min and Max properties in code at runtime. When you set these properties depends on whether the range for the progress bar will change.

NOTE

Notice in Figure 14.13 that `ocxProgressBar1`'s Scrolling property has been set to Smooth Scrolling. This causes the progress bar to display continuously rather than in the standard blocks. You can see this shortly in Figure 14.14.

**FIGURE 14.13**

The Progress control has few properties to deal with.

The code in Listing 14.6 is attached to the OnClick event of the cmdShowNewProgress command button.

LISTING 14.6 Chap14.mdb: Using the Progress ActiveX Control

```
Private Sub cmdShowNewProgress_Click()
    Dim intCounter As Integer

    '-- Initialize the Progress controls
    Me!ocxProgressBar1.Min = Me!txtMinimum
    Me!ocxProgressBar1.Max = Me!txtMaximum
    Me!ocxProgressBar2.Min = Me!txtMinimum
    Me!ocxProgressBar2.Max = Me!txtMaximum

    '-- Increment the Value properties for the Progress controls
    For intCounter = Me!txtMinimum To Me!txtMaximum Step Me!txtIncrement
        Me!ocxProgressBar1.Object.Value = intCounter
        Me!ocxProgressBar2.Object.Value = intCounter
    Next intCounter

    Beep
    MsgBox "Progress bar complete"

    '-- Clear the Progress controls
    Me!ocxProgressBar1.Object.Value = 0
    Me!ocxProgressBar2.Object.Value = 0

End Sub
```

The steps for setting up and updating the ProgressBar control are similar to those for Listing 14.5. The difference is that with the ProgressBar control, you can set both the Min and Max properties.

You can change the values in the txtMinimum, txtMaximum, and txtIncrement text boxes on the frmWindowsCommonControlsProgressBar form. Figure 14.14 shows the ProgressBar control in action, with the just-mentioned TextBox control values set to the defaults.

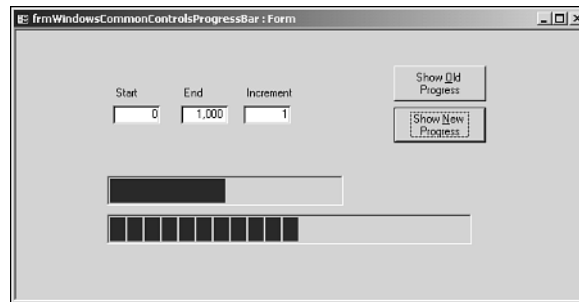


FIGURE 14.14

You can create different-sized and style ProgressBar controls on any form needed.

Sizing Text Boxes at Runtime with the Slider Control

The Slider control is very useful for tasks that need a visual method for incrementing or decrementing values. On the frmWindowsCommonControlsSlider form is an example of using a Slider control for controlling the Height and Top properties of two TextBox controls. Figure 14.15 shows the frmWindowsCommonControls form with frmWindowsCommonControlsSlider as the subform. These forms are in the Chap14.mdb database, which you can find on this book's Web site at www.sampublishing.com.

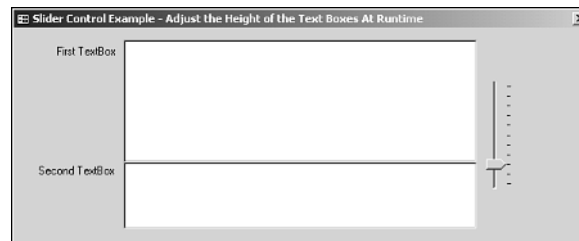


FIGURE 14.15

The Slider control is useful for controlling the size and location of controls.

Listing 14.7 shows the code for the Slider control, which is attached to the `OnClick` event for the `ocxSlider` control on the `frmWindowsCommonControlsSlider` form.

LISTING 14.7 Chap14.mdb: Adjusting the Slider Control

```
Private Sub ocxSlider_Click()
    '-- Height of the Text Boxes
    Dim intTxtHeight As Integer
    intTxtHeight = 1

    '-- Starting Top property value of the Second Text Box
    Dim sngTxt2Top As Single
    sngTxt2Top = 1.166

    '-- Turn off the painting for the form
    Me.Painting = False

    '-- Calculate the new dimensions offset
    Dim sngNewDimensions As Single
    sngNewDimensions = (Me!ocxSlider.Value - Me!ocxSlider.SelStart) _
        / Me!ocxSlider.Max

    '-- Set the new value of the Height Value of the Text Boxes
    Me!txtTextBox1.Height = (intTxtHeight + sngNewDimensions) * 1440
    Me!txtTextBox2.Height = (intTxtHeight - sngNewDimensions) * 1440
    Me!txtTextBox2.Top = (sngTxt2Top + sngNewDimensions) * 1440
    Me!lblTextBox2.Top = (sngTxt2Top + sngNewDimensions) * 1440

    '-- Turn on the painting for the form
    Me.Painting = True

End Sub
```

One thing needs to be set up if the slider's starting point isn't the minimum value (for example, if you want the minimum value to be 1 but the starting point to be 5). To make this adjustment, set the `Value` property to 5. This code is on the form's `Load` event:

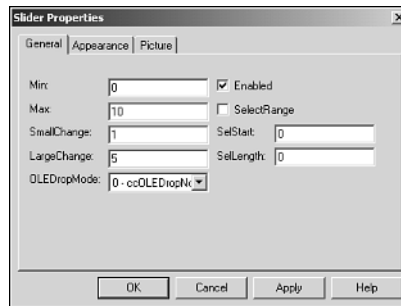
```
Private Sub Form_Load()
    Me!ocxSlider.Value = 5
End Sub
```

Because the code in Listing 14.7 is pretty well documented, let's look directly at some of the Slider control's more useful properties (see Table 14.3).

TABLE 14.3 Properties of the Slider Control

<i>Property</i>	<i>Description</i>
Min	Indicates the control's minimum value
Max	Indicates the control's maximum value
LargeChange	Indicates the increment for clicking the control, not for dragging the selector
SmallChange	Indicates the increment for dragging the selector
Orientation	Lets you display the control horizontally or vertically (0, horizontal, is the default)
TickFrequency	Controls how far apart the ticks should be
TickStyle	Positions the ticks at various angles on the control (0 = bottom right, 1 = top left, 2 = both, and 3 = none)

You can also specify the properties in the Access GUI. The last three properties are found on the Appearance page; the rest are on the General page. Figure 14.16 shows the General page of the `ocxSlider` control's property sheet.

**FIGURE 14.16**

Here's the General property page on the `ocxSlider` Slider control.

Telling It Like It Is with the Rich Textbox Control

Access text boxes are pretty handy to use as far as inputting data for storage and display. With the capability to specify scroll bars and use the `CanGrow/CanShrink` properties, they're as versatile as most other DBMSs in the market today. However, with the Windows technology and developers giving users more WYSIWYG ("what you see is what you get"), the plain ol' text box just doesn't cut it sometimes.

The ActiveX Rich Textbox control gives you another tool for enriching your interfaces. With the Rich Textbox control, you can

- Set fonts and font sizes
- Select text inside the control
- Specify special effects, such as boldface and italic
- Load and save text from or to files
- Use bullets
- Convert text to or from Rich Text Format
- Set paragraph alignment

The `frmWindowsCommonControlsRichText` form lets users load and save text from a Rich Textbox control. You also can select text inside the control and then toggle the text to be boldface or italic, and choose the alignment you desire. You also can print the text.

Figure 14.17 shows `frmWindowsCommonControlsRichText` as a subform on `frmWindowsCommonControls`.

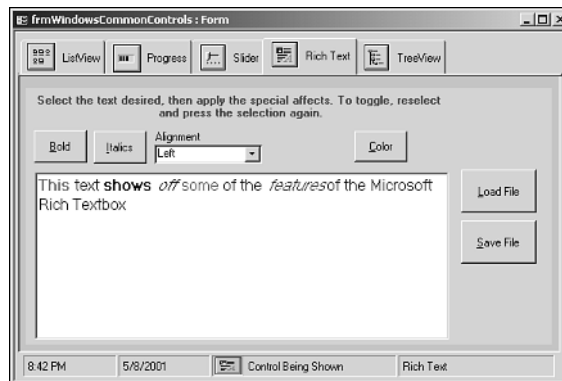


FIGURE 14.17

The Microsoft Rich Textbox control offers an abundance of features for formatting and presenting text.

Properties of the Rich Textbox Control

The code behind each button on the `frmWindowsCommonControlsRichText` form is examined in a moment. Table 14.4 lists some of the control's properties that affect the selected text.

TABLE 14.4 Properties of the Microsoft Rich Textbox Control

<i>Property</i>	<i>Description</i>
<code>SelAlignment*</code>	Affects alignment of selected text. Possibilities are 0 – Left, 1 – Right, and 2 – Center. Also returns the current setting.
<code>SelBold*</code>	Boldfaces or unbolds text based on the value set (0 – Un-Bold or 1 – Bold). Also returns the current setting.
<code>SelBullet</code>	Sets paragraphs with bullets, indented by whatever the <code>BulletIndent</code> property is set (0 – No Bullets, 1 – Bullets, or Null – Spans more than one paragraph). Also returns the current setting.
<code>SelColor*</code>	Sets the text's foreground color. Can be set by using <code>RGB()</code> or the <code>CommonDialog</code> ActiveX control. Also returns the current setting.
<code>SelFontName</code>	Specifies the selected text's font name. Can be set by supplying the name or by using the <code>CommonDialog</code> ActiveX control. Also returns the current setting.
<code>SelFontSize</code>	Specifies the selected text's font size. Can be set by supplying the size or by using the <code>CommonDialog</code> ActiveX control. Also returns the current setting.
<code>SelHangingIndent</code>	Used with <code>SelIndent</code> and <code>SelRightIndent</code> to specify paragraph indentation. Also returns the current setting.
<code>SelIndent</code>	Used with <code>SelHangingIndent</code> and <code>SelRightIndent</code> to specify paragraph indentation. Also returns the current setting.
<code>SelItalic*</code>	Applies or removes italic in text based on the value set (0 – Unset Italics, 1 – Italics). Also returns the current setting.
<code>SelLength</code>	Sets the amount of text to currently select. Used with <code>SelStart</code> . Also returns the current setting.
<code>SelRightIndent</code>	Used with <code>SelHangingIndent</code> and <code>SelIndent</code> to specify paragraph indentation. Also returns the current setting.
<code>SelRTF</code>	Converts selected text into Rich Text Format.
<code>SelStart</code>	Specifies starting position of selected text. If no code is selected, gives the cursor position.
<code>SelStrikethru</code>	Applies or removes strikethrough text based on the value set (0 – Unset Strikethru or 1 – Strikethru). Also returns the current setting.
<code>SelText</code>	Returns the currently selected text.
<code>SelUnderline*</code>	Applies or removes underline text based on the value set (0 – Unset Underline or 1 – Underline). Also returns the current setting.

**Example found on the frmWindowsCommonControlsRichText form*

Code Behind the Microsoft Rich Textbox Control

As with other properties, you can see the current status of each property by simply viewing it. On the frmWindowsCommonControlsRichText form, the cmdBold and cmdItalics buttons toggle the SelBold and SelItalic properties, respectively. Listing 14.8 shows the code for the cmdBold button; Listing 14.9 shows the cmdItalics button's OnClick code.

LISTING 14.8 Chap14.mdb: Changing Selected Text to Bold

```
Private Sub cmdBold_Click()  
    On Error GoTo Error_cmdBold_Click  
  
    '-- Toggle Bold  
    Me!ocxRichText.SelBold = Not Me!ocxRichText.SelBold  
  
Exit_cmdBold_Click:  
    Exit Sub  
  
Error_cmdBold_Click:  
    MsgBox Error$  
    Resume Exit_cmdBold_Click  
  
End Sub
```

LISTING 14.9 Chap14.mdb: Changing Selected Text to Italic

```
Private Sub cmdItalics_Click()  
    On Error GoTo Error_cmdItalics_Click  
  
    '-- Toggle Italics  
    Me!ocxRichText.SelItalic = Not Me!ocxRichText.SelItalic  
  
Exit_cmdItalics_Click:  
    Exit Sub  
  
Error_cmdItalics_Click:  
    MsgBox Error$  
    Resume Exit_cmdItalics_Click  
  
End Sub
```

Setting alignment takes a little bit more work, but not much. Because the SelAlignment property takes one of three values, a combo box is set up on the form to reflect these choices. Here is the RowSource for the cboAlignment combo box, which is a Value list:

```
0;"Left";1;"Right";2;"Center"
```

The AfterUpdate event contains the event routine that actually sets the SelAlignment property (see Listing 14.10).

LISTING 14.10 Chap14.mdb: Changing Selected Text Alignment

```
Private Sub cboAlignment_AfterUpdate()  
    On Error GoTo Error_cboAlignment_AfterUpdate  
  
    '-- Set the alignment  
    Me!ocxRichText.SelAlignment = Me!cboAlignment  
  
Exit_cboAlignment_AfterUpdate:  
    Exit Sub  
  
Error_cboAlignment_AfterUpdate:  
    MsgBox Error$  
    Resume Exit_cboAlignment_AfterUpdate  
  
End Sub
```

The last two pieces of code for the Rich Textbox control are for loading from and saving to files, using the text in the control. Listing 14.11 shows the cmdLoad_Click routine, which is attached to the OnClick event of the cmdLoad command button.

LISTING 14.11 Chap14.mdb: Loading a File into the Control

```
Private Sub cmdLoad_Click()  
    On Error GoTo Error_cmdLoad_Click  
  
    Me!ocxCommon.Filter = "Rich Text Format files|*.rtf"  
    Me!ocxCommon.ShowOpen  
    Me!ocxRichText.LoadFile Me!ocxCommon.FileName, 0  
  
Exit_cmdLoad_Click:  
    Exit Sub  
  
Error_cmdLoad_Click:  
    MsgBox Error$  
    Resume Exit_cmdLoad_Click  
  
End Sub
```

This routine uses a Common Dialog control, using the ShowOpen method, to get the file to open. The routine to save the text, in Listing 14.14, is similar to the load routine and is found on the cmdSave command button's OnClick event.

LISTING 14.14 Chap14.mdb: Saving a File from the Control

```

Private Sub cmdSaveFile_Click()
    On Error GoTo Error_cmdSaveFile_Click

    Me!ocxCommon.ShowSave
    Me!ocxRichText.SaveFile Me!ocxCommon.Filename, 0

Exit_cmdSaveFile_Click:
    Exit Sub

Error_cmdSaveFile_Click:
    MsgBox Error$
    Resume Exit_cmdSaveFile_Click

End Sub

```

You can use the Microsoft Rich Textbox control in a lot more tasks just by expanding on the frmWindowsCommonControlsRichText form.

Creating Status Bars for Individual Forms with the StatusBar Control

The ActiveX StatusBar control is one of the easiest of the controls to use. It can be pretty passive and just show the date and time, if you want, or it can be updated constantly and give a good picture of what's now taking place. The StatusBar control is used at the bottom of the main frmWindowsCommonControls form in Figure 14.18.

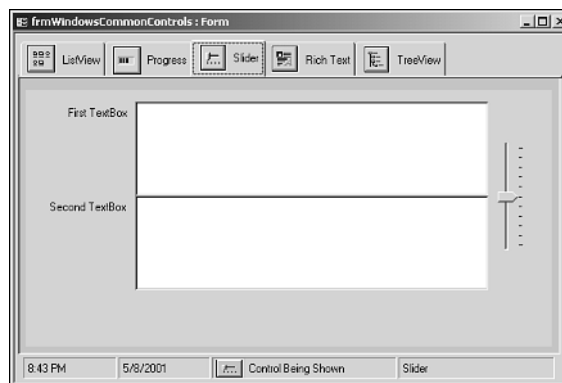


FIGURE 14.18

The StatusBar control is a great way to let the user know what's happening on each of your forms.

Like other ActiveX controls, the StatusBar control is partially made up of a collection. Some examples of collections used are the TabStrip control, which uses the Tabs collection, and the ImageList control, which uses the ListImages collection. The StatusBar control uses the Panels collection (examined in a moment).

The StatusBar control doesn't use as many properties from the overall control as some of the other ActiveX controls. Table 14.5 shows a few properties used overall by the StatusBar control.

TABLE 14.5 StatusBar Properties

<i>Property</i>	<i>Description</i>
Style	Allows you to choose multiple panels (0 – Normal) or a single panel (1 – Simple).
SimpleText	Sets the simple text displayed when the Style of 1 – Simple is chosen.
MousePointer	Lets you choose which style to use for the mouse pointer, the same as for the other ActiveX controls.

Properties of the StatusBar Panels Collection

The real power behind the StatusBar control lies in the Panels collection. With this collection, you can specify not only the number of panels to use, but also what style of information you want displayed in each panel. This includes Text, Caps Lock status, Num Lock status, Date, and Time. Table 14.6 lists some useful properties for the StatusBar Panel object.

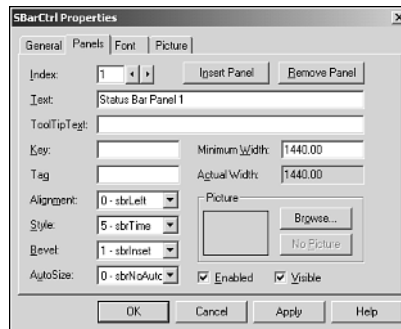
TABLE 14.6 The StatusBar Panel Object's Properties

<i>Property</i>	<i>Description</i>
Index	The current panel index (read-only)
Text	Text to be displayed; used when the Style property is set to 0 – Text.
Key	As with other control collections, allows you to set a text value to refer to the current panel rather than to the index.
Alignment	How you want to align the information in the current panel: 0 – Left, 1 – Center, or 2 – Right.
Style	Style of information to be displayed: 0 – Text, 1 – Caps, 2 – Num Lock, 3 – Ins, 4 – Scroll, 5 – Time, 6 – Date, and 7 – Kana Lock (used for Asian characters).
Bevel	The special effect you want to use with the panel: 0 – None, 1 – Inset, or 2 – Raised.

TABLE 14.6 Continued

<i>Property</i>	<i>Description</i>
AutoSize	How you want the size of the panel to be affected: 0 – None means no autosize, and the size stays at whatever the panel width is set; 1 – Spring changes the panel width with the width of the form; 2 – Contents matches the panel's width with the width of the panel's contents.
Picture	A bitmap or icon.

You can set these properties at design time or runtime. The one most likely to be set at runtime is the Text property. Figure 14.19 shows the panel properties.

**FIGURE 14.19**

Each panel's properties can be set independently of each other.

Setting Status Bar Properties at Runtime

The fourth and fifth panels of the StatusBar control located on the `frmWindowsCommonControls` form are set at runtime by the `UpdateStatusBarControl` routine, found in the General Routines in the form module.

To open the module to the routine desired, follow these steps:

1. Open the `frmWindowsCommonControls` form in Design mode.
2. Click the Code toolbar button on the Form Design toolbar.
3. Select `UpdateStatusBarRoutine` from the Procedure combo box.

The routine in Listing 14.13 performs two actions: it sets the picture property of the fourth panel to the selected tab picture and displays (in the fifth panel) the caption of the chosen tab.

NOTE

Like other collections in Access and other Windows development languages, the collections are zero-based. This is why the index that says 3 is the fourth panel.

LISTING 14.13 Chap14.mdb: Updating the Status

```
Sub UpdateStatusBarControl(intChoice As Integer)
    On Error GoTo Error_UpdateStatusBarControl

    '-- Set the fourth panel to the currently selected tab
    Set Me!ocxStatusBar.Panels(3).Picture =
        Me!ocxImageList.ListImages(intChoice).Picture

    '-- Set the fifth panel the caption of the currently selected tab
    Me!ocxStatusBar.Panels(4).Text = Me!ocxTabStrip.Tabs(intChoice).Caption

Exit_UpdateStatusBarControl:
    Exit Sub

Error_UpdateStatusBarControl:

    MsgBox Error$
    Resume Error_UpdateStatusBarControl

End Sub
```

The `intChoice` parameter is passed into the subroutine to indicate which tab is active.

Docking Toolbars on Forms Using the ToolBar Control

For the most part, Access's built-in toolbars do a good enough job when it comes to using them in applications. The ActiveX ToolBar control has one advantage over the Access toolbars in that you can place a toolbar on an individual form. The way this is demonstrated here is to emulate the main `frmWindowsCommonControls` form by using a ToolBar control instead of a TabStrip control. The form used, `frmWindowsCommonControlsToolBar`, is in the `Chap14.mdb` database (on this book's Web page at www.sampublishing.com).

The ToolBar control can take images supplied by the ImageList control just like the TabStrip and a few of the other Windows interface controls do. Figure 14.20 shows the

frmWindowsCommonControlsToolBar form, which uses the ToolBar control complete with ScreenTips. The images are supplied with a copy of the ImageList control; this control is covered earlier in the section “Using the ImageList Control with the TabStrip Control.”



FIGURE 14.20

You can place ToolBar controls on individual forms.

Creating a ToolBar control follows along the lines of the other controls in that you insert buttons, which are kept in a collection called Buttons, and each has a set of properties. Figure 14.21 shows the properties for a button on the ToolBar control’s property sheet. (To open the property sheet, simply double-click the ToolBar control.)



FIGURE 14.21

Each button has a set of properties on the ToolBar control.

You can create different toolbar button styles, based on what you set the `Style` property to be for the individual buttons. The possible settings for the `Style` property are

- 0 – `tbrDefault`, the default standard toolbar button.
- 1 – `tbrCheck`, a check button; can be checked and unchecked.
- 2 – `tbrButtonGroup`, which stays selected until another button is clicked.
- 3 – `tbrSeparator`, used for separating buttons on the toolbar. It's a fixed width of 8 pixels.
- 4 – `tbrPlaceholder`, which is the same as `tbrSeparator`, except that you can specify a width by using the `Width` property.
- 5 – `tbrDropDown`, which allows for the use of a drop-down control.

At runtime, the `ToolBar` control has a `ButtonClick` event that you use from other controls. When you create a `ButtonClick` event routine, Access automatically sets it up to pass the current button in as an object. Listing 14.14 shows this event routine.

LISTING 14.14 Chap14.mdb: Swapping Source Objects Based on a Button

```
Private Sub ocxToolbar_ButtonClick(ByVal Button As Object)
    Me.Painting = False
    Me!ocxToolbar.SetFocus

    Select Case Button.KEY
        Case "ListView"
            Me!subControlExample.SourceObject = "frmWindowsCommonControlsListView"
        Case "Progress"
            Me!subControlExample.SourceObject = "frmWindowsCommonControlsProgressBar"
        Case "Slider"
            Me!subControlExample.SourceObject = "frmWindowsCommonControlsSlider"
        Case "Rich"
            Me!subControlExample.SourceObject = "frmWindowsCommonControlsRichText"
        Case "TreeView"
            Me!subControlExample.SourceObject = "frmWindowsCommonControlsTreeView"
    End Select

    UpdateStatusBarControl Button.Index

    Me.Painting = True
End Sub
```

This code performs the following steps:

1. The current toolbar button is passed in as an object.
2. Painting is turned off, and the focus is set to the `ToolBar` control named `ocxToolbar`.

3. Its `Select Case` statement reads the `Key` property that has been set and swaps out the `SourceObject` property accordingly.
4. The `UpdateStatusBarControl` routine is called with the `Button` index property being passed.
5. Painting is turned on.

The `UpdateStatusBarControl` routine in step 4 updates the ActiveX `StatusBar` control located at the bottom of the form. The code for this is as follows:

```
Sub UpdateStatusBarControl(intChoice As Integer)
    Set Me!ocxStatusBar.Panels(3).Picture = _
        Me!ocxImageList.ListImages(intChoice).Picture
    Me!ocxStatusBar.Panels(4).Text = Me!ocxToolbar.Buttons(intChoice).ToolTipText
End Sub
```

The `UpdateStatusBarControl` routine first updates one of the panels of the `StatusBar` control with the image of the current `ToolBar` control button selected. Next, another panel is updated with the `ToolTipText` property of the current `ToolBar` control button.

For more information on using the `StatusBar` control, see the earlier section “Creating Status Bars for Individual Forms with the `StatusBar` Control.”

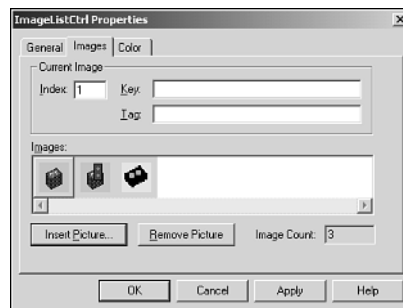
Viewing Data File Manager Style with the TreeView Control

The last control to examine is `TreeView`. This ActiveX control lets you view your data in a manner similar to Windows Explorer, with the added feature of being able to include graphics images. You can find an example of a `TreeView` control on the `frmWindowsCommonControlsTreeView` form in `Chap14.mdb`, which is available from this book’s Web page at www.sampublishing.com. Figure 14.22 shows the `frmWindowsCommonControlsTreeView` form as a source object for the `frmWindowsCommonControls` form.

Notice in Figure 14.22 how the expanded category, called a *node*, displays a different image than an unexpanded node. This `TreeView` control uses three different images: unexpanded node, expanded node, and leaf node (sometimes called the *bottom node*). Figure 14.23 shows these images in the `ImageList` control named `ocxImageList`. (For more information on setting up the `ImageList` control, see the earlier section “Using the `ImageList` Control with the `TabStrip` Control.”)

**FIGURE 14.22**

The ImageList control is again used for the TreeView control to store the bitmaps displayed.

**FIGURE 14.23**

The highlighted image is used for the expanded node.

After setting up the images, you need to set the TreeView control's ImageList property to `ocxListImageSub`.

NOTE

Call this ListImage control `ocxListImageSub` instead of just `ocxListImage`. Otherwise, Access gets confused if you have two ListImage controls open at the same time—even on two separate forms. (It's a problem because `frmWindowsCommonControlsTreeView` is used as a subform for the `frmWindowsCommonControls` form.) To work around this, add `Sub` to the subform ListImage control name.

PART III

In addition to setting up the ImageList control, you need to set the `Style` property so that it displays images with lines. Eight different styles are available:

<i>Value</i>	<i>Description</i>
0	Text Only (default)
1	Picture, Text
2	Plus/Minus, Text
3	Plus/Minus, Picture, Text
4	Lines, Text
5	Lines, Picture, Text
6	Lines, Plus/Minus, Text
7	Lines, Plus/Minus, Picture, Text

The style chosen for this example is 7 (Lines, Plus/Minus, Picture, Text). You can see Style 7, as well as other properties for the TreeView control, in Figure 14.24.

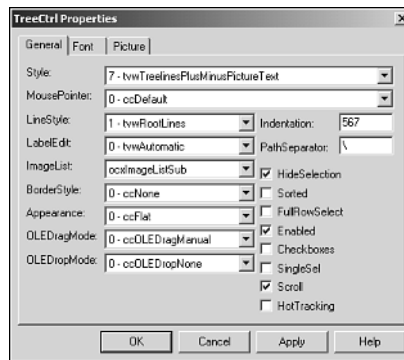


FIGURE 14.24

You can control how the TreeView looks by setting these properties.

That's all you have to do to set up a TreeView for `frmWindowsCommonControlsTreeView`. The rest is performed in Listing 14.15's code, which is on the `OnLoad` event of the `frmWindowsCommonControlsTreeView` form.

LISTING 14.15 Chap14.mdb: Populating the TreeView Control

```
Private Sub Form_Load()
    Dim objCurrNode As Object
    Dim rstCategories As New ADODB.Recordset
    Dim rstMovies As New ADODB.Recordset
```

LISTING 14.15 Continued

```

'-- Open the necessary recordset
rstCategories.Open "tblCategories", CurrentProject.Connection
rstMovies.Open "qryMoviesByCategories", CurrentProject.Connection

'-- Loop through the categories
Do Until rstCategories.EOF
    '-- Add a top node (Category)
    Set objCurrNode = Me!ocxTreeView.Nodes.Add(, , _
        "Category" & CStr(rstCategories!CategoryCode), _
        rstCategories!Description, 1)
    '-- Establish which image to use for an expanded branch
    objCurrNode.ExpandedImage = 2
    If Not rstMovies.EOF Then
        '-- Add a branch to the current top node (Category)
        Do While rstMovies!CategoryCode = rstCategories!CategoryCode
            Set objCurrNode = Me!ocxTreeView.Nodes.Add( _
                "Category" & CStr(rstCategories!CategoryCode), 4, , _
                rstMovies!Title, 3)
            rstMovies.MoveNext
            If rstMovies.EOF Then
                Exit Do
            End If
        Loop
    End If
    rstCategories.MoveNext
Loop

Me!ocxTreeView.Refresh

End Sub

```

NOTE

This code is completely dependent on the way you set up your sorting design for `tblCategories` and `qryMoviesbyCategory`. If the Category order in the `qryMoviesbyCategory` varies from the order in the `tblCategories` table, this code won't work.

The code in Listing 14.15 actually takes the data from within tables and stores it into the TreeView control's nodes. Similar to other ActiveX controls, the TreeView control uses a collection, called Nodes. You see this collection in the steps this code performs:

1. Open the necessary recordsets.
2. Loop through the `tblCategories` recordset. This is the top node in the `TreeView`.
3. For each category, add a top node. This is performed with the `Add` method off the `Nodes` collection. The 1 in the `Add` method means it's a parent node:

```
Set objCurrNode = Me!ocxTreeView.Nodes.Add(, , _  
    "Category" & CStr(rstCategories!CategoryCode), _  
    rstCategories!Description, 1)
```

4. Establish which image to use for an expanded branch:

```
objCurrNode.ExpandedImage = 2
```
5. Add the branches to the current top node. The movies are looped through according to their category. The `Add` method is again used with a 3, which tells Access that this is a leaf, or *child node*:

```
Do While rstMovies!CategoryCode = rstCategories!CategoryCode  
    Set objCurrNode = Me!ocxTreeView.Nodes.Add( _  
        "Category" & CStr(rstCategories!CategoryCode), 4, , _  
        rstMovies!Title, 3)
```

6. Refresh the `TreeView` control.

The `TreeView` control can use quite a few properties and methods through the `Nodes` collection and the control itself. This example just touched on some of the `TreeView` control's power.

TIP

If performance becomes an issue when using the `TreeView` control, add nodes on lower levels just in time, as they're expanded, by using the `Expand` event. You can add a dummy node at the higher level so that the little + shows up. (The plus sign doesn't show if there are zero child nodes.)

Summary

The ActiveX Common Controls add more functionality to Access. Knowing how to use these controls is merely a matter of finding out which properties and methods each one uses. Third-party vendors provide additional controls all the time.

- Chapter 4, "Working with Access Collections and Objects," deals with navigating through collections and shows examples of some of the collections available to Access.
- Appendix B, "Getting Started with ActiveX Controls," gives you a good basic understanding on how to work with ActiveX controls. You can find this appendix at www.sampublishing.com (enter this book's ISBN in the Search field).

Extending the Power of Access with API Calls

CHAPTER

15

IN THIS CHAPTER

- Understanding Dynamic Link Libraries 463
- Examining the Syntax for API Calls 463
- Finding API Declarations 466
- Viewing the Possible API Calls 466
- Considering Some Issues When Using API Calls 470
- Looking at Some Examples of API Calls 471
- Displaying Pertinent Folders from Within Your Application 483
- Using the Open File Dialog API Call 486

API (Application Programming Interface) calls, sometimes also referred to as *routines*, are called from dynamic link libraries (.dll). These routines allow you to expand the power that Access already gives you in the VBA programming language. One example of using an API call would be when you need to connect to a share on a network programmatically. Although you can't do this with VBA, some API routines will accomplish this.

Access continues to expand in its native language, including adding features that you once handled by making API calls. Some tasks still need to be handled by using VBA with the API.

Most languages that allow you to develop at the system level use API calls. The good news is that when you learn the calls in one language, switching to another language to make the same call doesn't require a great deal of modification. This is especially true between VB and Access VBA. In fact, many API calls you "borrow" will possibly come from applications written in Visual Basic.

TIP

It's highly recommended that you "borrow" code that has already proven to work with the API call you're trying to use, for two reasons:

- Why re-create something that has already been done?
- One of the most frustrating things about using API calls is the lack of good sample code for all the functions.

Finding code that already has a good wrapper function around the API call also will save countless hours of development time. A VBA *wrapper* routine sets up the necessary environment for calling the routines, such as needed variables. In some cases, you can skip using a wrapper routine by simply calling the routine directly where it's needed.

Chapter 16, "Extending Your VBA Library Power with Class Modules and Collections," covers class modules, which are great for encapsulating code and making it very painless to use API calls with properties and methods. The final example in this chapter, using the Open File dialog, is also used in Chapter 16 with a class module.

CAUTION

The frustration factor is especially true because when you're dealing with API calls, you can GPF (General Protection Fault) quite easily if they aren't set up correctly. GPFs are more controlled in Windows in that you don't have to reboot. However, you can pretty well count on losing whatever work wasn't saved. The moral of the story is save often, especially before you try out an API call.

Understanding Dynamic Link Libraries

Before jumping into the syntax of API calls and how to use them, let's look at the files these routines are located in—dynamic link libraries. DLLs are exactly what the name implies—libraries of routines that are linked dynamically at runtime.

In the DOS days, if you made a change to a routine library, separate from the application or to the application itself, you would have to link both again. Not true with DLLs. A DLL doesn't get pulled into an application until runtime, so you can make changes independently and not have to worry about compiling and linking the files. Of course, you still have to be careful that the calls that you make to the DLL will still work after your modification.

TIP

Routines kept in DLLs are pulled into memory individually with some overhead, rather than the whole library, thereby saving memory. When the routine is completed, it's then released, freeing up memory again.

Some of the most commonly used DLLs are those used by Windows itself: Advapi32.dll, Gdi32.dll, Kernel32.dll, Mpr.dll, Winmm.dll, and User32.dll. Others that might be useful are Winspool.dll (printer manipulation), Shell32.dll (shell commands not within Kernel32), NETAPI32.dll (complementary to mpr for network connections and setup), and comdlg32.dll (for dialogs). You'll see API calls with these files as the libraries later, in the section "Looking at Some Examples of API Calls."

Examining the Syntax for API Calls

Calling API routines from Access generally takes at least two steps, sometimes three:

1. Declare the function in a module's Declarations section.
2. (Optional) Create a wrapper routine to call the routine.
3. Call the API routine.

The syntax for these steps is very much like those used when calling VBA routines. The interesting step is the first, the declaration of an API routine. (You might have noticed that the terms *routines* and *calls* have been used rather than *subroutines* and *functions*. Like VBA routines, API routines can be subroutines or functions.)

In a module's Declarations section, you'll have the following syntax for each API call. For a subroutine-type API call, you would have

PART III

```
[Public | Private] Declare Sub Name Lib "LibName" _  
    [Alias AliasName] ([ArgList])
```

For a function-type API call, you would have

```
[Public | Private] Declare Function Name Lib "LibName" _  
    [Alias "AliasName"] ([ArgList]) [As Type]
```

These parameters are used:

- *Name* changes based on whether the routine is being called with an alias. If an alias is used, this should be a name that you make up so that it doesn't conflict with other names and Access commands. If no alias is used, this should be the name of the routine used in the DLL.
- *LibName* is the name of one of the DLLs mentioned earlier in the section "Understanding Dynamic Link Libraries."

NOTE

If the DLL being used isn't one of the regular Windows DLLs, you might have to supply the full path and name. When using a DLL that's located in the System path, you simply have to use the name of the DLL.

- *AliasName* should be the routine's actual name when you use an alias.
- *ArgList* is the list of arguments, similar to how Access passes arguments, although the types vary somewhat in certain cases.

NOTE

By default, Access passes variables by reference. Many API calls expect parameters by value instead, so use the `ByVal` keyword before the argument. For more information on passing arguments, see Chapter 2, "Coding in Access 2002 with VBA."

- *Type* refers to the data type of the return value and is used for functions only. The data type of the return value can be Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (variable length only), Object, Variant, a user-defined type, or an object type.

Listing 15.1 shows the Declarations section of the module modWindowsAPIUtils, located in Chap15.mdb on this book's Web page at www.sampublishing.com. Notice that each declaration uses the Alias statement and puts wu_ before the routine's name, so that no conflicts will occur between existing routines and the new API calls being made.

LISTING 15.1 Chap15.mdb: Sample API Declarations

```
Option Compare Database
Option Explicit
'--API Call for finding the associated file
Declare Function wu_FindExecutable Lib "shell32.dll" Alias _
    "FindExecutableA" (ByVal lpFile As String, _
    ByVal lpDirectory As String, ByVal lpResult As String) As Long
'--API Calls for adding and cancelling network connection programmatically
Declare Function wu_NetAddConnection Lib "mpr" Alias "WNetAddConnectionA" _
    (ByVal NetPath$, ByVal Password$, ByVal LocalDrive$) As Long
Declare Function wu_NetCancelConnection Lib "mpr" Alias _
    "WNetCancelConnectionA" (ByVal NetPath$, ByVal FileForce%) As Long
'--API Calls for calling the network connect and disconnect dialogs
Declare Function wu_WNetConnectDialog Lib "mpr" Alias _
    "WNetConnectDialog" (ByVal hwnd As Long, ByVal dwType As Long) As Long
Declare Function wu_WNetDisconnectDialog Lib "mpr" Alias _
    "WNetDisconnectDialog" (ByVal hwnd As Long, ByVal dwType As Long) As Long
'--API Calls for getting the current user and computer names
Declare Function wu_GetUserName Lib "advapi32.dll" Alias _
    "GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As Long
Declare Function wu_GetComputerName Lib "kernel32" Alias _
    "GetComputerNameA" (ByVal lpBuffer As String, nSize As Long) As Long
'--API Calls for getting the windows, system, and temp directory for
'--the current computer.
Declare Function wu_GetWindowsDirectory Lib "kernel32" Alias _
    "GetWindowsDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long
Declare Function wu_GetSystemDirectory Lib "kernel32" Alias _
    "GetSystemDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long
Declare Function wu_GetTempPath Lib "kernel32" Alias "GetTempPathA" _
    (ByVal nBufferLength As Long, ByVal lpBuffer As String) As Long
```

NOTE

These calls all have an A on the end of them, as in GetComputerNameA, because these routines are the ANSI version. Some calls would show W for the Unicode version.

If you examine the first declaration,

```
Declare Function wu_FindExecutable Lib "shell32.dll" Alias _  
    "FindExecutableA" (ByVal lpFile As String, _  
        ByVal lpDirectory As String, ByVal lpResult As String) As Long
```

notice that it's pretty much a by-the-book declaration. You can break it down as follows:

- The routine being called is a function.
- This application is named `wu_FindExecutable`, so that it won't conflict with another one, possibly named `FindExecutableA`.
- Three parameters are being passed by value; all of them are strings.
- The return value is a `Long`.

This routine is classic also in that the return value actually returns whether it was successful. The actual desired answer—the path and filename of the executable program associated with the given `lpFile` and `lpDirectory`—is passed back in `lpResult`, as shown in detail later in “Finding an Executable Application Associated with a File.”

You'll also often see the use of user-defined variables to pass arguments to API calls. The routine found later in the section “Using the Open File Dialog API Call” uses a structure of type `FILE`.

Finding API Declarations

You might be wondering where to find these API calls. You can find them in several places:

- On this book's Web page at www.sampublishing.com, in the `Win32api.txt` file.
- On Microsoft Developer's Network CDs, a service you can get from Microsoft. These CDs have a number of articles on programming with API calls.
- In Microsoft Office Developer (MOD), which has 16- and 32-bit versions of API declarations. An API Viewer is included, with data stored in text and `.mdb` format. The following section examines this tool further.
- In a number of VB books that contain information on calling API routines. One such book is Que's *Special Edition Using Microsoft Access 2002* by Roger Jennings (ISBN 0-7897-2510-X).

Viewing the Possible API Calls

You can view API calls immediately in a couple of ways: by using the API Viewer that comes with Microsoft Office Developer (MOD) or by copying the example in the `Win32api.txt` text file.

Using the API Viewer to Locate Calls

A good source for API calls is the API Viewer that comes with MOD. This viewer is useful for getting an idea of the available API calls. To use it, follow these steps:

1. From the Start menu, choose Programs, Microsoft Office Developer, and finally API Viewer. A blank viewer appears.
2. From the File menu, choose Load Text File. The Select a Text API File dialog appears.
3. Select Win32api.txt and click Open to populate the Available Items list box with API call names (see Figure 15.1).

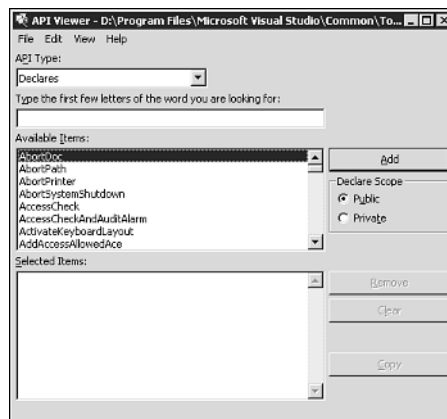


FIGURE 15.1

You can not only see the available API calls, but also copy and paste them into your application.

TIP

You might have noticed the Load Text File command on the File menu. This text file is the same one available on this book's Web page at www.sampublishing.com. It's faster to use the database method through the API Viewer, however.

The API Type drop-down list allows you to choose from Constants, Declares, and Types. Constants and Types are actually used with the arguments used in declarations.

Copying and Pasting Calls from the API Viewer

To copy and paste calls from the API Viewer, follow these steps:

1. Click the Available Items list box.
2. Type **GetSystemDirectory**. The list box scrolls to the API name you typed.
3. Click Add. GetSystemDirectory shows up in the Selected Items list box.
4. Click the Available Items list box again and type **GetWindowsDirectory**.
5. Click Add again. You see the GetWindowsDirectory call in the Selected Items list box, under GetSystemDirectory.
6. Click the Copy button.

Now you need to load Access and pull the declarations into the actual module where they're needed. To do so, open Access and then follow these steps:

1. Open the Chap15.mdb database, on this book's Web page at www.sampublishing.com.
2. Open the modWindowsAPIRoutines module.
3. Go to the Declarations section and choose Paste from the Edit menu. (Or right-click the Declarations section and choose Paste.)

The full declarations should be pasted in the module. It saves a lot of hassle not having to remember the correct syntax. You can see the full declarations that were copied, as follows:

```
Declare Function GetSystemDirectory Lib "kernel32" Alias _
    "GetSystemDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long
Declare Function GetWindowsDirectory Lib "kernel32" Alias _
    "GetWindowsDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long
```

Now all you need to do is place `wu_` on the front of the name of the declarations. They would then look like this:

```
Declare Function wu_GetSystemDirectory Lib "kernel32" Alias _
    "GetSystemDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long
Declare Function wu_GetWindowsDirectory Lib "kernel32" Alias _
    "GetWindowsDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long
```

These two declarations are used in actual calls to display the Windows main folder and system folder later in the section "Looking at Some Examples of API Calls."

Finding API Calls in the Win32api.txt File

You can look at API calls in another way, by using the Win32api.txt available with the MOD and on this book's Web page at www.sampublishing.com. Then double-click the Win32api.txt

file. It will then appear in Notepad or WordPad, depending on how much memory you have available.

In addition to the disclaimer about it being royalty-free and unsupported, notice that the name of the file is Win32 API Declarations for Visual Basic. This is what I meant earlier when I said that these declarations are literally identical in the two products, VB and Access.

As you page down through the file, notice the constants, user-defined type declarations, and the API declarations themselves. To use the declarations given in the Win32api.txt file, simply copy and paste as you would with any other text into the Declarations section of an Access module.

TIP

Paging through the Win32api.txt file is a great way to learn the groupings of the API calls. For example, find the word *WNet* by choosing Find from the Edit menu, and then typing *WNet*. Now click Find Next. If you scroll down, notice that all the networking routines are grouped together (see Figure 15.2).

```

dwScope As Long
dwType As Long
dwDisplayType As Long
dwUsage As Long
lpLocalName As String
lpRemoteName As String
lpComment As String
lpProvider As String
End Type

Const CONNECT_UPDATE_PROFILE = &H1

Declare Function WNetAddConnection Lib "mpr.dll" Alias "WNetAddConnectionA" (ByVal lpzNetPath As String, ByVal lpLocalName As String, ByVal dwFlags As Long) As Long
Declare Function WNetAddConnection2 Lib "mpr.dll" Alias "WNetAddConnection2A" (lpNetResource As NetResource, lpLocalName As String, lpRemoteName As String, dwFlags As Long) As Long
Declare Function WNetCancelConnection Lib "mpr.dll" Alias "WNetCancelConnectionA" (ByVal lpzNetPath As String) As Long
Declare Function WNetCancelConnection2 Lib "mpr.dll" Alias "WNetCancelConnection2A" (ByVal lpName As String, dwFlags As Long) As Long
Declare Function WNetGetConnection Lib "mpr.dll" Alias "WNetGetConnectionA" (ByVal lpzLocalName As String, ByVal lpzRemoteName As String, ByVal dwFlags As Long) As Long
Declare Function WNetOpenEnum Lib "mpr.dll" Alias "WNetOpenEnumA" (ByVal dwScope As Long, ByVal dwType As Long, dwFlags As Long, lpProvider As String) As Long
Declare Function WNetEnumResource Lib "mpr.dll" Alias "WNetEnumResourceA" (ByVal hEnum As Long, lpBuffer As Long, lpBytesReturned As Long) As Long
Declare Function WNetCloseEnum Lib "mpr.dll" Alias "WNetCloseEnumA" (ByVal hEnum As Long) As Long

Declare Function WNetGetUser Lib "mpr.dll" Alias "WNetGetUserA" (ByVal lpName As String, ByVal lpProvider As String, lpReserved As Long) As Long
Declare Function WNetConnectionDialog Lib "mpr.dll" Alias "WNetConnectionDialogA" (ByVal hwnd As Long, lpzNetPath As String, lpzLocalName As String, lpzRemoteName As String, dwFlags As Long) As Long
Declare Function WNetDisconnectDialog Lib "mpr.dll" Alias "WNetDisconnectDialogA" (ByVal hwnd As Long, lpzNetPath As String, lpzLocalName As String, lpzRemoteName As String, dwFlags As Long) As Long
Declare Function WNetGetLastError Lib "mpr.dll" Alias "WNetGetLastErrorA" (lpError As Long, ByVal lpProvider As String, lpzNetPath As String, lpzLocalName As String, lpzRemoteName As String) As Long

' Status Codes
' This section is provided for backward compatibility. Use of the ERROR_
' codes is preferred. The WNet_error codes may not be available in future
' releases.
' General
Const WN_SUCCESS = NO_ERROR
Const WN_NOT_SUPPORTED = ERROR_NOT_SUPPORTED
Const WN_NET_ERROR = ERROR_UNEXP_NET_ERR
Const WN_MORE_DATA = ERROR_MORE_DATA
Const WN_BAD_POINTER = ERROR_INVALID_ADDRESS
Const WN_BAD_VALUE = ERROR_INVALID_PARAMETER
  
```

FIGURE 15.2

These network routines are a good example of how API calls are grouped in the Win32api.txt file.

Another great feature is being able to grab the type declarations right out of this file. The type declaration shown in Figure 15.3, `WNDCLASS`, is used in the API call declaration that follows it.



FIGURE 15.3

Rather than enter the user-defined types needed by some API calls, just copy and paste.

Considering Some Issues When Using API Calls

Before jumping into some examples, let's look at some of the issues of using API calls, such as what to keep in mind when creating your own API declarations from scratch, and what it takes to convert 16-bit API calls to 32-bit API calls.

Creating Your Own API Declarations from Scratch

Two words describe the situation when you find an obscure declaration, and there isn't a document declaration already built: *Be careful!* Although it's very rare that anything dangerous will happen, in some cases people have literally trashed their hard drives or reset their CMOS by using API calls they weren't quite sure of. As mentioned earlier, one of the most common problems is that your system will GPF, and you could lose whatever changes you made to your code if you haven't saved recently. It's always better if you can find routines that have sample code for them or get them from other developers.

Converting 16-Bit to 32-Bit API Calls

When you move into the Windows 95/98/NT world from Windows 3.x or Windows for Workgroups 3.1, you switch any 16-bit API calls to 32-bit. You need to remember a few things that will get you past most API conversions:

- Start over from scratch. (Just kidding!)
- The DLL names themselves changed. Here’s a list of known DLL name changes:

<i>16-Bit Name</i>	<i>32-Bit Name(s)</i>
Kernel.dll	Kernel32.dll
User.dll	User32.dll, Mpr.dll (network routines)
Gdi.dll	Gdi32.dll

- The names of the API routines also changed. As mentioned earlier in the section “Examining the Syntax for API Calls,” the most notable change is the addition of the A to the end of the actual API routine to indicate that it’s an ANSI version. W is used for Unicode.

TIP

Typically, if you get a Can't find routine message when switching to 32-bit, add a 32 to the end of the library called and an A to the actual name of the routine.

- Some arguments and return data types have changed. The most common are those that were once Integer are now Long. This makes sense; because the operating system has switched from 16-bit to 32-bit, it requires the capability to return larger (32-bit) values.

That’s about it for switching from 16-bit to 32-bit. You will have to go through and make these changes yourself. It will take some patience in converting your application, but it can be fairly painless.

Looking at Some Examples of API Calls

The rest of this chapter will look at some actual examples of API calls being used. Before jumping into the individual examples, however, look again at the API declarations used for all of them, merely for your convenience and to refresh them in your mind (see Listing 15.2). These can be found in the module modWindowsAPIRoutines, in Chap15.mdb.

LISTING 15.2 Chap15.mdb: Declaring API Calls

```

Option Compare Database
Option Explicit
'--API Call for finding the associated file
Declare Function wu_FindExecutable Lib "shell32.dll" Alias _
    "FindExecutableA" (ByVal lpFile As String, _
        ByVal lpDirectory As String, ByVal lpResult As String) As Long
'--API Calls for adding and cancelling network connection programmatically
Declare Function wu_NetAddConnection Lib "mpr" Alias "WNetAddConnectionA" _
    (ByVal NetPath$, ByVal Password$, ByVal LocalDrive$) As Integer
Declare Function wu_NetCancelConnection Lib "mpr" Alias _
    "WNetCancelConnectionA" (ByVal NetPath$, ByVal FileForce%) As Integer
'--API Calls for calling the network connect and disconnect dialogs
Declare Function wu_WNetConnectionDialog Lib "mpr" Alias _
    "WNetConnectionDialog" (ByVal hwnd As Long, ByVal dwType As Long) As Long
Declare Function wu_WNetDisconnectDialog Lib "mpr" Alias _
    "WNetDisconnectDialog" (ByVal hwnd As Long, ByVal dwType As Long) As Long
'--API Calls for getting the current user and computer names
Declare Function wu_GetUserName Lib "advapi32.dll" Alias _
    "GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As Long
Declare Function wu_GetComputerName Lib "kernel32" Alias _
    "GetComputerNameA" (ByVal lpBuffer As String, nSize As Long) As Long
'--API Calls for getting the windows, system, and temp directory for
'--the current computer.
Declare Function wu_GetWindowsDirectory Lib "kernel32" Alias _
    "GetWindowsDirectoryA" (ByVal lpBuffer As String, _
        ByVal nSize As Long) As Long
Declare Function wu_GetSystemDirectory Lib "kernel32" Alias _
    "GetSystemDirectoryA" (ByVal lpBuffer As String, _
        ByVal nSize As Long) As Long
Declare Function wu_GetTempPath Lib "kernel32" Alias _
    "GetTempPathA" (ByVal nBufferLength As Long, _
        ByVal lpBuffer As String) As Long

```

Finding an Executable Application Associated with a File

The first example will use the `FindExecutable()` API routine. This function is useful when you want to let users choose between different file types to open. Rather than hard-code which application to use, you can check to make sure that it's associated with an application.

The form used for this example, `frmFindExecutableExample`, is located in `Chap15.mdb` on this book's Web page at www.sampublishing.com. The form consists of a combo box named `cboFileToFindExe`, a text box named `txtExecutablePath`, and a command button named `cmdOpenApplication`.

cboFileToFileExe is interesting because it uses a function to populate itself. This function, ListFiles(), is placed in the combo box's Row Source Type property (see Figure 15.4). Listing 15.3 shows the code used in the ListFiles() function.

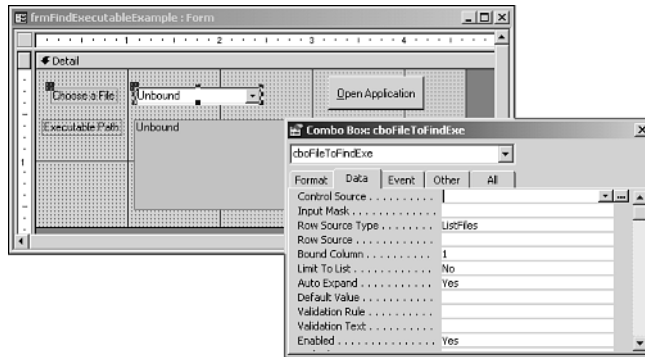


FIGURE 15.4

Using a function to populate a combo box is sometimes the only way to quickly create one dynamically.

LISTING 15.3 Chap15.mdb: Filling in a Combo Box

```
Function ListFiles(fld As Control, id As Variant, row As Variant, _
    col As Variant, code As Variant) As Variant
    Static astrFiles(127) As String, intEntries As Integer
    Dim varReturnVal As Variant
    varReturnVal = Null

    Select Case code
        Case LB_INITIALIZE      ' Initialize.
            intEntries = 0
            '--Get the path of the application
            strAppPath = Left$(CurrentDb.Name, InStrRev(CurrentDb.Name, "\"))
            '--Populate the array
            astrFiles(intEntries) = Dir(strAppPath & "*.*)")
            Do Until astrFiles(intEntries) = "" Or intEntries >= 127
                intEntries = intEntries + 1
                astrFiles(intEntries) = Dir
            Loop
            varReturnVal = intEntries
        Case LB_OPEN            ' Open.
            varReturnVal = Timer      ' Generate unique ID for control.
        Case LB_GETROWCOUNT    ' Get number of rows.
            varReturnVal = intEntries
```

LISTING 15.3 Continued

```

    Case LB_GETCOLUMNCOUNT      ' Get number of columns.
        varReturnVal = 1
    Case LB_GETCOLUMNWIDTH      ' Column width.
        varReturnVal = -1      ' -1 forces use of default width.
    Case LB_GETVALUE            ' Get data.
        varReturnVal = astrFiles(row)
    Case LB_END                  ' End.
        For intEntries = 0 To 127
            astrFiles(intEntries) = ""
        Next
    End Select
    ListFiles = varReturnVal
End Function

```

The `ListFiles()` function is standard for populating a combo box programmatically. The combo box places all the filenames in the current folder in the `astrFiles` array.

When the combo box is populated, users can choose a file to examine from the list. When they do, the `AfterUpdate` event of the combo box has an event attached to it that uses the `FindExecutable()` API function to locate the executable application for the file, if one exists. Listing 15.4 shows the code for the `cboFileToFindExe_AfterUpdate` sub.

LISTING 15.4 Chap15.mdb: Displaying the Associated File

```

Private Sub cboFileToFindExe_AfterUpdate()
    Dim strFileName As String
    Dim lngReturnVal
    '--Set up the string buffer used for the actual return value.
    Dim EXEPath As String
    EXEPath = String$(255, 0)
    '--Actual API Call
    lngReturnVal = wu_FindExecutable(Me!cboFileToFindExe, _
        strAppPath, EXEPath)
    If lngReturnVal > 32 Then
        Me!txtExecutablePath = Left(EXEPath, InStr(1, EXEPath, Chr(0)) - 1)
        '--Greater than 32 means an exe has been found.
    Else
        Me!txtExecutablePath = "Not Associated with an Application"
    End If
End Sub

```

Here are a couple of main points about the `cboFileToFindExe_AfterUpdate` routine:

- The following lines are quite common for setting up strings used to return a string value from an API call:

```
'--Set up the string buffer used for the actual return value.
Dim EXEPath As String
EXEPath = String$(255, 0)
```

NOTE

You have to pad out a string to a maximum fixed length because API calls can't deal with Access variable-length strings. The API call can't expand a string, so you need to make it the maximum size possible.

- In the following code lines, `Me!cboFileToFindExe` is the combo box value that's a file-name, `strAppPath` is the path of the application that was retrieved in the `ListFiles()` function, and `EXEPath` is the return buffer set up earlier in the function:

```
'--Actual API Call
lngRetVal = wu_FindExecutable(Me!cboFileToFindExe, _
    strAppPath, EXEPath)
```

- The last line trims off the Null value placed at the end of a string retrieved from the API call:

```
Me!txtExecutablePath = Left(EXEPath, InStr(1, EXEPath, Chr(0)) - 1)
```

Figure 15.5 shows the form just after it looks up the executable for `Win32api.txt`.

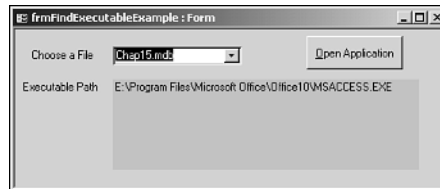


FIGURE 15.5

The `FindExecutable()` API call is useful for locating associated applications.

The other interesting code takes the executable path stored in `txtExecutablePath` and uses `Shell()` to start the application, opening the file chosen in the `cboFileToFindExe` combo box. The code, in Listing 15.5, is attached to the `cmdOpenApplication` command button on the `OnClick` event. This code can be useful when implemented in an environment that you don't know which applications will be needed.

LISTING 15.5 Chap15.mdb: Opening the Requested File

```
Private Sub cmdOpenApplication_Click()  
    Dim lngRetVal As Long  
    '--Test to make sure a file has been chosen  
    If IsNull(Me!cboFileToFindExe) Then  
        MsgBox "No File Chosen!"  
        Exit Sub  
    End If  
    '--Make sure an application is associated  
    If Me!txtExecutablePath = "Not Associated with an Application" Then  
        MsgBox "This file has no application associated with it!"  
        Exit Sub  
    End If  
    '--Call the shell command  
    lngRetVal = Shell(Me!txtExecutablePath & " " & strAppPath & _  
        Me!cboFileToFindExe, 3)  
End Sub
```

CAUTION

If you have multiple installations of Access (Access 2000 versus 2002 or 97), this call won't always give you the desired effect. Test for each type of installation with which you will be using API calls.

Connecting and Disconnecting Network Drives from Within Access

Being able to manipulate the network from within your application gives you just that much more power, so you won't have to rely on outside applications to help control the environment. The following sections demonstrate two ways to connect and disconnect network drives:

- Programmatically connecting and disconnecting network drives directly
- Calling the standard dialogs to connect and disconnect the network drives

Programmatically Connecting and Disconnecting Network Drives Directly

The following API declarations—`WNetAddConnectionA` and `WNetCancelConnectionA`—are dealt with in this section:

```

Declare Function wu_NetAddConnection Lib "mpr" Alias "WNetAddConnectionA" _
    (ByVal NetPath$, ByVal Password$, ByVal LocalDrive$) As Integer
Declare Function wu_NetCancelConnection Lib "mpr" Alias _
    "WNetCancelConnectionA" (ByVal NetPath$, ByVal FileForce%) As Integer

```

This example uses a form named `frmConnectAndDisconnectDriveExample`. In Figure 15.6, see what My Computer shows for existing drive connections.

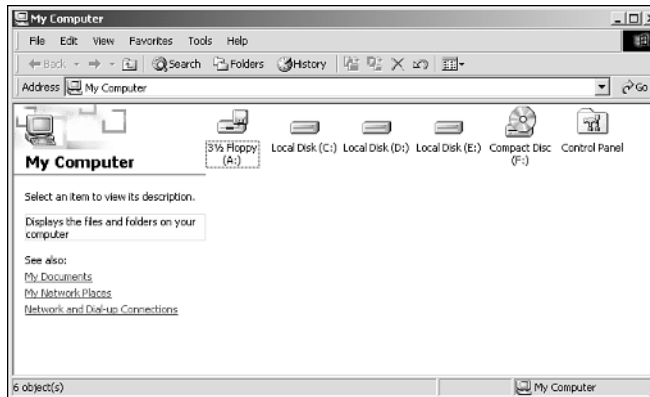


FIGURE 15.6

Notice that no drive G is currently connected.

Open the `frmConnectAndDisconnectDriveExample` form. Next, fill in the Server Path to Connect text box, also called `txtPathToConnect`. Then fill in the Drive to Connect text box, also called `txtDriveToConnect`. Figure 15.7 shows the resulting text in the form.

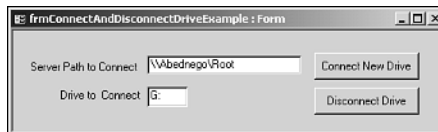
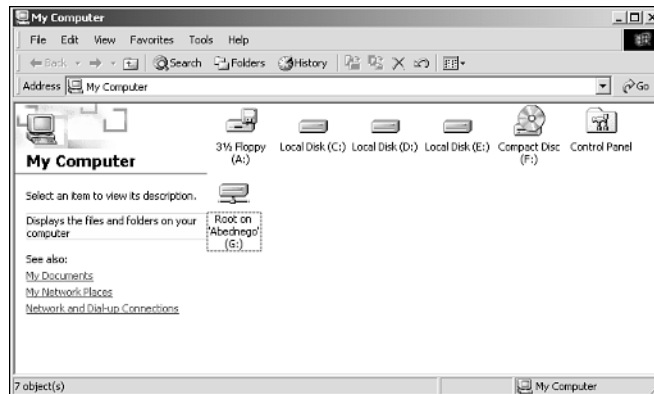


FIGURE 15.7

These two fields will work together to create a new connection.

Click the Connect New Drive command button, otherwise known as `cmdConnectNewDrive`. If the connection is successful, you'll see a message saying so; if not, you'll see an error message. After the connection, My Computer should look like Figure 15.8.

**FIGURE 15.8**

Drive G is now connected and looking fine.

The actual code to perform this magic is a little more than a dozen lines (see Listing 15.6). One API call performs the tough part. The routine, `cmdConnectNewDrive_Click`, is attached to the command button `cmdConnectNewDrive`.

LISTING 15.6 Chap15.mdb: Connecting a Network Drive Programmatically

```
Private Sub cmdConnectNewDrive_Click()
    Dim lngResult As Long
    '--Make sure both fields are there.
    If IsNull(Me!txtPathToConnect) Or IsNull(Me!txtDriveToConnect) Then
        MsgBox "Please supply both a path and a drive!"
        Exit Sub
    End If
    '--Attempt the connection
    lngResult = wu_NetAddConnection(Me!txtPathToConnect, _
        "(enter password here, if required)", Me!txtDriveToConnect)
    If lngResult = 0 Then
        MsgBox "Drive connected successfully!"
    Else
        MsgBox "Error occurred!"
    End If
End Sub
```

As you can see, the workhorse is the line that reads

```
lngResult = wu_NetAddConnection(Me!txtPathToConnect, _
    "(enter password here, if required)", Me!txtDriveToConnect)
```

This routine takes the path and the drive to connect. (The empty quotation marks in the middle are for a password, if required.)

The last part of this code disconnects the drive specified in `txtDriveToConnect`. The interesting part is that it tries to disconnect gently first; if any files are still open, the disconnection will fail. The routine then brings back a message box, asking whether the user wants to strong-arm the file(s) and force the closing. This code, in Listing 15.7, is attached to the `OnClick` event of the `cmdDisconnect` command button.

LISTING 15.7 Chap15.mdb: Disconnecting a Network Drive

```
Private Sub cmdDisconnect_Click()
    Dim lngResult As Long
    '--Try a gentle disconnect first
    If wu_NetCancelConnection(Me!txtDriveToConnect, 0) <> 0 Then
        Beep
        If MsgBox("The Connection could not be disconnected," & _
            "this is probably because you have files open on this drive." & _
            vbCrLf & vbCrLf & "Do you want to force the disconnection with a" & _
            " possible loss of data?", vbYesNo, "Disconnect Error") = vbYes Then
            '--If permission was granted, then force the disconnect
            lngResult = wu_NetCancelConnection(Me!txtDriveToConnect, 1)
        Else '--If no, then leave
            Exit Sub
        End If
    End If
    MsgBox "Drive disconnected successfully!"
End Sub
```

Use these routines when you don't want to have user intervention. They're used on a form here but can be called without users even knowing about them. Generally, if you want users to supply the paths, just call the standard dialogs.

Calling Standard Dialogs to Connect and Disconnect Network Drives

Calling the standard network dialogs gives users a familiar way to choose network drives and paths in which to create connections. The routines to call the `WNetConnectionDialog` and `WNetDisconnectDialog` functions are very straightforward but require some arguments to be passed. You might not want to have to deal with these arguments each time you call because they have no real effect and don't change when you call them. When this is the case, you call them from other routines.

NOTE

Routines such as these are sometimes called *wrappers* because they're wrapped around the function and set up variables within them. You can then call them fairly easily from anywhere in the application.

To simplify things even further, however, I've created some wrappers for the API calls. The declarations for the two routines are as follows:

```
Declare Function wu_WNetConnectionDialog Lib "mpr" Alias _
    "WNetConnectionDialog" (ByVal hwnd As Long, ByVal dwType As Long) _
    As Long
Declare Function wu_WNetDisconnectDialog Lib "mpr" Alias _
    "WNetDisconnectDialog" (ByVal hwnd As Long, ByVal dwType As Long) As Long
```

The form used for these examples is frmConnectAndDisconnectDialogsExample. Figure 15.9 shows this form with the connect dialog (Map Network Drive) displayed by clicking the command button labeled Call Network Connect Dialog.

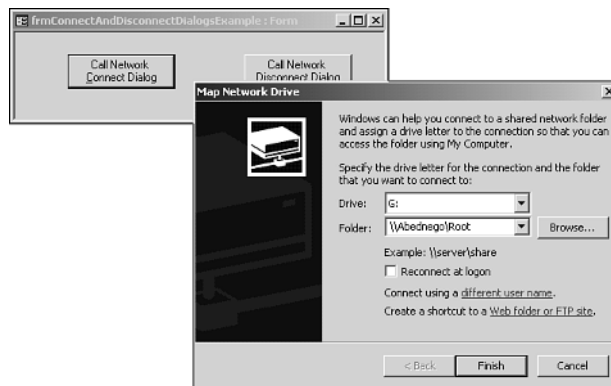


FIGURE 15.9

The wrapper function is called directly from a command button event on this form.

ap_CallNetworkConnectDialog() and ap_CallNetworkDisconnectDialog(), the wrapper functions used, are placed in the OnClick event of each command button. Listing 15.8 shows the code for ap_CallNetworkConnectDialog().

LISTING 15.8 Chap15.mdb: Calling the Network Connect Dialog

```
Public Function ap_CallNetworkConnectDialog()  
    Dim lngDummy As Long  
    Dim lngHWind As Long  
    lngDummy = wu_WNetConnectionDialog(lngHWind, 1)  
End Function
```

Pretty extensive, isn't it? As mentioned before, this call doesn't even necessarily require a wrapper function. The function `ap_CallNetworkDisconnectDialog()` isn't really any more difficult, as you can see in Listing 15.9.

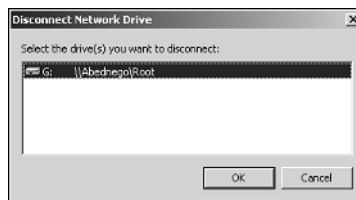
LISTING 15.9 Chap15.mdb: Calling the Network Disconnect Dialog

```
Function ap_CallNetworkDisconnectDialog()  
    Dim lngDummy As Long  
    Dim lngHWind As Long  
    lngDummy = wu_WNetDisconnectDialog(lngHWind, 1)  
End Function
```

NOTE

The return value of both API calls isn't used for these examples. If successful in their tasks, the routines return a 0; anything else means that a problem occurred or Cancel was chosen. Different values are returned for various situations, and you need to play with them to determine which to trap.

Figure 15.10 shows the dialog opened by clicking the Call Network Disconnect Dialog button.

**FIGURE 15.10**

This is just one of many system utilities that you can include in your applications.

There are additional networking API calls as well. The examples shown here have proven to be the most valuable to me.

Displaying the Current User and Computer Name

The next two API routines, `GetUserNameA` and `GetComputerNameA`, return the current user and computer name. Here are the declarations for these routines:

```
Declare Function wu_GetUserName Lib "advapi32.dll" Alias "GetUserNameA" _
    (ByVal lpBuffer As String, nSize As Long) As Long
Declare Function wu_GetComputerName Lib "kernel32" Alias _
    "GetComputerNameA" (ByVal lpBuffer As String, nSize As Long) As Long
```

These routines come in handy for creating a logon screen for your application that automatically polls who the user is from the system. You can then pass the information on when dealing with ODBC queries, as well as when dealing with Access's own security.

The form, `frmGetUserAndComputerNameExample`, is located in the `Chap15.mdb` database. It has two text boxes, `txtUserName` and `txtComputerName`. As you can see in Figure 15.11, the actual API calls are called from within other wrapper functions. Listing 15.10 shows the code for `ap_GetUserName()`.

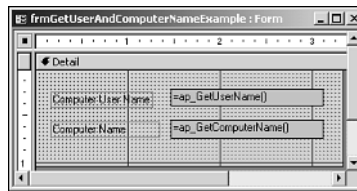


FIGURE 15.11

The actual calls can be seen here in the `ControlSource` properties of these text boxes.

LISTING 15.10 Chap15.mdb: Retrieving the User's Name

```
Function ap_GetUserName() As Variant
    Dim strUserName As String
    Dim lngLength As Long
    Dim lngResult As Long
    '--Set up the buffer
    strUserName = String$(255, 0)
    lngLength = 255
    '--Make the call
    lngResult = wu_GetUserName(strUserName, lngLength)
    '--Assign the value
    ap_GetUserName = strUserName
End Function
```

The code for `ap_GetComputerName()` is about as “hard,” as Listing 15.11 shows.

LISTING 15.11 Chap15.mdb: Retrieving the Computer’s Name

```
Function ap_GetComputerName() As Variant
    Dim strComputerName As String
    Dim lngLength As Long
    Dim lngResult As Long
    '--Set up buffer.
    strComputerName = String$(255, 0)
    lngLength = 255
    '--Make the call.
    lngResult = wu_GetComputerName(strComputerName, lngLength)
    '--Clean up and assign the value.
    ap_GetComputerName = Left(strComputerName, InStr(1, strComputerName, _
        Chr(0)) - 1)
End Function
```

Figure 15.12 shows the form in Run mode, with the user and computer name showing.

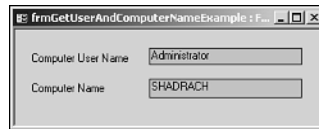


FIGURE 15.12

Grabbing the user and computer are just two of hundreds of system operations you can perform.

The next set of API calls displays additional information about the computer you’re using. These calls display various paths of the three most important folders used by most Windows setups: the Windows, Windows System, and Temp folders.

Displaying Pertinent Folders from Within Your Application

The following are some useful function calls to know when you’re in the middle of your application, and the other API calls listed here are pretty straightforward. Three routines—`GetWindowsDirectoryA`, `GetSystemDirectoryA`, and `GetTempPathA`—are called. You can see which routine performs what function just by looking at the declarations:

```
Declare Function wu_GetWindowsDirectory Lib "kernel32" Alias _
    "GetWindowsDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long
```

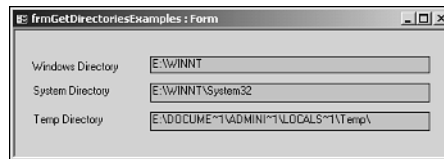

PART III

```

Declare Function wu_GetSystemDirectory Lib "kernel32" Alias _
    "GetSystemDirectoryA" (ByVal lpBuffer As String, _
        ByVal nSize As Long) As Long
Declare Function wu_GetTempPath Lib "kernel32" Alias "GetTempPathA" _
    (ByVal nBufferLength As Long, ByVal lpBuffer As String) As Long

```

As with the other examples, a form displays the results. Figure 15.13 shows this form, frmGetDirectoriesExamples, in action. Listing 15.12 shows the code for each routine.

**FIGURE 15.13**

These are three of the most important folders on your system.

NOTE

Notice that the Temp folder includes an extra backslash at the end. I believe this was not intended originally to follow the other two API calls. This is also obvious from the difference in the routine names. Whereas the first two have the word *Directory* on the end, the Temp routine has the word *Path*. Use the `Left()` function to trim off the last backslash.

LISTING 15.12 Chap15.mdb: Routines for Viewing Three Different Folders

```

Function ap_GetWindowsDir() As Variant
    Dim strWindowsDir As String
    Dim lngLength As Long
    Dim lngResult As Long
    '--Set up buffer.
    strWindowsDir = String$(255, 0)
    lngLength = 255
    '--Make the call.
    lngResult = wu_GetWindowsDirectory(strWindowsDir, lngLength)
    '--Clean up and assign the value.
    ap_GetWindowsDir = Left(strWindowsDir, InStr(1, strWindowsDir, Chr(0)) - 1)
End Function

```

LISTING 15.12 Continued

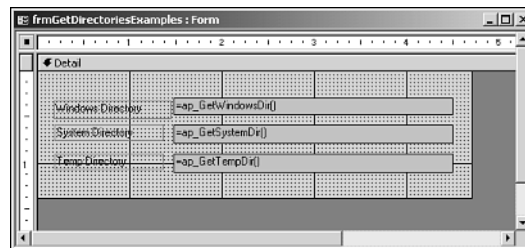
```

Function ap_GetSystemDir() As Variant
    Dim strSystemDir As String
    Dim lngLength As Long
    Dim lngResult As Long
    '--Set up buffer.
    strSystemDir = String$(255, 0)
    lngLength = 255
    '--Make the call.
    lngResult = wu_GetSystemDirectory(strSystemDir, lngLength)
    '--Clean up and assign the value.
    ap_GetSystemDir = Left(strSystemDir, InStr(1, strSystemDir, Chr(0)) - 1)
End Function

Function ap_GetTempDir() As Variant
    Dim strTempDir As String
    Dim lngLength As Long
    Dim lngResult As Long
    '--Set up buffer.
    strTempDir = String$(255, 0)
    lngLength = 255
    '--Make the call.
    lngResult = wu_GetTempPath(lngLength, strTempDir)
    '--Clean up and assign the value.
    ap_GetTempDir = Left(strTempDir, InStr(1, strTempDir, Chr(0)) - 1)
End Function

```

These routines basically follow the same pattern set for making API calls. You can see how these routines are called by looking at the frmGetDirectoriesExamples form in Figure 15.14.

**FIGURE 15.14**

Finding out where to look for these routines is as easy as looking at the text boxes' ControlSource properties.

The toughest part of using API routines is figuring out which routine performs what action, especially because the routine names aren't all that descriptive. Some careful sleuthing can

help you out in that regard if you look through the resources listed earlier in the section “Finding API Declarations.”

Using the Open File Dialog API Call

The Microsoft Office Developer tools include an ActiveX control called Common Dialogs. But if you’re like me, you would rather not use ActiveX controls in your application and pull in the overhead. All the dialogs available in the ActiveX Common Dialog methods have a comparable API call. The API call covered here is the Open File dialog. Some of the other API calls are in Listing 15.13.

LISTING 15.13 Win32API.txt: API Declarations for Equivalent Common Dialog ActiveX Methods

```

Declare Function GetOpenFileName Lib "comdlg32.dll" Alias _
    "GetOpenFileNameA" (pOpenfilename As OPENFILENAME) As Long
Declare Function GetSaveFileName Lib "comdlg32.dll" Alias _
    "GetSaveFileNameA" (pOpenfilename As OPENFILENAME) As Long
Declare Function ChooseColor Lib "comdlg32.dll" Alias _
    "ChooseColorA" (pChoosecolor As CHOOSECOLOR) As Long
Declare Function ChooseFont Lib "comdlg32.dll" Alias _
    "ChooseFontA" (pChoosefont As CHOOSEFONT) As Long
Declare Function PrintDlg Lib "comdlg32.dll" Alias _
    "PrintDlgA" (pPrintdlg As PRINTDLG) As Long
Declare Function PageSetupDlg Lib "comdlg32.dll" Alias _
    "PageSetupDlgA" (pPagesetupdlg As PAGESETUPDLG) As Long

```

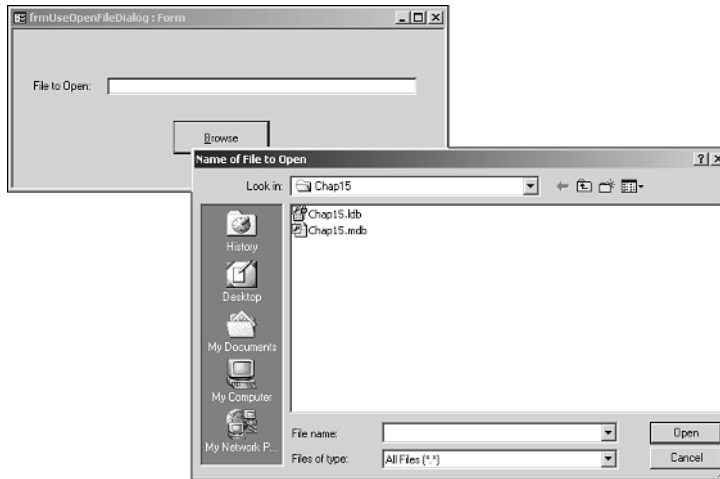
You can use the Open File dialog in just about every application written when front- and back-end scenarios are used. Primarily, it’s used to let users locate the back end if the links are broken to the tables. If you want to look at this case, check out Chapter 25, “Startup Checking System Routines Using ADO.” Figure 15.15 shows the dialog in action. This figure also shows the calling form `frmUseOpenFileDialog`, which is located in `Chap15.mdb` on this book’s Web page at www.sampublishing.com.

The event procedure call to the API wrapper requires few lines of code, attached to the command button `cmdOpenFile`:

```

Private Sub cmdOpenFile_Click()
    If IsNull(Me!txtFileToOpen) Then
        Me!txtFileToOpen = ap_OpenFile()
    Else
        Me!txtFileToOpen = ap_OpenFile(Me!txtFileToOpen)
    End If
End Sub

```

**FIGURE 15.15**

This API call is necessary for most applications you write.

Listing 15.14 shows the code for declaring the API, along with creating a custom type structure to pass to the API call. This code is located in the `modFileOpenRoutine` module, also in `Chap15.mdb`. Next are some constants used for the common dialog routines. Last is the wrapper function itself, `ap_OpenFile()`.

LISTING 15.14 Chap15.mdb: Getting a Filename Through the Open File Dialog

```
Option Compare Text
Option Explicit
Private Declare Function wu_GetOpenFileName Lib "comdlg32.dll" _
    Alias "GetOpenFileNameA" (pOpenfilename As OPENFILENAME) As Long

Private Type OPENFILENAME
    lStructSize As Long
    hwndOwner As Long
    hInstance As Long
    lpstrFilter As String
    lpstrCustomFilter As String
    nMaxCustFilter As Long
    nFilterIndex As Long
    lpstrFile As String
    nMaxFile As Long
    lpstrFileTitle As String
    nMaxFileTitle As Long
```

LISTING 15.14 Continued

```

    lpstrInitialDir As String
    lpstrTitle As String
    flags As Long
    nFileOffset As Integer
    nFileExtension As Integer
    lpstrDefExt As String
    lCustData As Long
    lpfnHook As Long
    lpTemplateName As String
End Type

Const cdIOFNAllowMultiselect = &H200
Const cdIOFNCreatePrompt = &H2000
Const cdIOFNExplorer = &H80000
Const cdIOFNExtensionDifferent = &H400
Const cdIOFNFileMustExist = &H1000
Const cdIOFNHelpButton = &H10
Const cdIOFNHideReadOnly = &H4
Const cdIOFNLongNames = &H200000
Const cdIOFNNoChangeDir = &H8
Const CdIOFNNoDereferenceLinks = &H100000
Const cdIOFNNoLongNames = &H40000
Const CdIOFNNoReadOnlyReturn = &H8000
Const cdIOFNNoValidate = &H100
Const cdIOFNOverwritePrompt = &H2
Const cdIOFNPathMustExist = &H800
Const cdIOFNReadOnly = &H1
Const CdIOFNShareAware = &H4000

Public Function ap_OpenFile(Optional ByVal strFileNameIn As String = "", _
    Optional strDialogTitle As String = "Name of File to Open")
    Dim lngReturn As Long
    Dim intLocNull As Integer
    Dim strTemp As String
    Dim ofnFileInfo As OPENFILENAME
    Dim strInitialDir As String
    Dim strFileName As String

    '--if a file path passed in with the name, parse it and split it off.
    If InStr(strFileNameIn, "\") <> 0 Then
        strInitialDir = Left(strFileNameIn, InStrRev(strFileNameIn, "\"))
        strFileName = Left(Mid$(strFileNameIn, _
            InStrRev(strFileNameIn, "\" ) + 1) & String(256, 0), 256)
    
```

LISTING 15.14 Continued

```

Else
    strInitialDir = CurrentProject.Path
    strFileName = Left(strFileNameIn & String(256, 0), 256)
End If

With ofnFileInfo
    .lStructSize = Len(ofnFileInfo)
    .lpstrFile = strFileName
    .lpstrFileTitle = String(256, 0)
    .lpstrInitialDir = strInitialDir
    .hwndOwner = Application.hWndAccessApp
    .lpstrFilter = "All Files (*.*)" & Chr(0) & "*.*" & Chr(0)
    .nFilterIndex = 1
    .nMaxFile = Len(strFileName)
    .nMaxFileTitle = ofnFileInfo.nMaxFile
    .lpstrTitle = strDialogTitle
    .flags = cd10FNFileMustExist Or cd10FNHideReadOnly Or cd10FNNoChangeDir
    .hInstance = 0
    .lpstrCustomFilter = String(255, 0)
    .nMaxCustFilter = 255
    .lpfnHook = 0
End With

lngReturn = wu_GetOpenFileName(ofnFileInfo)
If lngReturn = 0 Then
    strTemp = ""
Else
    '-Trim off any null string
    strTemp = Trim(ofnFileInfo.lpstrFile)
    intLocNull = InStr(strTemp, Chr(0))
    If intLocNull Then
        strTemp = Left(strTemp, intLocNull - 1)
    End If
End If

ap_OpenFile = strTemp

End Function

```

You can either supply a file path to start with, or leave the text box blank. Another enhancement you can use is copying the `ap_OpenFile` routine, and then replacing the `wu_GetOpenFileName` call with one for `wu_GetSaveFileName`. Don't forget to declare the function as shown here:

```
Private Declare Function wu_GetSaveFileName Lib "comdlg32.dll" Alias _  
    "GetSaveFileNameA" (pOpenfilename As OPENFILENAME) As Long
```

Summary

As powerful as Access is with VBA as its native language, limitations exist. Although Microsoft has been overcoming those limitations with every version of the product the company puts out, shortcomings still can be seen. API calls are a way to handle those shortcomings.

Dynamic link libraries allow you to call routines from sources other than Access, and they don't have to be recompiled when the application is. You can use the routines from these libraries in ways that are totally invisible to users, although some issues need to be dealt with concerning the understanding of built-in and user-defined variables.

- Chapter 2, "Coding in Access 2002 with VBA," covers in detail how arguments are passed, what user-defined variables are, and what different data types are used in VBA.
- Chapter 13, "Driving Office Applications with Automation," shows how to extend the power of Access by controlling other Office applications using Automation through VBA.
- Chapter 18, "Manipulating the Registry with VBA," covers using API calls to work with the Registry.

Extending Your VBA Library Power with Class Modules and Collections

CHAPTER

16

IN THIS CHAPTER

- **Setting Up a Bookmark Tracker** 492
- **Managing Multiple Instances of the Same Form** 506

Class modules and collections were first introduced in Chapter 2, “Coding in Access 2002 with VBA.” That chapter barely scratches the surface of using class modules and collections, which is as it should be because you should walk before you run.

When Access 95 first included class modules, developers were hard-pressed to find uses for them. After VB developers started providing ideas, it became hard not to see class modules as a possible solution for most coding issues. They allow you to create objects that you control, including their own custom properties and methods. You can package mundane tasks so that you can use them over and over to cut down coding time.

Collections, which help you manage your environment, are much more intuitive to use than arrays. If you know how to use Access collections (and you should by now), creating your own is easy.

This chapter delves quite a bit more into creating custom collections and class modules. It also looks at manipulating bookmarks with custom collections and class modules. You will also see how to use a custom collection to track multiple instances of the same form.

Setting Up a Bookmark Tracker

In many instances, clients want to track multiple records during a session and be able to switch between them. Access provides the recordset Bookmark property to mark and return to records. This property is simple to take advantage of, but somewhat limited.

A more advanced example is a student database. School administrators might want to work on a few student records at once and be able to switch between them. They would then want to keep track of their bookmarks rather than sequentially move through the recordset each time.

Another example for tracking multiple bookmarks is an employee database. If human resources personnel were assigned various employees to work with, or needed to look back and forth between multiple people, it would be convenient to have the list with the records in them, and be able to switch with a mouse click. Figure 16.1 shows this example, where you can switch between multiple records by using a combo box.

This chapter focuses on this previous scenario. Because every utility worth its salt needs a name, let's call this one the Bookmark Tracker. The Bookmark Tracker and all its objects can be found on this book's Web page at www.sampublishing.com.

Feature Set of the Bookmark Tracker

Before looking at the elements that make up the Bookmark Tracker, you need to understand how useful the Bookmark Tracker can be. See what features are included:

- You can add bookmarks just by moving to the record you want to bookmark, and then selecting << Add a New Bookmark >> (refer to Figure 16.1).

- You can delete bookmarks by selecting << Remove Bookmark(s) >> from the combo box in Figure 16.1. You then see the dialog shown in Figure 16.2.
- You can move to a stored bookmark by choosing a bookmark in the list.

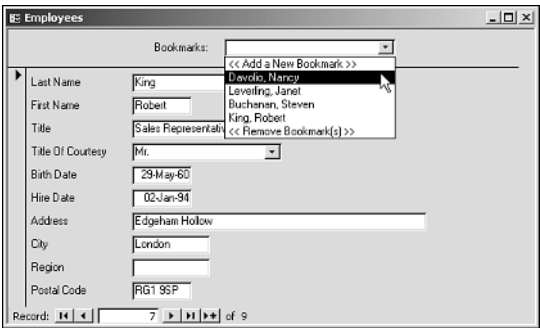


FIGURE 16.1
This combo box lets you add, remove, and move between multiple bookmarks.

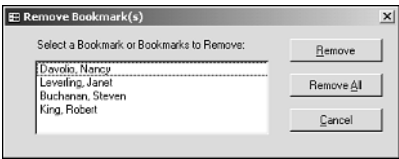


FIGURE 16.2
With this dialog, you can pick one or more bookmarks to remove, or all.

Basic Objects of the Bookmark Tracker

First look at the `tblEmployees` table (in Figure 16.3), which contains standard employee information.

tblEmployees : Table							
Employee ID	Last Name	First Name	Title	Title Of Courtesy	Birth Date	Hire Date	
1	Davolio	Nancy	Sales Representative	Ms.	08-Dec-48	01-May-92	
2	Fuller	Andrew	Vice President, Sales	Dr.	19-Feb-52	14-Aug-92	
3	Leverling	Janet	Sales Representative	Ms.	30-Aug-63	01-Apr-92	
4	Peacock	Margaret	Sales Representative	Mrs.	19-Sep-37	03-May-93	
5	Buchanan	Steven	Sales Manager	Mr.	04-Mar-65	17-Oct-93	
6	Suyama	Michael	Sales Representative	Mr.	02-Jul-63	17-Oct-93	
7	King	Robert	Sales Representative	Mr.	29-May-60	02-Jan-94	
8	Callahan	Laura	Inside Sales Coordinator	Ms.	09-Jan-68	05-Mar-94	
9	Dodsworth	Anne	Sales Representative	Ms.	27-Jan-66	15-Nov-94	

FIGURE 16.3
The `tblEmployees` table is the record source for the `frmBookmarkTrackerExample` form.

Next, look at the `frmBookmarkTrackerExample` form, shown in Form view in Figure 16.1. This form is pretty standard except for a few code lines and a combo box labeled Bookmarks. You'll look at the code lines after you're introduced to the main players—the class modules. These class modules are discussed in the next couple of sections, and then run through for each task performed.

Before getting to the class modules, collections, and bookmarks, consider the `modFormUtilities` module, which uses the following function. The `ap_FormIsOpen()` function is used within the `clsBookmarkManagement` class module, as discussed in the section “`clsBookmarkManagement` Performs the Hard Work”:

```
Public Function ap_FormIsOpen(strFormName As String) As Boolean
    ap_FormIsOpen = Application.CurrentProject.AllForms (strFormName).IsLoaded
End Function
```

NOTE

The `Application` object is optional in this statement. Both the `CurrentProject` object and the `AllForms` collection were new to Access 2000. For more techniques on taking advantage of both items, see Chapter 4, “Working with Access Collections and Objects.”

Let's Have a Little Class...Modules, That Is

The Bookmark Tracker uses two class modules, as seen in the Database window in Figure 16.4.

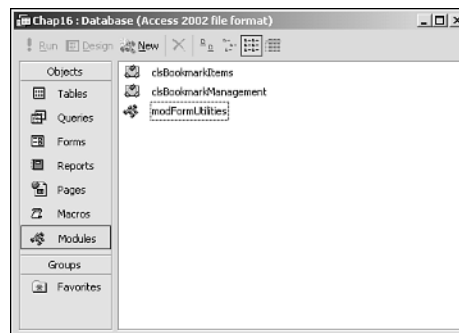


FIGURE 16.4

Notice the difference in icons between standard modules (`modFormUtilities`) and class modules.

clsBookmarkItems Stores the Actual Bookmarks

Remember that class modules are custom objects you can create that consist of properties and/or methods you specify. The first class module, `clsBookmarkItems` (in Listing 16.1), specifies two properties with no methods. `clsBookmarkItems` tracks individual entries of the stored bookmarks. Each stored bookmark has this class object entered into a collection.

LISTING 16.1 Chap16.mdb: Storing Information for Individual Bookmarks

```
Option Compare Database
Option Explicit

Private strBookmark As String
Private strDescription As String

Property Get ActualBM() As String
    ActualBM = strBookmark
End Property

Property Let ActualBM(strCurrBookmark As String)
    strBookmark = strCurrBookmark
End Property

Property Get BMDescription() As String
    BMDescription = strDescription
End Property

Property Let BMDescription(strCurrDescription As String)
    strDescription = strCurrDescription
End Property
```

TIP

Class modules can expand to meet your programming needs. If you use the code in Listing 16.1 for yourself, you can add properties and methods as needed in the future.

The first property, `ActualBM`, stores a bookmark value. The second property, `BMDescription`, stores a specified description for the bookmark entry, which is displayed in the combo box. You will see more of these as you walk through the code for the other class module.

clsBookmarkManagement Performs the Hard Work

The second class module, clsBookmarkManagement, consists of two methods and one support subroutine. The two methods are explained here:

- InitBookmarks initializes the Bookmark Tracker.
- BookmarkAction actually performs the various tasks needed.

NOTE

You might be asking yourself why a separate initialization method is needed when you have the Class_Initialize method, which you can use with the class modules you create. InitBookmarks uses some passed arguments. You can't pass values to the built-in method.

The Calls to the Class Module Methods

The best way to understand the methods is to see them called from the form (see Listing 16.2).

LISTING 16.2 Chap16.mdb: Evoking the clsBookmarkManagement Methods

```
'-- Declare a clsBookmarkManagement variable.
Private bmtEmployeeForm As clsBookmarkManagement
Private Sub Form_Load()
    '-- Create an instance of the clsBookmarkManagement class
    Set bmtEmployeeForm = New clsBookmarkManagement
    '-- Call the custom initialization routine, passing the necessary items
    bmtEmployeeForm.InitBookmarks Me!cboBookMark, "LastName", "FirstName"
End Sub

Private Sub cboBookMark_AfterUpdate()
    '-- Call the BookmarkAction method, for all actions.
    bmtEmployeeForm.BookmarkAction
End Sub
```

Now, see what occurs in the frmBookmarkTrackerExample form module:

1. Starting with the Declarations section, you can see the variable being declared as a clsBookmarkManagement class:
`Private bmtEmployeeForm As clsBookmarkManagement`
2. The variable bmtEmployeeForm is instantiated with the following line in the Load event:
`Set bmtEmployeeForm = New clsBookmarkManagement`

3. Next comes the call to the initialization method created (InitBookmarks):

```
bmtEmployeeForm.InitBookmarks Me!cboBookMark, "LastName",  
"FirstName"
```

The InitBookmarks method takes up to three parameters:

- The combo box that's used on the form for the bookmarks
- The first part of the description displayed in the combo box
- The second part of the description displayed in the combo box (optional)

You'll look at this method in detail in a moment.

4. Then comes the call to the BookmarkAction method, which performs the actual work during the course of working with the form open:

```
Private Sub cboBookMark_AfterUpdate()  
    '-- Call the BookmarkAction method, for all actions.  
    bmtEmployeeForm.BookmarkAction  
End Sub
```

Notice that this method is called from the combo box's AfterUpdate event.

That's it for the code behind the form. When you want to use the Bookmark Tracker on another form, just copy the combo box and the few lines of code just discussed.

TIP

Check out Chapter 17, "Creating Your Own Wizards and Add-Ins," which offers an add-in for adding the Bookmark Tracker to forms, without any additional coding.

The Declarations Section for the Class Module

Before looking at the InitBookmarks method, examine the clsBookmarkManagement class module's Declarations area. These variables and constants are pretty self-explanatory and are used throughout the class module:

```
Option Compare Database  
Option Explicit
```

```
Private frmBookmark As Form  
Private cboBookMark As ComboBox  
Private strBookMarkDesc1 As String  
Private strBookMarkDesc2 As String  
Private colBookMarks As Collection
```

PART III

```
Const conQuotes = """"
Const apAddNewBM = -1
Const apRemoveBM = -2
```

The InitBookmarks Method

Look now at InitBookmarks, one of two methods found in the clsBookmarkManagement module (see Listing 16.3).

LISTING 16.3 Chap16.mdb: Setting Up the Bookmark Tracker

```
Public Sub InitBookmarks(cboCurrent As ComboBox, str1stDesc as String, _
    Optional str2ndDesc As String = "")
    '-- Store the form and combo box that is utilizing this class
    Set frmBookmark = cboCurrent.Parent
    Set cboBookMark = cboCurrent
    '-- Instantiate the collection that will store the bookmarks
    Set colBookMarks = New Collection
    '-- Set up the combo box to handle manipulating the bookmarks
    cboBookMark.ColumnCount = 2
    cboBookMark.ColumnWidths = "0;2"
    cboBookMark.RowSourceType = "Value list"
    cboBookMark.RowSource = "-1; '<< Add a New Bookmark >>' "
    '-- Create the mask that will be used to display the description
    strBookMarkDesc1 = str1stDesc
    If len(str2ndDesc) > 0 Then
        strBookMarkDesc2 = str2ndDesc
    End If
End Sub
```

Actually, the steps from this method can be listed based on the comments in the code itself. Any comment that needs further explanation will have it.

1. Store the form and combo box that's using this class. Based on the combo box passed to the method, these control references are used throughout the class:

```
Set frmBookmark = cboCurrent.Parent
Set cboBookMark = cboCurrent
```

2. Instantiate the collection that will store the bookmarks:

```
Set colBookMarks = New Collection
```

3. Set up the combo box to handle bookmark manipulation:

```
cboBookMark.ColumnCount = 2
cboBookMark.ColumnWidths = "0;2"
cboBookMark.RowSourceType = "Value list"
cboBookMark.RowSource = "-1; '<< Add a New Bookmark >>' "
```

This combo box also adds the first value, which consists of a –1 and the << Add a New Bookmark >> description.

NOTE

If you plan to re-create the combo box from scratch each time, leave this code the way it is. If you just want to copy the same combo box to another form, place the values in these code lines into the actual combo box.

4. Create the mask that will display the description. This part of the code allows you to specify up to two fields to use as a description for each bookmark stored. In the case of the employees, it's the fields Last Name and First Name. For companies, it might just be one field, such as Company Name.

```
strBookMarkDesc1 = str1stDesc
If Len(str2ndDesc) > 0 Then
    strBookMarkDesc2 = str2ndDesc
End If
```

After the load event executes and the form opens, clicking the combo box will show something similar to Figure 16.5. As soon as users click the combo box and select << Add a New Bookmark >>, the combo box calls the `BookmarkAction` method, which stores the bookmark in the collection. It then rebuilds the combo box `RowSource` property and displays a message that the bookmark is recorded. If the bookmark is the first to be added to the list, another choice is added—<< Remove Bookmark(s) >>—in the `RebuildBookmarkCombo` routine later in Listing 16.5.

The screenshot shows a form titled "Employees" with a "Bookmarks:" label and a dropdown menu. The dropdown menu is open, showing the option "<< Add a New Bookmark >>". Below the dropdown, there are several text boxes for employee information: Last Name (Davolio), First Name (Nancy), Title (Sales Representative), Title Of Courtesy (Ms.), Birth Date (08-Dec-48), Hire Date (01-May-92), Address (507 20th Ave. E.), City (Seattle), Region (WA), and Postal Code (98122). At the bottom, there is a "Records:" label and a status bar showing "1 of 9".

FIGURE 16.5

This form has just opened with no bookmarks recorded yet.

The BookmarkAction Method

This method is actually the most code for this example (see Listing 16.4).

LISTING 16.4 Chap16.mdb: Performing Most of the Work

```
Public Sub BookmarkAction()
    Dim clsCurrBM As clsBookmarkItems
    Dim strFullDesc As String
    Dim intCurrItem As Integer

    '-- Based on the currently selected item in the combo
    Select Case cboBookmark
        '-- Add a new bookmark
        Case apAddNewBM
            '-- Create the current description
            strFullDesc = frmBookmark(strBookmarkDesc1)
            If Len(strBookmarkDesc2) <> 0 Then
                strFullDesc = strFullDesc & ", " & frmBookmark(strBookmarkDesc2)
            End If
            '-- Check to see if the bookmark already is stored in the collection
            For intCurrItem = 1 To colBookMarks.Count
                Set clsCurrBM = colBookMarks.Item(intCurrItem)
                '-- Compare the current description to the current
                ' bookmark item description.
                If clsCurrBM.BMDescription = strFullDesc Then
                    MsgBox "This bookmark has already been recorded!", _
                        vbCritical, "Bookmark Already Exists"
                    Exit Sub
                End If
            Next
            '-- Add the current description and bookmark to the current
            ' bookmark items object
            Set clsCurrBM = New clsBookmarkItems
            clsCurrBM.BMDescription = strFullDesc
            clsCurrBM.ActualBM = frmBookmark.Bookmark
            '-- Add the current bookmark items object to the collection
            colBookMarks.Add clsCurrBM
            '-- Rebuild the combo box
            RebuildBookmarkCombo
            MsgBox "The bookmark has been added.", vbInformation, "Bookmark Added"

        Case apRemoveBM
            Dim strRowSource As String
            '-- Create the rowsource for the list box on the remove BM dialog
```

LISTING 16.4 Continued

```

For intCurrItem = 1 To colBookMarks.Count
    Set clsCurrBM = colBookMarks.Item(intCurrItem)
    strRowSource = strRowSource & intCurrItem & ";" & _
        & conQuotes & clsCurrBM.BMDescription & conQuotes
    If intCurrItem < colBookMarks.Count Then
        strRowSource = strRowSource & ";"
    End If
Next
'-- Open the remove BM dialog, passing the string just created.
DoCmd.OpenForm "apRemoveBookmarks", WindowMode:=acDialog, _
    OpenArgs:=strRowSource
Dim varCurrItem As Variant
'-- If the form is still open (no error occurred) use it.
If apFormIsOpen("apRemoveBookmarks") Then
    Dim intNumRemoved As Integer
    intNumRemoved = 0
    '-- For any of the selected bookmarks to remove, do it.
    For Each varCurrItem In _
        Forms!apRemoveBookmarks!lboBookmarks.ItemsSelected
        colBookMarks.Remove varCurrItem + 1 - intNumRemoved
        intNumRemoved = intNumRemoved + 1
    Next varCurrItem
    DoCmd.Close acForm, "apRemoveBookmarks"
End If
'-- Rebuild the combo box and requery
RebuildBookmarkCombo

Case Else
    '-- Move the specified bookmark
    Set clsCurrBM = colBookMarks.Item(CInt(cboBookMark))
    frmBookmark.Bookmark = clsCurrBM.ActualBM
End Select
End Sub

```

Because this code is documented, it should be easy to follow. Nevertheless, let's analyze this code step by step.

Adding a Bookmark

The code to adding a bookmark starts in the `Select Case` statement with the line that reads

```
Case apAddNewBM
```

The comments in this section of code list out as follows, with further explanation:

1. Create the current description. This takes the control names specified in the `InitBookmarks` method and creates the actual description displayed in the combo box.
2. Check to see whether the bookmark already is stored in the collection. This is performed by cycling through the collection storing the `clsBookmarkItem` objects, creating a `clsBookmarkItem` variable called `clsCurrBM` to assign each collection item to, and then comparing it to the description of the current item possibly being added to the list:


```
For intCurrItem = 1 To colBookMarks.Count
    Set clsCurrBM = colBookMarks.Item(intCurrItem)
    '-- Compare the current description to the current
    ' bookmark item description.
    If clsCurrBM.BMDescription = strFullDesc Then
```
3. Add the current description and bookmark to the current bookmark items object. This not only shows how to deal with class module objects, but also stores a bookmark into the `ActualBM` property of the `clsCurrBM` object:


```
Set clsCurrBM = New clsBookmarkItems
clsCurrBM.BMDescription = strFullDesc
clsCurrBM.ActualBM = frmBookmark.Bookmark
```
4. Add the current bookmark items object to the collection:


```
colBookMarks.Add clsCurrBM
```
5. Rebuild the combo box with a call to the private subroutine `RebuildBookmarkCombo` (see Listing 16.5).

LISTING 16.5 Chap16.mdb: Re-creating the Combo Box from the `colBookmarks` Collection

```
Private Sub RebuildBookmarkCombo()
    Dim intCurrItem As Integer
    Dim strRowSource As String
    Dim clsCurrBM As clsBookmarkItems
    '-- Re-create the combo box on the form, adding the add feature, then
    ' going through the collection, and adding the descriptions.
    strRowSource = "-1; '<< Add a New Bookmark >>' "
    For intCurrItem = 1 To colBookMarks.Count
        Set clsCurrBM = colBookMarks.Item(intCurrItem)
        strRowSource = strRowSource & ";" & intCurrItem & _
            "&conQuotes & clsCurrBM.BMDescription & conQuotes
    Next
```

LISTING 16.5 Continued

```
'-- Lastly, add the remove bookmark choice if any exist.
If colBookMarks.Count > 0 Then
    strRowSource = strRowSource & ";-2; '<< Remove Bookmark(s) >>' "
End If

'-- Clean up the combo box.
cboBookMark.RowSource = strRowSource
cboBookMark = Null
cboBookMark.Requery
End Sub
```

Again, this routine is pretty well commented. Basically, it adds the << Add a New Bookmark >> selection to the top of the list, goes through the collection to add the descriptions, and then, if at least one entry is in the collection, adds the << Remove Bookmarks(s) >> entry to the list. The last tasks that occur are to reset the RowSource for the combo box, clear the combo by setting it to Null, and then requery it.

Removing a Bookmark

As with adding a bookmark, let's walk through the code step by step, starting with the line of code in Listing 16.4 that reads

Case apRemoveBM

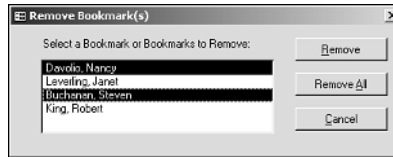
1. Create the row source for the list box on the Remove Bookmark(s) dialog. This is similar to creating the row source for the main combo box in the RebuildBookmarkCombo subroutine in Listing 16.5.

```
For intCurrItem = 1 To colBookMarks.Count
    Set clsCurrBM = colBookMarks.Item(intCurrItem)
    strRowSource = strRowSource & intCurrItem & _
        ";" & conQuotes & clsCurrBM.BMDescription & conQuotes
    If intCurrItem < colBookMarks.Count Then
        strRowSource = strRowSource & ";"
    End If
Next
```

2. Open the Remove Bookmark(s) dialog, passing the string just created. As mentioned, the next step is to open the form used for removing bookmarks. The strRowSource string variable is passed by using the OpenArgs property:

```
DoCmd.OpenForm "apRemoveBookmarks", WindowMode:=acDialog, _
    OpenArgs:=strRowSource
```

Figure 16.6 shows the form apRemoveBookmarks; Listing 16.6 shows the VBA code for each command button on apRemoveBookmarks.

**FIGURE 16.6**

You can delete individual or all entries by using the Remove Bookmark(s) dialog.

LISTING 16.6 Chap16.mdb: Removing Selected Bookmarks or All Based on Selected Button

Option Compare Database

Option Explicit

```
Private Sub cmdClose_Click()
```

```
    DoCmd.Close
```

```
End Sub
```

```
Private Sub cmdRemoveAllBookmarks_Click()
```

```
    Dim intCurrItem As Integer
```

```
    '-- Hide the form, then select all bookmarks for removal.
```

```
    Me.Visible = False
```

```
    With Me!lboBookmarks
```

```
        For intCurrItem = 0 To .ListCount
```

```
            .Selected(intCurrItem) = True
```

```
        Next intCurrItem
```

```
    End With
```

```
End Sub
```

```
Private Sub cmdRemoveBookmark_Click()
```

```
    Me.Visible = False
```

```
    '-- Simply give a message if no bookmarks are selected, then close  
    ' the form.
```

```
    If Me!lboBookmarks.ItemsSelected.Count = 0 Then
```

```
        Beep
```

```
        MsgBox "No bookmarks were selected!", vbCritical, "No Bookmarks Removed"
```

```
        DoCmd.Close acForm, Me.Name
```

```
    End If
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
    Me!lboBookmarks.RowSource = Me.OpenArgs
```

```
End Sub
```

- Continuing with the `BookmarkAction` method in Listing 16.4, this step first checks to see whether the `apRemoveBookmarks` form is still open (meaning that no error occurred). If it is, it looks at any selected bookmarks to remove and removes them from the `colBookMarks` collection.

```
If apFormIsOpen("apRemoveBookmarks") Then
    Dim intNumRemoved As Integer
    intNumRemoved = 0
    '-- For any of the selected bookmarks to remove, do it.
    For Each varCurrItem In _
        Forms!apRemoveBookmarks!lboBookmarks.ItemsSelected
        colBookMarks.Remove varCurrItem + 1 - intNumRemoved
        intNumRemoved = intNumRemoved + 1
    Next varCurrItem
    DoCmd.Close acForm, "apRemoveBookmarks"
End If
```

TIP

This code segment demonstrates a useful way to utilize a dialog to get user input, and then read values from the dialog after users complete their entries and “close” the form. In this case, the form is opened with window mode being `acDialog`, so the code in `BookmarkAction` is halted until the form is either closed or hidden. The form is hidden by setting the `Visible` property to `False` in the two remove command buttons on the `apRemoveBookmark` form—`cmdRemoveAllBookmarks` and `cmdRemoveBookmarks`. When it resumes, the code examines the selected items and removes the appropriate bookmark items from the collection. Last, the form is closed for real.

- Rebuild the combo box by calling the `RebuildBookmarkCombo` routine described in the section “Adding a Bookmark.”

Moving to a Bookmark

In addition to being the last task, moving to an existing bookmark is actually the shortest task found in the `BookmarkAction` method, consisting of the following two code lines:

```
Set clsCurrBM = colBookMarks.Item(CInt(cboBookmark))
frmBookmark.Bookmark = clsCurrBM.ActualBM
```

This code creates a reference to the specified item in the `colBookMarks` collection, and then moves to the bookmark recorded there.

Managing Multiple Instances of the Same Form

The following sections look at a more advanced example for managing multiple instances of forms. In Chapter 2, you were introduced to this technique by opening the same form for employees and supervisors.

Looking at the Feature Set

Now you can see how to open multiple copies of the same form, using the same record source, but with different records chosen. This is useful when you have a browse window open for viewing records, and then want users to be able to open up multiple records at the same time with the same form. Some features also included are as follows:

- Selecting and deselecting from a list creates and closes forms automatically.
- Clicking a copy of the opened form deselects the selection in the browse window.
- Closing the browse window closes all copies of the edit form.

The code for performing these actions is examined later in the section “Examining the Code for Managing Multiple Copies.” In the meantime, look at some other objects used in this example. You will use the same table from the previous example, `tblEmployees` (refer to Figure 16.3). This example is easier to understand because you deal with the code behind only two forms, `frmCollectionFormExample` and `frmEmpForCollectionFormExample`, which again can be found on this book’s Web page.

Looking at the Forms Used to Open Copies of the Same Form

Look at the simple browse window created, `frmCollectionFormExample` (see Figure 16.7). To see how the form works, click a few names in the list. The `frmEmpForCollectionFormExample` form opens for each employee record chosen (see Figure 16.8).

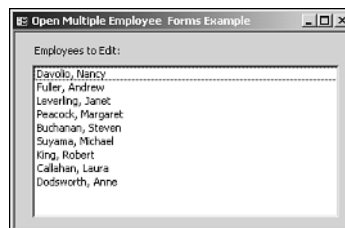
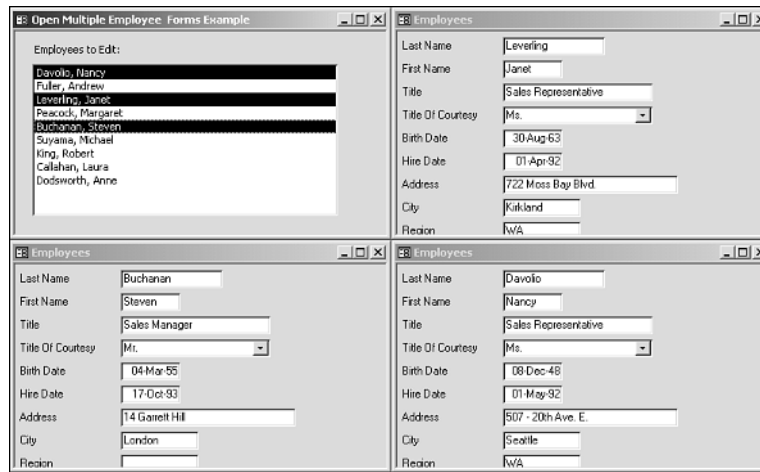


FIGURE 16.7

You can see the `frmCollectionFormExample` without any employee records displayed.

**FIGURE 16.8**

After opening three records, I tiled them by using the Window menu's Tile Vertically command.

Examining the Code for Managing Multiple Copies

Look at the code used with the `lboEmployees` list box. Listing 16.7 shows this code, plus the Declarations section of the `frmCollectionFormExample` form.

LISTING 16.7 Chap16.mdb: Opening and Closing Multiple Instances of the Form

```
Option Compare Database
'-- This collection is used to store the multiple instances
'   of the frmEmpForCollectionFormExample.
Public colEmpForms As New Collection
Private Sub lboEmployees_AfterUpdate()
    '-- If an employee is selected open the form.
    If lboEmployees.Selected(Me!lboEmployees.ListIndex) Then
        '-- Create a new instance of the form.
        Dim frmCurrEmp As Form_frmEmpForCollectionFormExample
        Set frmCurrEmp = New Form_frmEmpForCollectionFormExample
        '-- Set the Filter property to the current record selected in
        '   the list. Then set the FilterOn property to enact the filter.
        frmCurrEmp.Filter = "EmployeeID = " & _
            Me!lboEmployees.ItemData(Me!lboEmployees.ListIndex)
        frmCurrEmp.FilterOn = True
        '-- Make the form visible
        frmCurrEmp.Visible = True
        '-- Store the current EmployeeID into the Tag property on the form.
        frmCurrEmp.Tag = Me!lboEmployees.ListIndex
```


LISTING 16.7 Continued

```

    '-- Add the form to the colEmpForms collection,
    '   with the Employee ID as the key.
    colEmpForms.Add Item:=frmCurrEmp, _
        Key:=Me!lboEmployees.ItemData(Me!lboEmployees.ListIndex)
Else '-- If unselected, close the form.
    On Error Resume Next
    '-- Remove the form from the collection.
    colEmpForms.Remove Me!lboEmployees.ItemData(Me!lboEmployees.ListIndex)
End If
End Sub

```

Before going through the `lboEmployees_AfterUpdate` event procedure one step at a time, look at the declaration of the `colEmpForms` collection variable, which will contain the form copies. This variable is declared in the form's Declarations section:

```

Option Compare Database
'-- This collection is used to store the multiple instances
'   of the frmEmpForCollectionFormExample.
Public colEmpForms As New Collection

```

Next, walk through the code of the `After Update` event:

1. If an employee is selected, open the form. This comparison is performed by looking at the `Selected` property of the multiselect list box, `lboEmployees`:

```
If lboEmployees.Selected(Me!lboEmployees.ListIndex) Then
```
2. Create a new instance of the form:

```
Dim frmCurrEmp As Form_frmEmpForCollectionFormExample
Set frmCurrEmp = New Form_frmEmpForCollectionFormExample
```
3. Set the `Filter` property to the current record selected in the list. Then set the `FilterOn` property to enact the filter:

```
frmCurrEmp.Filter = "EmployeeID = " & _
    Me!lboEmployees.ItemData(Me!lboEmployees.ListIndex)
frmCurrEmp.FilterOn = True
```
4. Make the form visible:

```
frmCurrEmp.Visible = True
```
5. Store the current `EmployeeID` into the form's `Tag` property:

```
frmCurrEmp.Tag = Me!lboEmployees.ListIndex
```

TIP

Using the Tag property is a great way to store extra information about objects that you can examine in your code later during execution. Forms, reports, and controls all have Tag properties. The value stored in Tag must be of the data type String. To see another example of using Tag with a form, see Chapter 9, “Creating Powerful Forms.”

16

EXTENDING YOUR
VBA LIBRARY
POWER

6. Add the form to the colEmpForms collection, with the Employee ID as the key:

```
colEmpForms.Add Item:=frmCurrEmp, _
    Key:=Me!lboEmployees.ItemData(Me!lboEmployees.ListIndex)
```

The rest of the code in this event procedure occurs if the employee is deselected in the list. Then the following piece of code executes:

```
colEmpForms.Remove Me!lboEmployees.ItemData(Me!lboEmployees.ListIndex)
```

That's it for this event. The next (and final) routine is the event when you close a copy of the frmEmpForCollectionFormExample form directly. The event that occurs is the Close event of the form itself (see Listing 16.8).

LISTING 16.8 Chap16.mdb: If Form Is Closed Directly, Deselect Item in List Box

```
Private Sub Form_Close()
    On Error Resume Next
    With Forms!frmCollectionFormExample
        '-- if the form is selected, unselect-it, then remove the
        ' form from the colEmpForms collection.
        If len(!lboEmployees.Selected(Me.Tag)) > 0 Then
            !lboEmployees.Selected(CInt(Me.Tag)) = False
            .colEmpForms.Remove !lboEmployees.ItemData (CInt(Me.Tag))
        End If
    End With
End Sub
```

The last action to occur, where all forms close when the browse window is closed, is a freebie because the variable colEmpForms is declared in the frmCollectionFormExample form. Even though it was a public variable, because it was declared in the code behind form, it's released when the form is closed.

Summary

My goal with this chapter was to get you experimenting with collections and class modules. Work with these examples and then try to build on them. For example, make the Bookmark Tracker so that it will take an arbitrary number of description elements.

You should now have at least a few good ideas of where to use class modules and collections. They are such powerful VBA features that it's a shame some developers don't use them as often as they could. In the next chapter you add the Bookmark Tracker to any form in your application by using an add-in that you create. For more background information on class modules, collections, and bookmarks, see these chapters:

- Chapter 2, “Coding in Access 2002 with VBA,” gives an overview of using VBA in Access, including using class modules and collections.
- Chapter 4, “Working with Access Collections and Objects,” discusses various Access objects.

Creating Your Own Wizards and Add-ins

CHAPTER

17

IN THIS CHAPTER

- Understanding Access Wizards, Builders, and Add-Ins 512
- Looking at the Wizards and Add-In Registry Entries 513
- Creating Your Own Add-Ins 515
- Using Access Code Libraries 537

When it comes to expandability (as well as the capability to call API routines from VBA as explained in Chapter 15, “Extending the Power of Access with API Calls”), Access allows you to create tools known as wizards and builders. These tools, along with VBA code libraries, are the subject of this chapter.

Understanding Access Wizards, Builders, and Add-Ins

You’ve likely seen the terms *wizards*, *builders*, and *add-ins* used in several places, including in this book. You also might be somewhat confused as to which is which.

Wizards and builders can be add-ins. An *add-in* tool is not part of the main Access product, but gives additional capability. Some add-ins included with Access fall into three categories:

- *Wizards* are generally multipage forms that help you create an object or perform tasks step by step (Figure 17.1 shows an example). Access uses wizards to create new instances of all major objects, including forms, reports, tables, queries, and Data Access Pages. Other examples are the Table and Performance Analyzers, which started out as add-ins on the Add-ins menu but were later incorporated into the product.

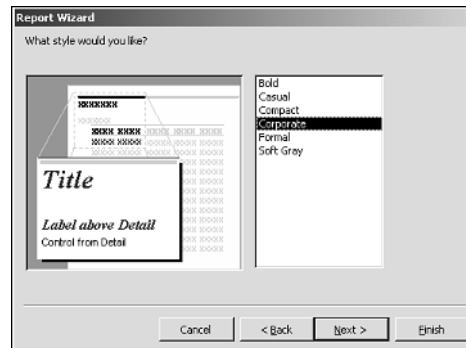


FIGURE 17.1

Using a wizard to create a report gives you a great start with a standard look and feel.

- *Builders* are generally single-page wizards that you can attach to a property or command. An example of a builder is the Input Mask Wizard.
- General *add-ins* are products you can purchase to enhance your development environment. Check out the add-in demos on this book’s Web page at www.sampublishing.com.

Looking at the Wizards and Add-In Registry Entries

All wizards (and builders) used by Access 2002 are specified in the Windows Registry under this key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\Office\10.0\Access\Wizards
```

This key includes subkeys for the various wizard types. Figure 17.2 shows examples of the wizards and menu add-ins keys under the Access key. You can see that quite a few wizards are packed in Access 2002. Table 17.1 shows you almost all the possible wizards, complete with their actual Registry key names.

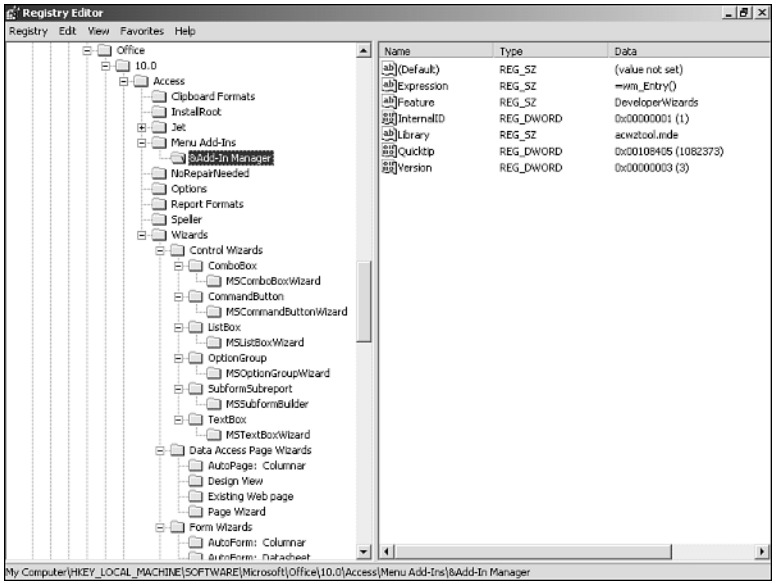


FIGURE 17.2
Notice both the menu add-ins and start of the wizards available in Access 2002.

TABLE 17.1 Registry Entries of Access Wizard Objects
Control Wizards

MSComboBoxWizard	MSOptionGroupWizard
MSCommandButtonWizard	MSSubformBuilder
MSListBoxWizard	MSTextBoxWizard

TABLE 17.1 Continued

	<i>Data Access Page Wizards</i>
AutoPage: Columnar	Existing Web Page
Design View	Page Wizard
	<i>Form Wizards</i>
AutoForm: Columnar	Chart Wizard
AutoForm: Datasheet	Design View
AutoForm: PivotChart	Form Wizard
AutoForm: PivotTable	PivotTable Wizard
AutoForm: Tabular	
	<i>Preferences</i>
AutoFormat	Documentor
	<i>Property Wizards</i>
MSFieldBuilder	MSLinkMasterFieldsBuilder
MSFuringanaControlBuilder	MSODBCCConnectStrBuilder
MSInputMaskBuilder	MSPictureBuilder
MSLinkChildFieldsBuilder	
	<i>Query Wizards</i>
Crosstab Query Wizard	Find Unmatched Query Wizard
Design View	Simple Query Wizard
Find Duplicates Query Wizard	
	<i>Report Wizards</i>
AutoReport: Columnar	Design View
AutoReport: Tabular	Label Wizard
Business Form Wizard	Post Card Wizard
Chart Wizard	Report Wizard
	<i>Table Wizards</i>
Datasheet View	Link Table
Design View	Table Wizard
Import Table	

Figure 17.2 also shows the entry for menu add-ins and the Add-In Manager. What's interesting is that the Add-In Manager itself is an add-in. Entries that must be set for your add-ins and wizards to work are shown on the right side of Figure 17.2.

NOTE

To get all the wizards, you had to specify that you want all wizards loaded when you install Access 2002. By default, they are *not* installed.

Creating Your Own Add-Ins

I used to think that spending time to create your own add-in was silly. However, I'm starting to believe that creating add-ins to perform a number of tasks would be very cool. Hence, this chapter. Don't get me wrong—add-ins have always been cool, but I always thought that programming them belonged in the realm of third-party vendors such as FMS, who creates numerous Access and Visual Basic add-ins. (Some FMS product demos are available on this book's Web page at www.sampublishing.com.)

I changed my thinking because I've come up with a tool of my own that I find pretty useful, and I want a quick way to put it on most of my forms. I want it placed when a form is created, without the hassle of having to remember to paste the code behind the forms.

Working with the Bookmark Tracker Wizard

The tool created by the wizard in this chapter is the Bookmark Tracker created in the previous chapter. The wizard that you will create in the following sections will place a Bookmark Tracker combo box on the form you are in. It will also copy all objects used by the combo box for managing the bookmarks into the current database.

Reviewing the Bookmark Tracker

Before going into detail on what the wizard does, let's review the Bookmark Tracker itself. The Bookmark Tracker allows users to manage multiple bookmarks on an open form (see Figure 17.3).

Here are some of the features of the Bookmark Tracker:

- You can add bookmarks just by moving to the record you want to bookmark and then selecting << Add a New Bookmark >>.
- To delete bookmarks, select << Remove Bookmark(s) >> from the combo box. You are then given the dialog to remove bookmarks.
- You can move to a stored bookmark by selecting one from the list.

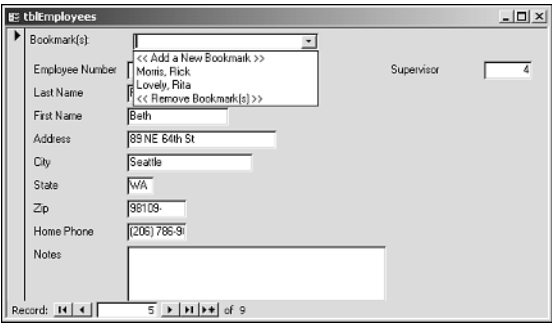


FIGURE 17.3

The Bookmark Tracker is an example of using collections and class modules.

The Bookmark Tracker consists of one form (not counting the form that the combo is placed on) and two class modules:

<i>Object Name</i>	<i>Purpose</i>
bmtRemoveBookmarks (Form type)	Allows for removing one or all bookmarks from the collection
clsBookmarkItems (Class Module type)	Tracks information for each bookmark
clsBookmarkManagement (Class Module type)	Performs maintenance routines for the Bookmark Tracker

These objects, with the code in Listing 17.1, are necessary for the Bookmark Tracker to do its job.

LISTING 17.1 Managing Multiple Bookmarks on a Given Form

```
Option Compare Database
'-- Declare a clsBookmarkManagement variable.

Private bmtForm As clsBookmarkManagement
Private Sub cboBookmarkTracker_AfterUpdate()
    '-- Call the BookmarkAction method, for all actions.
    bmtForm.BookmarkAction
End Sub

Private Sub Form_Load()
    '-- Create an instance of the clsBookmarkManagement class
    Set bmtForm = New clsBookmarkManagement
    '-- Call the custom initialization routine, passing the necessary items.
    bmtForm.InitBookmarks Me!cboBookmarkTracker, "LastName", "FirstName"
End Sub
```

NOTE

The Bookmark Tracker wizard created most of the code in Listing 17.1. However, the `Option` statement will already be on the form, not created by the wizard. It's included in Listing 17.1 to show that the line below the `Option` statement is in the module's Declarations section.

If you need more information on Listing 17.1 or the Bookmark Tracker in general, reread Chapter 16, "Extending Your VBA Library Power with Class Modules and Collections." Listing 17.1 is repeated from Chapter 16, in fact.

Features of the Bookmark Tracker Wizard

The Bookmark Tracker Wizard looks a lot like other wizards in Access 2002 in that it consists of multiple dialogs that take you logically from one step to another. You will see the wizard's dialogs in the next section. In the meantime, look at what the Bookmark Tracker Wizard accomplishes:

- It lets users select the target form, as well as the section of the form, on which to place the Bookmark Tracker combo box (see Figure 17.4).
- It combines up to two fields to display in the combo box from those available based on the target form's record source (see Figure 17.5).
- It creates a combo box with a user-specified name (see Figure 17.6).

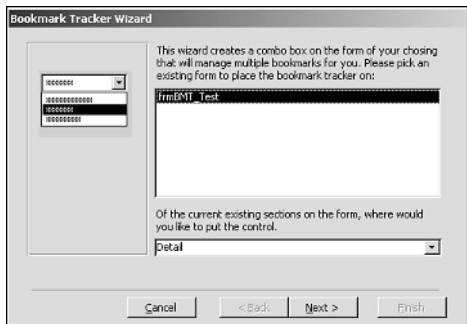
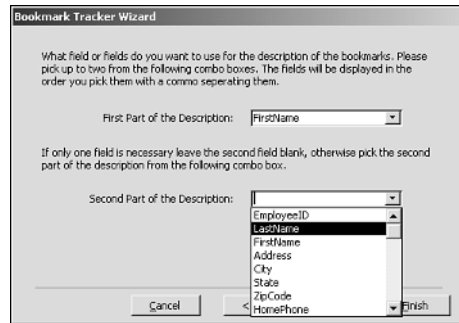


FIGURE 17.4

Select the form and area on the form where you want to place the combo box.

**FIGURE 17.5**

The fields in these combo boxes change based on the target form's record source.

**FIGURE 17.6**

The wizard also checks to see whether a control with the same name already exists on the form.

In addition to the items displayed on the Bookmark Tracker Wizard's user interface, the wizard generates code behind the target form, as shown in Listing 17.1. It also does the following:

- It checks to see whether the objects listed in Table 17.1 are already in the target database.
- If objects don't exist in the target database, it copies them into the database and then marks them as hidden.

As you should see by now, this wizard is full of features and will get you going to create add-ins of your own. But before pulling apart the wizard, let's see what it takes to install the new add-in.

Installing Add-Ins in Access

You can install your add-ins in Access in numerous ways. Here are two ways to install, the first of which will be discussed in greater detail in the next two sections:

- Manually register the add-in by using the Add-In Manager.
- Use the techniques discussed in Chapter 18, “Manipulating the Registry with VBA,” to set the values.

Looking at the USysRegInfo Table

No matter which method you decide to use to install the add-in, Access allows you to specify the settings necessary to update the Windows Registry. These values are set within the add-in database in the USysRegInfo table. Figure 17.7 shows this table for the add-in BMTWiz.mda, created for this chapter’s Bookmark Tracker Wizard. You can find this add-in database on this book’s Web page at www.sampublishing.com.

NOTE

To see this table, you must be able to see system objects. In the database explorer, choose Options from the Tools menu. On the View page, select System Objects in the Show category. Also check Hidden Objects so that you can see hidden objects later in the chapter.

Subkey	Type	ValName	Value
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Bookmark &Tracker Wizard	0		
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Bookmark &Tracker Wizard	1 Expression	=bmt_Entry()	
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Bookmark &Tracker Wizard	1 Library	AccDir\BMTWiz.mda	

FIGURE 17.7

Access makes it very easy to specify the Registry settings necessary to use your wizard.

TIP

Although the .mda (Microsoft database add-in) extension was used, it’s still an Access database. You use the .mda extension so that the file won’t show up when you’re opening databases, but will show when using the Add-In Manager.

You also could create an .mde, which strips out the source code, if you don’t want other developers looking at the code. This is how wizards are shipped with Access. In past versions, you could get the source code for Access wizards from Microsoft’s Web site. However, the company decided that because of support issues and code changes from version to version, wizard code shouldn’t be published.

The fields necessary for Access to register the add-in are as follows:

<i>Field Name</i>	<i>Purpose</i>
Subkey	The key listed under the menu add-ins subkey. The full key in this case will be HKEY_LOCAL_MACHINE\Software\Microsoft\Office\10.0\Access\Menu Add-Ins\&Bookmark Tracker Wizard.
Type	Denotes the first entry (0) or the entry's data type: 1 for String, 4 for DWORD.
ValName	Value names placed under the subkey. The value names used here specify the filename (library) and calling routine used to invoke the add-in (expression).
Value	The actual values used for the library name and expression.

NOTE

The Add-In Manager replaces the Subkey field text HKEY_CURRENT_ACCESS_PROFILE with the actual Registry path when the add-in is registered. Use the standard HKEY_LOCAL_MACHINE to supply the full path. The same goes for the AccDir\ text in the Value field. This is replaced by the add-in location used by Access. Access stores user add-ins in ApplicationData\Microsoft\AddIns, depending on where the Windows version stores preferences. Access add-ins are stored (as MDE files) in the Office\LCID directory (LCID is 1033 for English).

Installing Your Wizard with the Add-In Manager

To use the Add-In Manager, follow these steps while in the Chap17.mdb database:

1. Choose Add-Ins and then Add-In Manager from the Tools menu. (Add-Ins might be the only submenu item at this point.)
2. In the Add-In Manager, the Available Add-Ins list might be empty. Click the Add New button to display the Open File dialog, which will point to the location in which Access stores add-ins.

TIP

Note the add-ins' location in the Look In drop-down list. After you select the actual location for the add-in database, Access copies and stores it in the official add-ins' location. It's a good idea to know where the add-ins are ultimately located; if you

change the original copy, be sure not to update the copy in the add-ins' location (unless you stored the original there to begin with). I don't recommend storing your add-in originals in the Add-Ins folder for obvious versioning reasons.

3. Locate your add-in database—in this case, BMTWiz.mda, which you can download from this book's Web site at www.sampublishing.com. When it's selected, click Open.

You've installed the Bookmark Tracker Wizard and can see it in the Available Add-ins list with an × next to it (see Figure 17.8). The Add-In Manager also adds the necessary entries into the Registry as per the USysRegInfo table (see Figure 17.9).

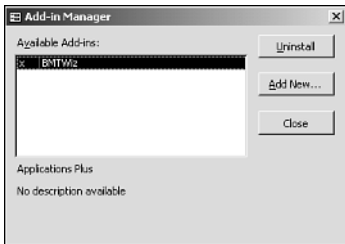


FIGURE 17.8

You can see and add available custom add-ins from the Add-In Manager.

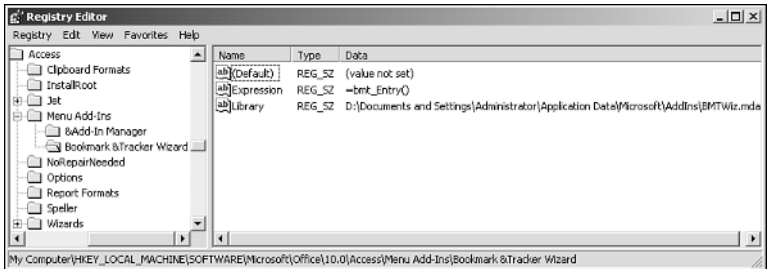


FIGURE 17.9

Let the Add-In Manager do all the work for you.

NOTE

Although you can create a routine to install your add-in programmatically, you then have to know the Registry locations for menu add-ins and the AccDir location for the add-in folder, both of which can change from version to version.

Programming the Bookmark Tracking Wizard

Notice a number of different VBA commands used for the Bookmark Tracking Wizard:

- Populating combo boxes based off current form sections and record source fields
- Creating controls
- Copying objects from the add-in database to an application database
- Locating code lines from modules
- Adding code lines to modules

You will see these techniques used as each page of the wizard is discussed. To start, look at the overall form itself, `frmBookmarkTrackerWizard`, in Design mode. This form uses Access's Tab control with no tabs visible (to set the tabs to be invisible, set the Style property on the Tab control to None). Notice in Figure 17.10 the pages listed in the Object drop-down list (names starting with `pg`). You can find this form in the `BMTWiz.mda` database in the `\Examples\Chap17` folder on this book's Web page.

TIP

Although you can name your pages 1, 2, 3, and so on, I like to give mine meaningful names that I can switch to in code. If I do this, I don't have to rename pages when I move them around.

TIP

Until you are ready to use your wizard for production, you might want to leave your tabs visible (`Style = Tabs`). It's more convenient to click the tabs when in Design view than it is to use the Object drop-down list.

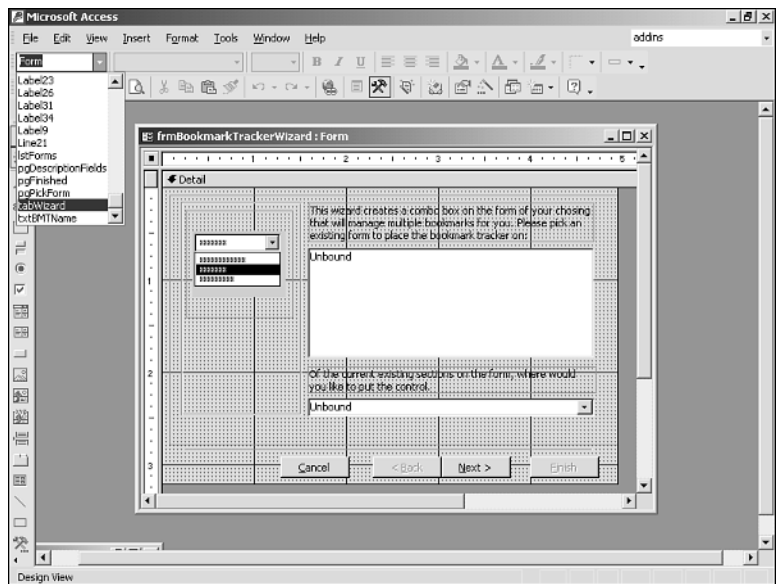


FIGURE 17.10
The Tab control really makes it a lot easier to create multipage forms such as the ones you use for wizards.

NOTE

You might be saying to yourself, “How can I open the database if I have it installed as an add-in?” Remember that you’re using the original BMTWiz.mda, not the one in the Add-Ins folder. If you were to use that database, you would get an error telling you it’s already opened as an add-in.

Initializing the Wizard Form

It’s time to look at some routines connected to the main wizard form. The first subroutine is the form’s Open event (shown in Listing 17.2).

LISTING 17.2 BMTWiz.mdb: Setting Up the Wizard for Use

```
Private Sub Form_Open(Cancel As Integer)
    Dim strFormList As String
    Dim strCurrDoc As String
    Dim aobjForm As AccessObject
```


LISTING 17.2 Continued

```

'-- Look through the forms in the application database and grab those
'-- that aren't hidden and not the same name as this form.
For Each aobjForm In CurrentProject.AllForms
    strCurrDoc = aobjForm.Name
    If Not GetHiddenAttribute(acForm, strCurrDoc) _
        And strCurrDoc <> Me.Name Then
        strFormList = strFormList & strCurrDoc & "; "
    End If
Next aobjForm

'-- Trim off the last semi-colon and assign the string to
'-- the forms listbox.
strFormList = Left$(strFormList, Len(strFormList) - 2)
Me!lstForms.RowSource = strFormList
End Sub

```

The `Open` event routine first populates the `lstForms` list box with the available unhidden forms that don't have the same name as the wizard form. (This form's name is important because when you start creating your own wizard, you will probably run the wizard from within itself.) The routine then creates a row source for the `cboSection` combo box by going through the target form's `Sections` array.

The `Form_Open` event makes good use of the new `CurrentProject` object and `AllForms` collection. To read more about these objects, see Chapter 4, "Working with Access Collections and Objects."

Looking at the Command Buttons

The tasks of the `cmdCancel`, `cmdPrevious`, and `cmdNext` command buttons are fairly straightforward (see Listing 17.3). The `cmdCancel_Click` routine closes the wizard form if the user clicks the button. The `cmdPrevious` and `cmdNext` buttons call the `CheckFilledFields` function to ensure that the required fields are filled in; then they increment and decrement the page value accordingly.

LISTING 17.3 BMTWiz.mdb: Command Buttons Doing a Lot of Work

```

Private Sub cmdCancel_Click()
    DoCmd.Close acForm, Me.Name
End Sub

Private Sub cmdNext_Click()
    '-- Increment the tab page
    If CheckFilledFields() Then

```

LISTING 17.3 Continued

```

    Me!tabWizard = Me!tabWizard + 1
End If
End Sub

Private Sub cmdBack_Click()
    '-- Decrement the tab page
    If CheckFilledFields() Then
        Me!tabWizard = Me!tabWizard - 1
    End If
End Sub

```

One cool thing about these buttons is that you can use them generically on another wizard without having to change any code. The only code change would be inside `CheckFilledFields` (see Listing 17.4) because the required fields will change from wizard to wizard.

LISTING 17.4 BMTWiz.mdb: Making Sure That Required Fields Are Filled In

```

Function CheckFilledFields() As Boolean
    CheckFilledFields = True
    '-- Check to make sure the needed fields are supplied.

    Select Case Me!tabWizard
        Case 0
            If IsNull(Me!lstForms) Then
                Me!lstForms.SetFocus
                MsgBox "A form must be selected to continue.", _
                    vbInformation, Me.Caption
                CheckFilledFields = False
            End If
        Case 1
            If IsNull(Me!cboField1) Then
                Me!cboField1.SetFocus
                MsgBox "You must select at least one field to include in the " & _
                    "description of the bookmark.", vbInformation, Me.Caption
                CheckFilledFields = False
            End If
    End Select
End Function

```

The code in Listing 17.4 checks for two fields to be filled in on the wizard. First, if we're leaving the first page of the wizard, a form must be picked from the `lstForms` list box. Otherwise,

if we're leaving the second page, the first combo box of the description (cboField1) must be filled in.

Switching Pages of the Wizard's Tab Control

Next, examine what occurs when you switch pages of the tabWizard Tab control by clicking the cmdNext and cmdPrevious command buttons. Switching the tabWizard pages triggers the Change event (see Listing 17.5).

LISTING 17.5 BMTWiz.mdb: Tasks Performed When Changing Pages on the Tab Control

```
Private Sub tabWizard_Change()  
    Me!tabWizard.Pages(Me!tabWizard).SetFocus  
  
    '-- If the combo box page, assign the chosen form's rowsource  
    '-- to the fields combo boxes.  
    If Me!tabWizard = 1 Then  
        Me!cboField1.RowSource = Forms(Me!lstForms).RecordSource  
        Me!cboField2.RowSource = Forms(Me!lstForms).RecordSource  
    End If  
  
    '-- Handle the cmdNext button based on the current page  
    If Me!tabWizard = Me!tabWizard.Pages.Count - 1 Then  
        Me!cmdNext.Enabled = False  
    ElseIf Not Me!cmdNext.Enabled Then  
        Me!cmdNext.Enabled = True  
    End If  
  
    '-- Handle the cmdBack button based on the current page  
    If Me!tabWizard = 0 Then  
        Me!cmdBack.Enabled = False  
    ElseIf Not Me!cmdBack.Enabled Then  
        Me!cmdBack.Enabled = True  
    End If  
  
End Sub
```

This event procedure performs the following steps:

1. If the tab control value is one (second page), it fills in the row source of each Description field (cboField1 and cboField2) with the field list located in the RecordSource property of the chosen form in lstForms.

TIP

Like other objects and collections in Access, the Tab control value is zero-based. Therefore, page 1 has a value of 0, page 2 a value of 1, and so on.

2. The code checks whether the Tab control is showing the last page. If this control is on the last page, the code disables the `cmdNext` button; otherwise, the code enables it.
3. The code looks to see whether the Tab control is on the first page and performs the same task for the `cmdPrevious` button as performed for `cmdNext` in step 2. If the Tab control is on the first page, the code disables `cmdPrevious`; otherwise, the code enables it.

The final command button to look at is the `cmdFinish` button. However, because the `Click` event for this button performs the bulk of the work, I want to save it for last. For now, look at the validation that goes on for each field on the wizard because the validation performed will be things you need to watch out for when creating your own wizards.

Validating Fields on the First Page of the Wizard Form

The first control to look at, `lstForms`, has two events programmed: `BeforeUpdate` and `AfterUpdate`. You can see the `BeforeUpdate` event here:

```
Private Sub lstForms_BeforeUpdate(Cancel As Integer)
    Dim mdlForm As Module
    DoCmd.OpenForm Me!lstForms, acDesign, WindowMode:=acHidden
    Set mdlForm = Forms(Me!lstForms).Module
    If CheckBMTAlreadyExists(mdlForm) Then
        MsgBox "This form already has a bookmark tracker combo on it!", _
            vbInformation, Me.Name
        Cancel = True
    Exit Sub
End If
End Sub
```

This code performs three interesting tasks:

1. It opens the target form hidden in Design view with the line of code that reads
`DoCmd.OpenForm Me!lstForms, acDesign, WindowMode:=acHidden`
2. The form's module is referenced by setting it to the variable `mdlForm` in the line
`Set mdlForm = Forms(Me!lstForms).Module`
3. The module is then passed to the `CheckBMTAlreadyExists()` function, which checks to see whether the Bookmark Tracker already exists on the form. The section of code that uses this is as follows:

```
If CheckBMTAlreadyExists(mdlForm) Then
    MsgBox "This form already has a bookmark tracker combo on it!", _
        vbInformation, Me.Name
    Cancel = True
    Exit Sub
End If
```

Listing 17.6 shows the code for the `CheckBMTAlreadyExists` function.

LISTING 17.6 BMTWiz.mdb: Seeing Whether a Bookmark Tracker Combo Box Already Exists on the Form

```
Function CheckBMTAlreadyExists(mdlForm As Module) As Boolean
    Dim lngStartLine As Long, lngStartCol As Long
    Dim lngEndLine As Long, lngEndCol As Long
    mdlForm.Find "Private bmtForm As clsBookmarkManagement", lngStartLine, _
        lngStartCol, lngEndLine, lngEndCol
    If lngStartLine <> 0 Then
        CheckBMTAlreadyExists = True
    End If
End Function
```

The routine finds whether the Bookmark Tracker already exists on the form by looking through the module's VBA code. By using the `Find` method of the `Module` object, it searches for the existence of the following statement:

```
"Private bmtForm as clsBookmarkManagement"
```

Finding this statement in the module code tells the wizard that a copy of the Bookmark Tracker exists. At this point, users can select another form to place the Bookmark Tracker on or cancel the wizard.

The module `Find` method is a cool command that uses the following syntax:

```
module.Find strTextToFind, lngStartLine, lngStartCol, lngEndLine, lngEndCol
```

If the text is found in the module, the line number where the text is located is placed in *lngStartLine*.

NOTE

The `Find` method also returns a Boolean that you can use to see whether the text was found.

In the `AfterUpdate` event of the `lstForms` list box control, the code goes through the `Sections` array and builds the row source for the `cboSection` combo box. Listing 17.7 shows the code for `lstForms_AfterUpdate`.

LISTING 17.7 BMTWiz.mdb: Building the Row Source for the `cboSections` Combo Box

```
Private Sub lstForms_AfterUpdate()
    Dim intCurrSec As Integer
    Dim strSecName As String
    Dim frmTarget As Form
    Dim strSource As String
    On Error Resume Next

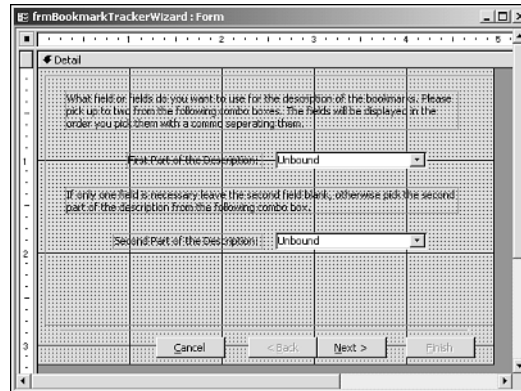
    '-- Build a list of possible sections if they exist.
    For intCurrSec = 0 To 4
        strSecName = Forms(Me!lstForms).Section(intCurrSec).Name
        If Err.Number = 0 Then
            strSource = strSource & "; " & intCurrSec & "; '" & strSecName & "'"
        Else
            Err.Clear
        End If
    Next
    strSource = Mid$(strSource, 2)
    Me!cboSection.RowSource = strSource
End Sub
```

The last event procedure to look at is the `Enter` event for the `cboSection` combo box. This event checks to make sure that a form is selected in the `lstForms` list box. Otherwise, a message is given to the effect that a form must be selected. Here's the code:

```
Private Sub cboSection_Enter()
    If IsNull(Me!lstForms) Then
        Me!lstForms.SetFocus
        MsgBox "A form must be selected to continue.", _
            vbInformation, Me.Caption
    End If
End Sub
```

Fields and Event Procedures on the Second Page of the Wizard Form

Page two of the wizard consists of two combo boxes: `cboField1` and `cboField2`. Figure 17.11 shows page two of the wizard in Design view.

**FIGURE 17.11**

Let users select two fields for a bookmark's description.

Only two event procedures are used on the second page of the wizard:

```
Private Sub cboField1_AfterUpdate()
    Me!cmdFinish.Enabled = Not IsNull(Me!cboField1)
End Sub

Private Sub cboField2_Enter()
    If IsNull(Me!cboField1) Then
        MsgBox "You need to fill in the field for the first part of " & _
            "description before filling in part two.", vbInformation, Me.Caption
        Me!cboField1.SetFocus
    End If
End Sub
```

The first procedure, `cboField1_AfterUpdate`, enables the `cmdFinished` button if `cboField1` is filled in with data. The next procedure, `cboField2_Enter`, makes sure that you've filled in `cboField1` before attempting to enter data in `cboField2`.

TIP

Always determine the minimal number of controls required for completing the wizard's task before enabling your `cmdFinished` button. It's worth the effort because it makes your wizard that much more convenient.

Fields and Event Procedures on the Wizard's Third Page

Page three, the wizard's final page, consists of the txtBMTName text box and chkDesign check box. Figure 17.12 shows page three of the wizard in Design view.

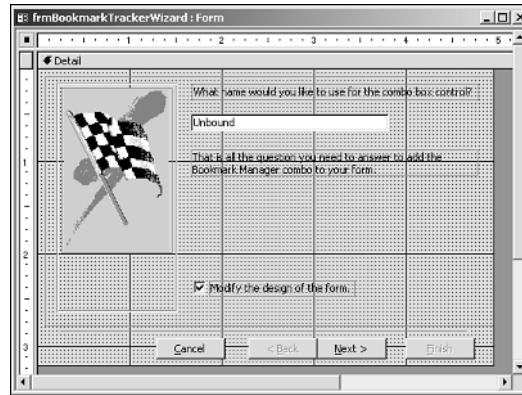


FIGURE 17.12

Finish up with the final page of the Bookmark Tracker Wizard.

The chkDesign control allows you to specify whether you want to keep the target form, including the VBE, open in Design view when the wizard is complete. If this isn't specified, the wizard closes the form and the VBE. (You'll see more about this in the next section.)

The txtBMTName text box allows you to pick your own name for the combo box control to be placed on the target form. txtBMTName has two events programmed—BeforeUpdate and AfterUpdate, as shown in Listing 17.8.

LISTING 17.8 BMTWiz.mdb: Testing the Combo Box Name Entered

```
Private Sub txtBMTName_BeforeUpdate(Cancel As Integer)
    If IsNull(Me!txtBMTName) Then
        MsgBox "You must supply a name for the combo box control." & _
            vbCrLf & vbCrLf & "Please enter a name.", vbInformation, Me.Caption
        Cancel = True
    End If
End Sub

Private Sub txtBMTName_AfterUpdate()
    If CheckBMTNameDupe() Then
        Exit Sub
    Else
        mfNameChecked = True
    End If
End Sub
```


PART III

The first routine, `txtBMTName_BeforeUpdate`, checks to ensure that the name of the combo box is blank. `txtBMTName_AfterUpdates` calls the `CheckBMTNameDupe()` function, which checks whether a control already exists on the target form with the name specified in `txtBMTName`.

Listing 17.9 shows the code for `CheckBMTNameDupe()`. This code goes through each control on the target form and compares the name to that entered in `txtBMTName`. If a match is found, focus moves to the `txtBMTName` text box and `True` is passed back.

LISTING 17.9 BMTWiz.mdb: Testing for an Already Existing Control Name

```
Function CheckBMTNameDupe() As Boolean
    Dim ctlCurrent As Control
    For Each ctlCurrent In Forms(Me!lstForms).Controls
        If ctlCurrent.Name = Me!txtBMTName Then
            MsgBox "The control already exists with the name " & _
                "specified for the combo box." & vbCrLf & vbCrLf & _
                "Please enter a new name.", vbInformation, Me.Caption
            Me!txtBMTName.SetFocus
            CheckBMTNameDupe = True
            Exit Function
        End If
    Next ctlCurrent
End Function
```

Finishing with the Wizard

After you enter all necessary data and click the `cmdFinished` button, the `cmdFinished_Click` event procedure is executed. This event procedure basically

- Builds the Bookmark Tracker combo box
- Copies the necessary support form and class modules to the target database and hides them
- Creates the code behind the form needed to use the Bookmark Tracker combo box

All these tasks, including creating the VBA code behind the target form, are done with VBA commands. Listing 17.10 shows the `cmdFinished_Click` event procedure.

LISTING 17.10 BMTWiz.mdb: Finishing Up the Wizard

```
Private Sub cmdFinish_Click()
    If Not mfNameChecked Then
        If CheckBMTNameDupe() Then
            Exit Sub
        End If
    End If
End Sub
```

LISTING 17.10 Continued

```

Dim frmTarget As Form
Set frmTarget = Forms(Me!lstForms)
Dim ctlCombo As Control
Dim ctlLabel As Control
Dim strTarget As String
Dim blnDesign As Boolean
Dim mdl As Module

strTarget = Me!lstForms
blnDesign = Me!chkDesign

'-- First, test to see if a module even exists. If not, create it.
If Not frmTarget.HasModule Then
    frmTarget.HasModule = True
End If

'-- Set up the combo box to handle manipulating the bookmarks
Set ctlCombo = CreateControl(frmTarget.Name, acComboBox, _
    Me!cboSection, , , 1710, 100, 2880)

With ctlCombo
    .Name = Me!txtBMTName
    .ColumnCount = 2
    .ColumnWidths = "0;2"
    .RowSourceType = "Value list"
    .RowSource = "-1; '<< Add a New Bookmark >>'"
End With

'-- Create the label for the bookmark combo.
Set ctlLabel = CreateControl(strTarget, acLabel, Me!cboSection, _
    Me!txtBMTName, , 60, 70, 1600, 300)
ctlLabel.Caption = "Bookmark(s):"

'-- Get a reference to the target form's module
Set mdl = frmTarget.Module

'-- Copy the Bookmark Tracker objects into the current application
CopyObjectsToDatabase

CreateDeclarationsArea mdl
CreateFormLoadEvent mdl
CreateComboAfterUpdateEvent mdl
DoCmd.Close acForm, Me!lstForms, acSaveYes
DoCmd.Close acForm, Me.Name

```

LISTING 17.10 Continued

```

Set frmTarget = Nothing
Set mdl = Nothing

If blnDesign Then
    DoCmd.OpenForm strTarget, acDesign
Else
    Application.VBE.MainWindow.Visible = False
End If
End Sub

```

The cmdFinished_Click routine does a lot, so I will go through it a step at a time:

1. This routine checks, if it hasn't checked in txtBMTName_AfterUpdate, to see whether a control already exists on the target form. That way, if users click the cmdFinished button and never go to txtBMTName, the default name is still checked.

```

If Not mfNameChecked Then
    If CheckBMTNameDupe() Then
        Exit Sub
    End If
End If

```

2. It tests to see whether the form already has a module. If the form doesn't have a module, it creates one by setting the form's HasModule property to True:

```

If Not frmTarget.HasModule Then
    frmTarget.HasModule = True
End If

```

3. Through the CreateControl() function, the cmdFinished_Click routine creates the Bookmark Tracker combo box and the combo box's label. The CreateControl() function does as its name implies—allows you to create controls on a form, specifying all the necessary properties.

```

'-- Set up the combo box to handle manipulating the bookmarks
Set ctlCombo = CreateControl(frmTarget.Name, acComboBox, _
    Me!cboSection, , , 1710, 100, 2880)

```

```

With ctlCombo
    .Name = Me!txtBMTName
    .ColumnCount = 2
    .ColumnWidths = "0;2"
    .RowSourceType = "Value list"
    .RowSource = "-1; '<< Add a New Bookmark >>'"
End With

```

```
'-- Create the label for the bookmark combo.
Set ctlLabel = CreateControl(strTarget, acLabel, Me!cboSection, _
    Me!txtBMTName, , 60, 70, 1600, 300)
ctlLabel.Caption = "Bookmark(s):"
```

4. The cmdFinished_Click routine gets a reference to the target form's module:
Set mdl = frmTarget.Module
5. The code calls the CopyObjectsToDatabase routine, shown in Listing 17.11.

LISTING 17.11 BMTWiz.mdb: Copying Objects from the Add-In Database to the Current Application Database

```
Sub CopyObjectsToDatabase()
    Dim strTest As String
    '-- If the first module is not already in the target db, copy them all.
    On Error Resume Next
    strTest = _
        CurrentDb.Containers!Modules.Documents("clsBookmarkManagement").Name

    If Err.Number > 0 Then
        DoCmd.CopyObject CurrentDb.Name, "clsBookmarkManagement", _
            acModule, "clsBookmarkManagement"
        DoCmd.CopyObject CurrentDb.Name, "clsBookmarkItems", _
            acModule, "clsBookmarkItems"
        DoCmd.CopyObject CurrentDb.Name, "bmtRemoveBookmarks", _
            acForm, "bmtRemoveBookmarks"
        SetHiddenAttribute acModule, "clsBookmarkItems", True
        SetHiddenAttribute acModule, "clsBookmarkManagement", True
        SetHiddenAttribute acForm, "bmtRemoveBookmarks", True
    End If
End Sub
```

The code in Listing 17.11 checks to see whether the class module clsBookmarkManagement exists in the target application. If the class module exists, the code doesn't bother to copy the objects into the target database. If clsBookmarkManagement doesn't exist, the code copies two class modules into the target database, and then sets them as hidden through the SetHiddenAttribute method.

6. The cmdFinished_Click routine (refer to Listing 17.10) generates the code line needed in the target form's Declarations section by calling CreateDeclarationsArea and passing it a reference to the target form's module. You can see the code for CreateDeclarationsArea here:

```
Sub CreateDeclarationsArea(mdl As Module)
    mdl.AddFromString vbCrLf & _
```

```

        "'-- Declare a clsBookmarkManagement variable." & _
        vbCrLf & "Private bmtForm As clsBookmarkManagement"
    End Sub

```

NOTE

By using the module's `AddFromString` method, VBA places the supplied string at the bottom of the Declarations section. Is that convenient, or what?

7. `cmdFinished_Click` calls `CreateFormLoadEvent`, which creates the Load event for the target form, if it doesn't already exist. Otherwise, `CreateFormLoadEvent` adds to the existing code base in Listing 17.12.

LISTING 17.12 BMTWiz.mdb: Creating the Target Form's Load Event

```

Sub CreateFormLoadEvent(mdl As Module)
    Dim lngStartLine As Long, lngStartCol As Long
    Dim lngEndLine As Long, lngEndCol As Long
    Dim intIncrement As Integer
    mdl.Find "Form_Load", lngStartLine, lngStartCol, lngEndLine, lngEndCol

    If lngStartLine = 0 Then
        lngStartLine = mdl.CreateEventProc("Load", "Form")
    End If

    intIncrement = 2
    mdl.InsertLines lngStartLine + intIncrement, _
        "'-- Create an instance of the clsBookmarkManagement class"
    intIncrement = intIncrement + 1
    mdl.InsertLines lngStartLine + intIncrement, _
        "    Set bmtForm = New clsBookmarkManagement" & vbCrLf
    intIncrement = intIncrement + 2
    mdl.InsertLines lngStartLine + intIncrement, _
        "'-- Call the custom initialization routine, passing the necessary items."
    intIncrement = intIncrement + 1
    mdl.InsertLines lngStartLine + intIncrement, _
        "    bmtForm.InitBookmarks Me!" & Me!txtBMTName & ", " & conQuotes & _
        Me!cboField1 & conQuotes & IIf(IsNull(Me!cboField2), "", ", " & _
        conQuotes & Me!cboField2) & conQuotes & vbCrLf
End Sub

```

The code in Listing 17.12, after declaring some variables, tries to see whether the text `"Form_Load"` is already in the module. If it doesn't find it, `CreateFormLoadEvent`

creates the event by using the module's `CreateEventProc` method. In either case, `lngStartLine` is updated to the location of the "Form_Load" line so that the routine can then use the `InsertLines` method to add the remaining code lines to the `Form_Load` event procedure.

8. The `CreateComboAfterUpdateEvent` routine does just what its name implies—creates the `AfterUpdate` event procedure for the Bookmark Tracker combo box:

```
Sub CreateComboAfterUpdateEvent(mdl As Module)
    Dim lngStartLine As Long
    lngStartLine = mdl.CreateEventProc("AfterUpdate", Me!txtBMTName)
    mdl.InsertLines lngStartLine + 2, _
        "    '-- Call the BookmarkAction method, for all actions."
    mdl.InsertLines lngStartLine + 3, "    bmtForm.BookmarkAction"
End Sub
```

By now, the statements used in the `CreateComboAfterUpdateEvent` routine should be old hat. It creates the event procedure by using `CreateEventProc`, and then inserts the necessary code lines.

9. The final step in Listing 17.10's `cmdFinish_Click` routine is to close the forms. If `chkDesign` is `True` (now assigned to `blnDesign`), the code reopens the form in Design view; otherwise, it hides the VBE as well.

```
DoCmd.Close acForm, Me!lstForms, acSaveYes
DoCmd.Close acForm, Me.Name
Set frmTarget = Nothing
Set mdl = Nothing

If blnDesign Then
    DoCmd.OpenForm strTarget, acDesign
Else
    Application.VBE.MainWindow.Visible = False
End If
```

NOTE

You might have noticed that the code simply sets the VBE to Not Visible and doesn't close it. If you try to close the VBE while code is executing, you will get an error.

Using Access Code Libraries

Through references and VBA, Access allows you to create your own VBA module libraries. As with add-ins, each library can then be used in multiple or distributed applications.

Looking at the Pros and Cons of Code Libraries

Using code libraries has some advantages:

- Code libraries allow you to create routines once and then reuse them in multiple applications. This reduces redundant code.
- When you make changes to the code for fixes or additions, they are made in only one place.
- If you want to distribute your library to clients without letting them have the source, you can make an MDE from the library, but still give them the source (which they pay for) of the main application.

The following are a few of the disadvantages. Other issues to be aware of are discussed later in the section “Looking at Some Library Coding Issues.”

- Be careful where library databases are placed in the system so that references can be maintained.
- Possible performance issues can arise.
- When you make changes to the library, take precautions against messing up calls from applications that use the library. Using the `Optional` keyword and setting defaults on the way into the routines can take care of this possibility.
- If you use an MDE for the library, issues arise when using it with some non-U.S. versions of Access.
- When you write routines to be used in a library, determine when the code is addressing the application objects instead of the objects located in the library.

Considering Where to Put the Library Database

You should give some consideration to where you place the library database. If you move or distribute the application after the library is referenced, you can break the reference. To have Access “fix up” the reference to the library when starting the application, you can

- Place the library file in the application’s, Access’s, Windows, or Windows\System folders.
- Create a Registry entry with the key `\HKEY_LOCAL_MACHINE\Software\Microsoft\Office\10.0\Access\RefLibPaths`.
- Create a reference yourself by using the `AddFromFile` method of the `References` collection. This code demonstrates this method:

```
Function AddReferenceFromFile(strFileName As String) As Boolean
    Dim refCurr As Reference
    Set refCurr = References.AddFromFile(strFileName)
End Function
```

In this code, `strFileName` is the full name and path of the .mda file. You want to delete the broken reference by using the `Remove` method from the `References` collection.

CAUTION

The problem with this preceding technique is that it's sometimes hard to even get code to run at all when a reference is broken, so the technique's effectiveness is limited.

17

Setting a Reference to a Library

Setting a reference to a library is the same as setting a reference to any other type of library, such as the Word 10.0 Object Library. To show you how to use a reference, I've created a small library with several functions in it. You can download this library, `Chap17lib.mda`, from this book's Web page at www.sampublishing.com.

To set a reference while in the VBE, follow these steps:

1. Choose `References` from the `Tools` menu.
2. Click the `Browse` button. The `Add Reference` dialog box appears.
3. Change the `Files of Type` drop-down list to `Add-Ins (*.mda)`.
4. Select `Chap17lib.mda` in the directory of your choosing (see Figure 17.13).
5. Click `Open`. The library is added to the references.

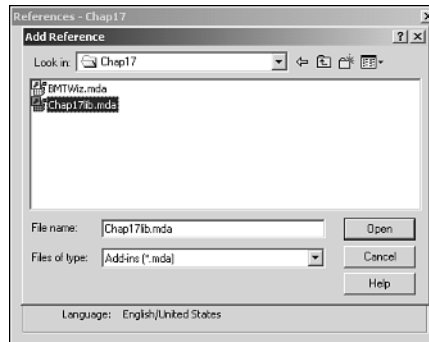


FIGURE 17.13

Choose a library database to reference.

NOTE

You can also reference .MDEs and MDBs.

Viewing Library Routines in the Object Browser

To check out the library's routines, choose Object Browser from the View menu and select Chap17lib from the Project/Libraries list. You then can see the library's modules and routines (see Figure 17.14).

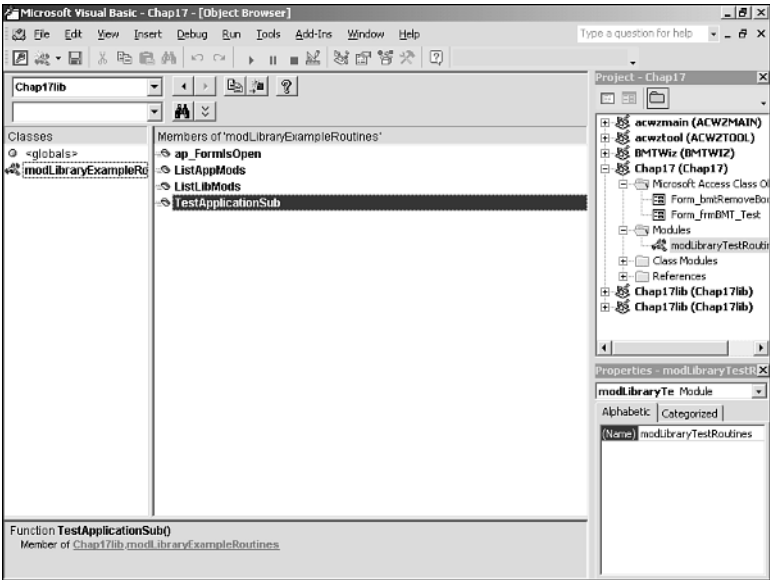


FIGURE 17.14
View routines from the library in the Object Browser.

NOTE

Although you can modify library routines while in the application database, the changes won't be saved and are only temporary. To actually save changes, you must open the library separately, make modifications, and then save the modules.

Looking at Some Library Coding Issues

When working with a library, you need to be aware of how you're creating your code. You can use some objects to make life easier as you build the library.

Using `CurrentProject` to Reference the Application

Access lets you use `CurrentProject` so that you can specify the location of the objects you want to deal with. An example of this is the classic `IsFormOpen` function:

```
Function ap_FormIsOpen(strFormName As String) As Boolean
    ap_FormIsOpen = CurrentProject.AllForms(strFormName).IsLoaded
End Function
```

Specifying `CurrentProject` tells Access exactly which Forms collection you want to deal with. Another example of using the current project is to print out all module names in the application database to the Debug window:

```
Sub ListAppMods()
    Dim aoCurr As AccessObject
    For Each aoCurr In CurrentProject.AllModules
        Debug.Print aoCurr.Name
    Next
End Sub
```

To access objects in the library database, use `CodeProject`.

Using `CodeProject` to Reference the Application

To show you how to use `CodeProject`, I modified the `ListAppMods` routine by simply replacing `CurrentProject` with `CodeProject` and changing the name. Between these two objects, you can clearly identify which database—the application or library—you want to deal with:

```
Sub ListLibMods()
    Dim aoCurr As AccessObject
    For Each aoCurr In CodeProject.AllModules
        Debug.Print aoCurr.Name
    Next
End Sub
```

Executing Application Routines from the Library

Another issue to consider is when you want to run a routine located in the application database from the library database. This can be the case if you have a routine that's run in all your applications but need to perform some custom action. To accomplish this, you can have a standard name for the custom routine and call the routine from the library:

```
Function TestApplicationSub()
    Application.Run "ApplicationSub"
End Function
```

You can also explicitly run routines from the library by using the `Application.Run` command.

Summary

This chapter has been pretty busy with my favorite kind of busy—coding. You can do a lot with creating your own add-ins and libraries. Work on it and who knows—maybe you will be selling them to others. In the meantime, to get more information on some VBA commands and Registry manipulation, check out these chapters:

- Chapter 2, “Coding in Access 2002 with VBA,” discusses some VBA syntax and how to use the DoCmd object to a greater extent.
- Chapter 18, “Manipulating the Registry with VBA,” shows how to work with the Registry with VBA to create any entries needed.

Manipulating the Registry with VBA

CHAPTER

18

IN THIS CHAPTER

- Looking at the Windows Registry's History 544
- Using VBA's Registry Commands 550
- Performing Tasks with the Registry API Calls 555

Working with the Windows Registry can intimidate the average Office developer. Mention the Registry, and developers think immediately of C++-level programming. After developers get their first assignment requiring manipulating the Registry from Access using VBA, many of them try to work on every other task on their plate before tackling it. This doesn't have to be the case. By looking at the different parts of the Registry and discussing what area you are most likely to use, this chapter should take the mystery out of it.

You might be saying to yourself, "I have no reason to mess around with the Registry. Why hassle with it?" After you take a brief look at the Registry's history, you might find some reasons for using the Registry with your applications.

NOTE

This chapter is geared for all levels of experience with the Registry—from those who haven't had experience working with the Registry at all to very experienced. If you've worked with the Registry by using Regedit but haven't programmed it through VBA, you can jump to the section "Using VBA's Registry Commands." If you've performed some tasks with VBA Registry commands but haven't delved into API calls, you might want to jump to "Performing Tasks with Registry API Calls."

Looking at the Windows Registry's History

For 16-bit applications, developers kept track of application-specific options by using INI (*.ini) files, also known as *profiles*. These files were set up so that you could specify sections and items with values assigned. These could be read from within your application to perform tasks or customize your application. Applications such as Access 2.0 would use an INI file to specify

- The workgroup file (system.mdb) to be used
- ISAM driver options
- Jet locking options
- Other system defaults

When you switched over to 32-bit with Windows NT/95, applications started using the Windows Registry.

NOTE

You can still use *.ini files with 32-bit apps—it's just not standard. This way, you can conveniently upgrade rather than start over.

Now that you have an idea of where the Registry comes from, look at some reasons you might want to use the Registry for your own applications.

Using the Windows Registry in Your Applications

Now that you know there's a Registry, and that commercial applications such as Office use it to keep track of things, you're probably wondering why you should hassle with it. One reason is to store custom properties for each application and each user. Access has a custom property page in the Database Properties sheet (choose Database Properties from the File menu), where you can track custom properties for your own purposes. In custom properties, you can keep track of items such as

- Back-end name and location
- Whether the client is using replication
- The application's version number
- Whether the application uses a Jet or SQL Server back end
- Whether a production or development machine is using the application
- The names of the production and development servers
- What type of menu to display for the current user

There are many other items as well, some of which are shown in Figure 18.1.

Other applications store this information differently. Even in Access, many developers store this information in a table. The problem with just storing configuration data or database properties in a table is that you have to worry about overwriting it when you update the application. For example, when you work on an application that uses a SQL Server back end, your development machine might use a different server than the production server used by the client. If the front-end database includes a flag to indicate which server to use (development or production), you must set the flag each time you give a copy to the client. The database obviously can't be kept on the back end because that's what the application is trying to link to. A good solution is the Registry. In Figure 18.2, the same properties stored in the Database Properties sheet are now stored in the Registry.

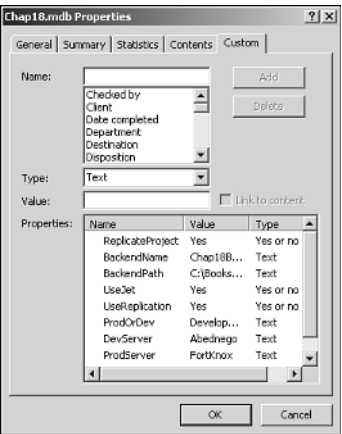


FIGURE 18.1

These are just a few pieces of information that can be stored for an Access application.

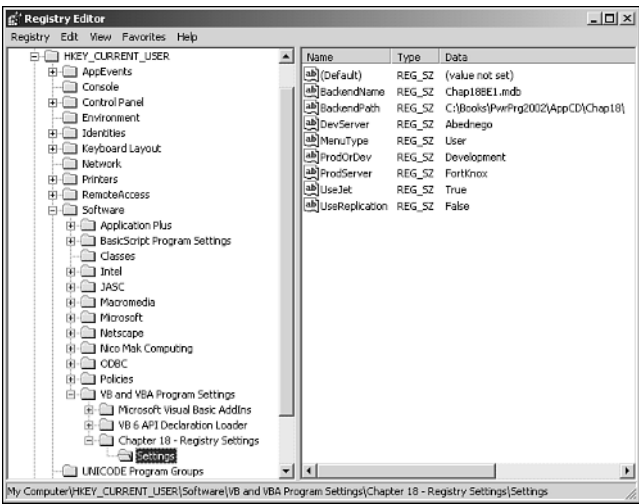


FIGURE 18.2

This information, now stored in the Registry, isn't overwritten the next time a copy of the front-end database is installed.

Parts Making Up the Registry

Technically, the Registry is actually made up of two types of items: keys and values presented in a tree view format, similar to the Explorer. When working with the Registry, you will use two types of keys: predefined keys and subkeys.

Figure 18.2 shows examples of everything discussed here. You can see one key on the left side of Figure 18.2; on the right are multiple values and the default value for the Settings subkey. The tool used to display the Registry in Figure 18.2, Regedit.exe, is part of the tool set included in all 32-bit Windows versions for editing the Registry.

Predefined Keys and Subkeys

Predefined keys are the base keys predefined by Windows and definitely shouldn't be messed with (see Table 18.1). You will utilize the predefined HKEY_CURRENT_USER (shown in Figure 18.2) and HKEY_LOCAL_MACHINE keys for most of the work you'll need to do with your applications.

TABLE 18.1 The Base Keys for Any Part of the Registry

<i>Predefined Key Name</i>	<i>Description</i>
HKEY_CLASSES_ROOT	Tracks the associations for file extensions. Also stores <i>GUIDs (Global Unique Identifiers)</i> for class objects such as ActiveX controls.
HKEY_CURRENT_USER	Changes contents depending on the current logged-on user. Set values here when you have multiple people using the machine and want your application to save settings for each user.
HKEY_LOCAL_MACHINE	Uses the same settings for all users on this machine.
HKEY_USERS	Stores information for network settings, including dialup accessing and connections.
HKEY_CURRENT_CONFIG	Stores configuration information regarding Windows itself.
HKEY_DYN_DATA	Tracks various information, such as performance statistics for applications.

A *subkey* is simply a child key. All keys below the predefined keys are subkeys. You will see plenty of examples of subkeys throughout this chapter.

Values

You can set values for every subkey, but not for the predefined keys. Keys can also have multiple values. Every key also has a default value, which you can set. You specify the value's name, data type, and data. Later in the chapter, you see how to create, update, and delete keys and values in the Registry.

Tools Used For Working with the Registry

As mentioned earlier, all 32-bit versions of Windows use Regedit.exe to edit the Registry. Choose Run from the Start menu, type **Regedit**, and click OK to display the Registry Editor (see Figure 18.3).

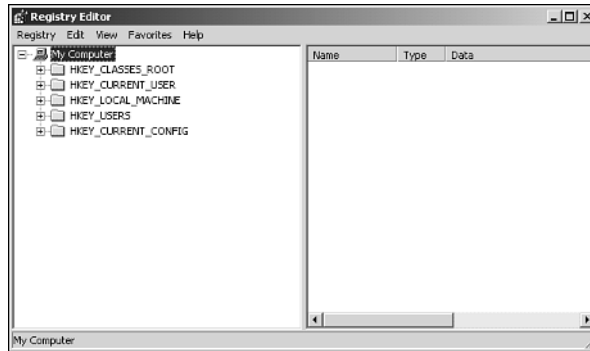


FIGURE 18.3

You might want to create a shortcut if you need to open the Registry Editor frequently.

In the Registry Editor for keys, subkeys, and individual values, you can

- Add
- Delete
- Edit
- Find text and numbers within them (use this when you want to find some values you enter through code later in this chapter)
- Import and export part or the entire Registry

The last feature allows you to restore the Registry from a previously saved version. It's a good idea to export the current settings before making major changes to the Registry.

CAUTION

Be very careful before modifying any part of the Registry. Otherwise, you can really mess up your system. That said, as soon as you understand the commands for modifying the Registry and know what you're doing, working with it is a piece of cake.

Before performing any of these tasks, in Regedit choose Export Registry File from the Registry menu. Then supply a filename for the Registry file you're going to create, and click OK.

Snoop around a bit to get comfortable with the Registry. Changing values at this time probably isn't a good idea, but you can find the Access settings starting at the HKEY_CURRENT_USER predefined key:

1. Highlight the HKEY_CURRENT_USER key.
2. Choose Find from the Edit menu.
3. Type **Access** in the Find What text box.
4. Deselect the Values and Data check boxes.
5. Select the Match Whole String Only check box. The Find dialog should now resemble Figure 18.4.

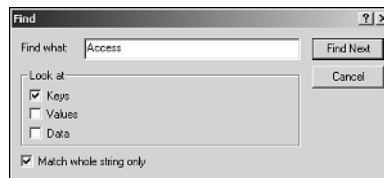


FIGURE 18.4

Regedit's Find dialog helps you find information in keys.

6. Click Find Next. You are taken to the first instance of the word Access in the tree, which happens to be the one you want. The key for the Access application entry is under Office in the subfolder 10.0. Also note that when the key is found, it is not expanded, so you can't see settings right away, and if more than a single version of Office/Access is installed, 10.0 will be the last one to show, vs. 8.0 or 9.0 (97 and 2000). Then click the Settings key to see the options set for Access on the right.

To add keys and values, right-click the key you want to add the key or value to, and then choose New and the type of item you want to add. This can be a key or the various types of values: Binary, String, or DWORD (numeric). You can also edit, rename, and copy keys and values from this right-click menu.

You will get much experience with Regedit as you work with the Registry. Let's look at the other tool used—Regsvr32.exe. You use Regsvr32 to add and delete entries in the Registry for ActiveX components, DLLs, and other system files that accompany tools and applications.

This process is called *registering* or *unregistering* the files. Although most applications perform this as a step in their setup programs, in some cases you have to do this yourself.

TIP

After you register and unregister any files, reboot the machine so that Windows includes the entry in the Registry.

To register a DLL in Regsvr32, choose Run from the Start menu. Then type **Regsvr32** and drag and drop the file from the Windows Explorer (if it's open). Basically, you need to indicate the full name and path of the DLL, using the following syntax:

```
Regsvr32 Path\Name.Ext
```

To unregister a file, place the command-line argument `/u` before the filename.

Some files and utilities have an accompanying file with a `.reg` extension. If you double-click this file, Windows will register all the files listed in it.

Using VBA's Registry Commands

VBA includes a number of commands to give developers Registry access. They are pretty straightforward to use when you have an idea how to work with the Registry, which at this point you should.

Six commands deal with the Registry; Table 18.2 lists four.

TABLE 18.2 Registry-Manipulating Commands in VBA

<i>Command</i>	<i>Purpose</i>
DeleteSetting	Deletes a key or value from an application's entry in the Registry or 16-bit INI file.
GetSetting	Retrieves a key or value from an application's entry in the Registry or 16-bit INI file.
GetAllSettings	Returns a list of key settings and their respective values from an application's entry in the Registry or 16-bit INI file, and places the information in a two-dimensional array.
SaveSetting	Saves or creates an application entry in the Registry or 16-bit INI file.

NOTE

Although these commands are available to all VBA development environments, you also can use some commands specifically for setting Access options:

- The `SetOption` statement sets individual options in Access (refer to Figure 18.4). To see more about this command, check out the help in Access.
- An *individual user profile* is a special set of Registry keys that you can create to override standard Access and Jet database engine settings as well as specify additional runtime options. To use a user profile when opening an Access database, specify `/profile` followed by the user profile filename in the command line. Look up profiles in the Access help for more on this.

A number of commands also work with Jet objects and references. You can use these commands in all VBA environments when working with these objects.

Let's first discuss the `SaveSetting` statement from Table 18.2. In VBA you can use `SaveSetting` to store the default settings you want to create for your application (see Listing 18.1). This routine, part of the `modVBAREgRoutines` module, can be found in `Chap18.mdb`, in the `\Examples\Chap18` folder on the book's Web page at www.sampublishing.com.

LISTING 18.1 Chap18.mdb: Creating the Default Registry Settings

```
Sub CreateRegSettingDefaults()
    ' Place some settings in the Registry.
    SaveSetting appname:="Chapter 18 - Registry Example App", _
        Section:="Settings", key:="UseJet", setting:=True
    SaveSetting appname:="Chapter 18 - Registry Example App", _
        Section:="Settings", key:="BackendName", setting:="Chap18BE1.mdb"
    SaveSetting appname:="Chapter 18 - Registry Example App", _
        Section:="Settings", key:="BackendPath", _
        setting:="c:\Books\PwrPrg2002\AppCD\Chap18\"
    SaveSetting appname:="Chapter 18 - Registry Example App", _
        Section:="Settings", key:="UseReplication", setting:=False
    SaveSetting appname:="Chapter 18 - Registry Example App", _
        Section:="Settings", key:="ProdOrDev", setting:="Development"
    SaveSetting appname:="Chapter 18 - Registry Example App", _
        Section:="Settings", key:="ProdServer", setting:="Fortknnox"
    SaveSetting appname:="Chapter 18 - Registry Example App", _
        Section:="Settings", key:="DevServer", setting:="Abednego"
    SaveSetting appname:="Chapter 18 - Registry Example App", _
        Section:="Settings", key:="MenuType", setting:="User"
End Sub
```

PART III

Normally, you would track default values by using a more data-driven method, rather than the hard-coded values shown here. You will see this method in a moment. However, running the `CreateRegSettingsDefaults` routine creates the Registry entries shown in Figure 18.5.

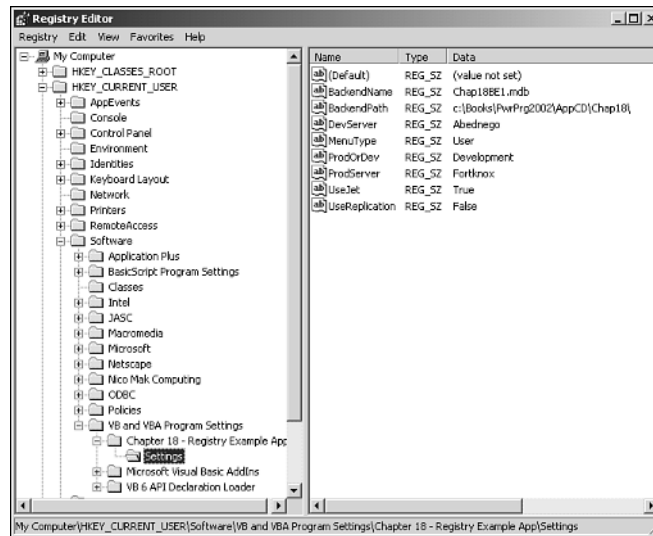


FIGURE 18.5

The HKEY_CURRENT_USER\Software\VB and VBA Program Settings path is where all settings are affected using Registry commands in VBA.

Take this a step further by using Access to hold initial values that will be stored in the Registry. The routine in Listing 18.2 deletes all settings for the application, cycles through a table, and generates the defaults. Figure 18.6 shows the values stored in the `tblDefaultSettings` table.

LISTING 18.2 Chap18.mdb: Removing and Creating Settings for the Current Application from a Table

```
Sub CreateAllRegSettingsFromTable()
    Dim rsDefaults As New ADODB.Recordset
    Dim strAppName As String
    Dim strSetting As String

    On Error GoTo Err_CreateAll

    rsDefaults.Open "tblDefaultSettings", CurrentProject.Connection
```

LISTING 18.2 Continued

```

'-- Get the application title to use as a key
strAppName = CurrentDb.Properties("AppTitle")

On Error Resume Next

'-- Remove all the settings
DeleteSetting strAppName, "Settings"

On Error GoTo Err_CreateAll

'-- Re-create all the settings.
With rsDefaults
    Do Until .EOF
        SaveSetting appname:=strAppName, Section:="Settings", _
            Key:=!SettingTitle, setting:=!DefaultValue
        .MoveNext
    Loop
End With

Exit_CreateAll:
Exit Sub

Err_CreateAll:
MsgBox Err.Description
Resume Exit_CreateAll

End Sub

```

18**MANIPULATING
THE REGISTRY
WITH VBA**

SettingTitle	DefaultValue
BackendName	Chap18BE.mdb
BackendPath	c:\Books\PwrPrg2002\AppCD\Chap18\
DefaultView	Student
DevServer	Abadnego
MenuType	User
ProdOrDev	Development
ProdServer	FortKnox
UseJet	True
UseReplication	False

FIGURE 18.6

These values will soon be stored in the Registry.

This routine uses two Registry commands: `DeleteSetting` and `SaveSetting`. It works great when you want to re-create all default settings for an application. If you want only to add

PART III

settings and not overwrite current settings, however, you need to modify the routine to add the `GetSetting` command. Listing 18.3 shows how this is done in the routine `CreateNewSettingsFromTable`.

LISTING 18.3 Chap18.mdb: Adding Only New Settings to the Registry

```
Sub CreateNewRegSettingsFromTable()
    Dim rsDefaults As New ADODB.Recordset
    Dim strAppName As String
    Dim strSetting As String

    On Error GoTo Err_CreateNew

    rsDefaults.Open "tblDefaultSettings", CurrentProject.Connection

    '-- Get the application title to use as a key
    strAppName = CurrentDb.Properties("AppTitle")

    '-- Create settings for those defaults where there is no entry
    With rsDefaults
        Do Until .EOF
            strSetting = GetSetting(appname:=strAppName, Section:="Settings", _
                                   Key:=!SettingTitle)
            If Len(strSetting) = 0 Then
                SaveSetting appname:=strAppName, Section:="Settings", _
                           Key:=!SettingTitle, Setting:=!DefaultValue
            End If
            .MoveNext
        Loop
    End With

Exit_CreateNew:
    Exit Sub

Err_CreateNew:
    MsgBox Err.Description
    Resume Exit_CreateNew

End Sub
```

This routine is useful when you're updating an application and don't want to overwrite current settings, but want to add to the defaults used by the system. Also, when using the application on the same machine, users can record the default settings in the Registry without overwriting other users' settings because VBA puts the entries under the `HKEY_CURRENT_USER` key.

To use the defaults recorded in the Registry, you use only the `GetSetting` command with the name of the value you want. A wrapper function, `GetDefaultSetting()`, performs this task:

```
Public Function GetDefaultSetting(strNameOfSetting As String) As String
    GetDefaultSetting = GetSetting(AppName:=CurrentDb.Properties("AppTitle"), _
        Section:="Settings", Key:=strNameOfSetting)
End Function
```

NOTE

You create VBA wrapper routines to simplify the calling of more complex routines and commands, including API calls. These routines are discussed in more detail in the next section.

Rounding out the VBA routines for simple Registry manipulation are a couple that update and delete individual settings. Note that the setting would be added again if any create setting routines are called.

```
Public Sub SetDefaultSetting(strNameOfSetting As String, strValue As String)
    SaveSetting AppName:=CurrentDb.Properties("AppTitle"), _
        Section:="Settings", Key:=strNameOfSetting, Setting:=strValue
End Sub
```

```
Public Sub DeleteDefaultSetting(strNameOfSetting As String)
    DeleteSetting AppName:=CurrentDb.Properties("AppTitle"), _
        Section:="Settings", Key:=strNameOfSetting
End Sub
```

Performing Tasks with Registry API Calls

Although the routines in the preceding section perform simple tasks to manage defaults, they have a number of limitations, including the following:

- All settings are maintained in the `HKEY_CURRENT_USER\Software\VB and VBA Program Settings` key path. Sometimes you might not want to have the settings in this path.
- If your path is more complicated and you need to include more keys and values with different branches, using the straight VBA commands can be somewhat cumbersome.

To overcome these limitations, you use API calls to the Registry. More specifically, you can use the wrapper routines supplied with the sample database (so you don't have to worry too much about hurting yourself). Remember that VBA wrapper routines set up variables needed

for calling API routines. They make life much easier, especially because you can usually cut and paste the routines into your own application without knowing all that's going on behind the API routine.

API calls can easily perform all the tasks that the built-in VBA commands can do, as well as many more. The rest of this chapter covers some of these tasks:

- Creating a new Registry key
- Checking to see whether a Registry key already exists
- Creating a new Registry value
- Updating a Registry value
- Deleting a key from the Registry
- Retrieving a value from the Registry
- Retrieving all subkeys for a given Registry key

To demonstrate these tasks, I've created a utility in Access that lets you relink a front-end database to various back-end databases. These back ends are tracked by using the Registry. Figure 18.7 shows the main form used in the example.

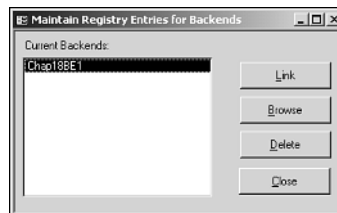


FIGURE 18.7

This form lets you browse for a database to link to, link to current databases, and delete (from the Registry) a database from the list.

The following are just a few examples of applications that could use this utility:

- Any application dealing with multiple companies for accounting purposes. The companies might need to have their data separate from each other for security reasons, but the same application would be used for maintaining the information.
- Mailing-list programs that use various data sets from multiple sources.
- Data analysis applications.

NOTE

Most database application development is performed by using a single back-end database. However, more and more I've needed to work with more than one back end for the same front end. When discussing multiple back ends in this chapter, I mean that the same back end is used for all tables at one time—however, the ability to switch which back end is being used at a given time is what's being covered.

Looking at the Sample Application

As you look at the sample application, first notice that the only Access objects used are a table and form; the rest of the example consists of VBA code. Figure 18.8 shows the `tblLinkedTables` table. Figure 18.9 shows the form used for the sample application, `frmMaintainBERegistryEntries`, in Design view.



FIGURE 18.8
tblLinkedTables lists the tables to be linked to in the various back ends.

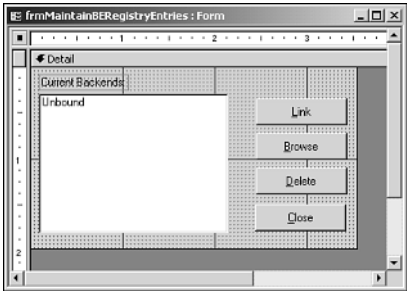


FIGURE 18.9
This form tracks multiple back ends.

The form contains one unbound list box control, `lstBackends`, which lists back ends now in the Registry. The form also uses the following controls, all command buttons:

- The Link button, `cmdLink`, calls code that links the selected back end in the `lstBackends` list box.
- The Browse button, `cmdBrowse`, calls the wrapper routine that calls the API Open File dialog, so users can browse to a new back-end database to add to the Registry list. (Users must click the Link button to actually link to the database.)

NOTE

The Browse button's routine doesn't check the validity of the back-end database. If you plan to use this utility for your own purposes, you should add some code to verify the back end.

- The Delete button, `cmdDelete`, deletes the Registry entry for the currently selected back end in `lstBackends`.
- The Close button, `cmdClose`, closes the `frmMaintainBERegistryEntries` form.

The rest of the application consists of VBA code, stored in standard modules:

- The `modUtilityRoutines` module stores general routines used throughout the application. I won't discuss these routines in detail because they have nothing to do with the Registry; they just support the overall application.
- The `modOpenFileAPIRoutines` module stores the code for the `OpenFile` API call.
- The `modRegistryAPIRoutines` module stores wrapper routines for the actual API call to the Registry. Although these routines are used for a specific purpose in this application, you can copy this module out into any other application that uses VBA and call these routines.
- The routines in the `modBERegistryRoutines` module are customized for tracking the multiple back ends in this application. They call the routines in the `modRegistryAPIRoutines` module.

Working with the Actual Code

Although it normally would be logical to look at the load event when looking at the code behind a form, in this case you'll examine the code behind the `cmdBrowse` command button. It contains code for allowing users to pick the back end to link and then update the application's Registry structure.

Browsing for a Back End to Track

The cmdBrowse button uses the click event to perform two tasks (discussed shortly). First, Listing 18.4 shows the code for cmdBrowse_Click.

LISTING 18.4 Chap18.mdb: The Windows Open File Dialog and Creating the Registry Entries

```
Private Sub cmdBrowse_Click()
    Dim strFileName As String

    strFileName = ap_OpenFile(, "Pick a Backend Database to register")

    If Len(strFileName) <> 0 Then
        AddBEToRegistry strFileName
        Me!lstBackends.RowSource = _
            GetRegistrySubKeys(HKEY_CURRENT_USER, MyAppBEsKey)
    End If
End Sub
```

First, the cmdBrowse_Click routine gets the full file path and name of the database that you want to track with the Registry and link to. (Note that it doesn't actually perform the link; this is done by using the Link command button.) The full file path and name is retrieved from the user through the ap_OpenFile function, which is located in modOpenFileRoutine and shown entirely in Listing 18.5.

LISTING 18.5 Chap18.mdb: Calling the Open File Dialog

```
Option Compare Text
Option Explicit

Private Declare Function ap_GetOpenFileName Lib "comdlg32.dll" _
    Alias "GetOpenFileNameA" (pOpenfilename As OPENFILENAME) As Long

Private Type OPENFILENAME
    lStructSize As Long
    hwndOwner As Long
    hInstance As Long
    lpstrFilter As String
    lpstrCustomFilter As String
    nMaxCustFilter As Long
    nFilterIndex As Long
    lpstrFile As String
    nMaxFile As Long
```

LISTING 18.5 Continued

```

    lpstrFileName As String
    nMaxFileName As Long
    lpstrInitialDir As String
    lpstrTitle As String
    flags As Long
    nFileOffset As Integer
    nFileExtension As Integer
    lpstrDefExt As String
    lCustData As Long
    lpfnHook As Long
    lpTemplateName As String
End Type

Const cdIOFNAAllowMultiselect = &H200
Const cdIOFNCreatePrompt = &H2000
Const cdIOFNExplorer = &H80000
Const cdIOFNExtensionDifferent = &H400
Const cdIOFNFileMustExist = &H1000
Const cdIOFNHelpButton = &H10
Const cdIOFNHideReadOnly = &H4
Const cdIOFNLongNames = &H200000
Const cdIOFNNoChangeDir = &H8
Const CdIOFNNoDereferenceLinks = &H100000
Const cdIOFNNoLongNames = &H40000
Const CdIOFNNoReadOnlyReturn = &H8000
Const cdIOFNNoValidate = &H100
Const cdIOFNOverwritePrompt = &H2
Const cdIOFNPathMustExist = &H800
Const cdIOFNReadOnly = &H1
Const CdIOFNShareAware = &H4000

Public Function ap_OpenFile(Optional ByVal strFileNameIn _
    As String = "", Optional strDialogTitle As String = "Name of File to Open")

    Dim lngReturn As Long
    Dim intLocNull As Integer
    Dim strTemp As String
    Dim ofnFileInfo As OPENFILENAME
    Dim strInitialDir As String
    Dim strFileName As String

    '-- if a file path passed in with the name, parse it and split it off.
    If InStr(strFileNameIn, "\") <> 0 Then
        strInitialDir = Left(strFileNameIn, InStrRev(strFileNameIn, "\"))
    
```

LISTING 18.5 Continued

```

    strFileName = Left(Mid$(strFileNameIn, _
        InStrRev(strFileNameIn, "\") + 1) & String(256, 0), 256)
Else
    strInitialDir = Left(CurrentDb.Name, InStrRev(CurrentDb.Name, "\") - 1)
    strFileName = Left(strFileNameIn & String(256, 0), 256)
End If

With ofnFileInfo
    .lStructSize = Len(ofnFileInfo)
    .lpstrFile = strFileName
    .lpstrFileTitle = String(256, 0)
    .lpstrInitialDir = strInitialDir
    .hwndOwner = Application.hWndAccessApp
    .lpstrFilter = "All Files (*.*)" & Chr(0) & "*.*" & Chr(0)
    .nFilterIndex = 1
    .nMaxFile = Len(strFileName)
    .nMaxFileTitle = ofnFileInfo.nMaxFile
    .lpstrTitle = strDialogTitle
    .flags = cd1OFNFileMustExist Or cd1OFNHideReadOnly Or cd1OFNNoChangeDir
    .hInstance = 0
    .lpstrCustomFilter = String(255, 0)
    .nMaxCustFilter = 255
    .lpfnHook = 0
End With

lngReturn = ap_GetOpenFileName(ofnFileInfo)

If lngReturn = 0 Then
    strTemp = ""
Else
    '-- Trim off any null string
    strTemp = Trim(ofnFileInfo.lpstrFile)
    intLocNull = InStr(strTemp, Chr(0))
    If intLocNull Then
        strTemp = Left(strTemp, intLocNull - 1)
    End If
End If

ap_OpenFile = strTemp

End Function

```

TIP

Notice that the structure used in this module has a distinct C flavor to it. As with many API calls and the wrapper functions created, they can originally be from another language and modified for use in your own functions. I tend to leave API functions and wrappers as close to the original form as possible while modifying them for my own needs. This leads to less “complications” when using them.

The routine in Listing 18.5 takes the structure created in the declarations area, sets it up with the desired defaults, and then calls the `ap_GetOpenFileName` API routine (my `Alias`). Figure 18.10 shows the Open File dialog in action.

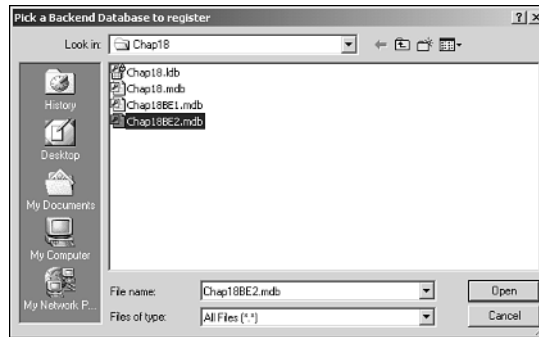


FIGURE 18.10

The second of the two back ends used for this example, Chap18BE2.mdb, is selected in the Open File dialog.

NOTE

The Common Dialogs ActiveX control isn't used in this example. Although ActiveX controls can be more convenient to use than API calls, users must have the *same version* of an ActiveX control on their system, which isn't the case in many installations.

Registering the Database

The sample application registers the back-end database into the application entry in the Registry, and stores the following pieces of information about the database:

- Filename only
- Filename and extension

- File path
- Full filename and path

Figure 18.11 shows Chap18BE2.mdb's back-end information as it's stored in the Registry. All this information was parsed from the file path and name retrieved, and then stored into `strFileName` in the `cmdBrowse_Click` routine. The following code calls the routine to add the entries:

```
AddBEToRegistry strFileName
```

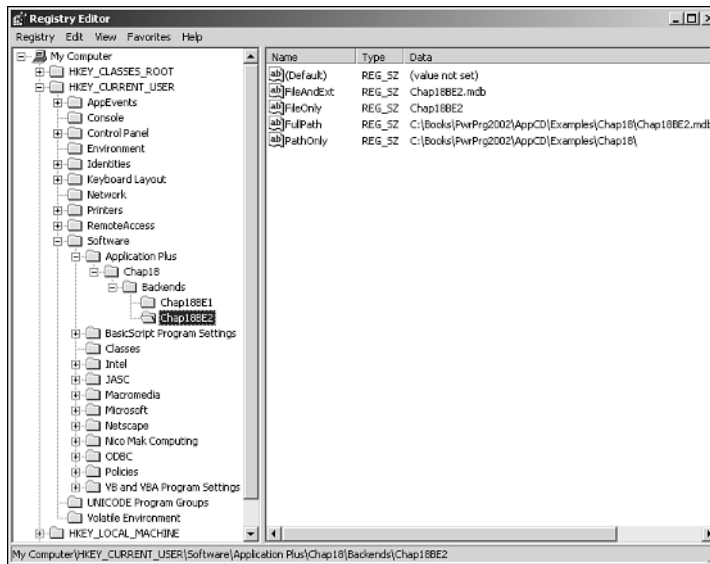


FIGURE 18.11

Storing the information in bite-size pieces makes it more convenient to use later in the application.

The `AddBEToRegistry` routine is located in the `modBERegistryRoutines` module, which was custom created to manipulate Registry entries for this particular application. You can see the `AddBEToRegistry` subroutine in Listing 18.6.

LISTING 18.6 Chap18.mdb: Parsing the Filename and Path to Add Entries to the Registry

```
Sub AddBEToRegistry(strFileName As String)
    Dim strFileAndExt As String
    Dim strFileOnly As String
    Dim strPathOnly As String

    '-- Parse all the pertinent information out of the full path & filename
    strFileAndExt = Mid$(strFileName, ap_LastInStr(strFileName, "\") + 1)
```


LISTING 18.6 Continued

```

strFileOnly = Left$(strFileAndExt, Len(strFileAndExt) - 4)
strPathOnly = Left$(strFileName, InStr(strFileName, strFileAndExt) - 1)

'-- Create a key based on the file name
CreateNewRegKey HKEY_CURRENT_USER, MyAppBEsKey & "\" & strFileOnly

'-- Record the individual values you want to track for the key.
SetRegKeyValue HKEY_CURRENT_USER, MyAppBEsKey & "\" & strFileOnly, _
    "FullPath", strFileName, REG_SZ
SetRegKeyValue HKEY_CURRENT_USER, MyAppBEsKey & "\" & strFileOnly, _
    "FileAndExt", strFileAndExt, REG_SZ
SetRegKeyValue HKEY_CURRENT_USER, MyAppBEsKey & "\" & strFileOnly, _
    "FileOnly", strFileOnly, REG_SZ
SetRegKeyValue HKEY_CURRENT_USER, MyAppBEsKey & "\" & strFileOnly, _
    "PathOnly", strPathOnly, REG_SZ

End Sub

```

Before going through this code step by step, check out a few constants:

- HKEY_CURRENT_USER represents the value of the predefined HKEY_CURRENT_USER key and can be found in the declaration section of the modRegistryAPIRoutines module. The module declares these predefined keys:

```

Public Const HKEY_CURRENT_USER = &H80000001
Public Const HKEY_CLASSES_ROOT = &H80000000
Public Const HKEY_LOCAL_MACHINE = &H80000002
Public Const HKEY_USERS = &H80000003

```

NOTE

Again, for most of your applications, you should use just HKEY_CURRENT_USER or HKEY_LOCAL_MACHINE, depending on whether you want settings for each individual user or each machine.

- MyAppBEsKey represents the Registry path for the sample application's values. This line of code, placed in a general module's Declaration section, performs this:

```
Public Const MyAppBEsKey = "Software\Applications Plus\Chap18\Backends"
```
- REG_SZ is one of several constants, again located in modRegistryAPIRoutines, used for defining the data type for the value you're creating in the Registry. Here is the whole list of declarations found there:

```
Global Const REG_SZ As Long = 1
Global Const REG_EXPAND_SZ As Long = 2
Global Const REG_BINARY As Long = 3
Global Const REG_DWORD As Long = 4
```

Note that more Registry types can be used but aren't necessarily recommended.

The routine in Listing 18.6 performs the following tasks:

1. It parses all the pertinent information out of the full path and filename:

```
strFileAndExt = Mid$(strFileName, ap_LastInStr(strFileName, "\") + 1)
strFileOnly = Left$(strFileAndExt, Len(strFileAndExt) - 4)
strPathOnly = Left$(strFileName, InStr(strFileName, strFileAndExt) - 1)
```

2. The routine creates a key based on the filename:

```
CreateNewRegKey HKEY_CURRENT_USER, MyAppBEsKey & "\" & strFileOnly
```

In this case, it creates the key Chap18BE2; if the rest of the branch didn't exist, it would be created. Because CreateNewRegKey is the first API call discussed, Listing 18.7 shows it in its entirety.

LISTING 18.7 Chap18.mdb: Creating a New Registry Key with the Passed Arguments

```
Public Sub CreateNewRegKey(lngPredefinedKey As Long, strNewKeyName As String)
    Dim hNewKey As Long          'handle to the new key
    Dim lRetVal As Long          'result of the RegCreateKeyEx function
    lRetVal = RegCreateKeyEx(lngPredefinedKey, strNewKeyName, 0&, _
        vbNullString, REG_OPTION_NON_VOLATILE, KEY_WRITE, 0&, hNewKey, lRetVal)
    RegCloseKey (hNewKey)
End Sub
```

NOTE

Another reason for using wrapper routines already created for you is that you must call the RegCloseKey routine to clear the reference to the key that you opened by calling RegCreateKeyEx(). If you didn't know this, you would start getting errors or just flaky results.

The RegCreateKeyEx() API function is one of many API routines supplied in the sample application. They are discussed in the next section.

3. Finally, the routine records the individual values you want to track for the key:

```
SetRegKeyValue HKEY_CURRENT_USER, MyAppBEsKey & "\" & _
    strFileOnly, "FullPath", strFileName, REG_SZ
```

```

SetRegKeyValue HKEY_CURRENT_USER, MyAppBESKey & "\" & _
    strFileOnly, "FileAndExt", strFileAndExt, REG_SZ
SetRegKeyValue HKEY_CURRENT_USER, MyAppBESKey & "\" & _
    strFileOnly, "FileOnly", strFileOnly, REG_SZ
SetRegKeyValue HKEY_CURRENT_USER, MyAppBESKey & "\" & _
    strFileOnly, "PathOnly", strPathOnly, REG_SZ

```

Step 3 calls the `SetRegKeyValue` routine, another API wrapper in the `modRegistryAPIRoutines` module. With this routine, as with the other API wrapper routines, it's not necessary to go through step by step what it does in setting up and calling the API routine. In this case, it's known that the `SetRegKeyValue` routine takes the parameters passed to it and creates the individual Registry values.

In the following section, you can see `RegCreateKeyEx()` and other API declarations used in the sample application, as well as others you might find useful your own applications.

Listing the API Declarations and Wrapper Routines for the Registry

The API declarations in Listing 18.8 can be found in `modRegistryAPIRoutines`. After going through the wrapper routines in `modRegistryAPIRoutines`, you should have a good idea of what it takes to work with the API calls for the Registry.

LISTING 18.8 Chap18.mdb: Declaring API Routines Before Using Them

```

Declare Function RegCloseKey Lib "advapi32.dll" (ByVal hKey As Long) As Long
Declare Function RegCreateKeyEx Lib "advapi32.dll" Alias _
    "RegCreateKeyExA" (ByVal hKey As Long, ByVal lpSubKey As String, _
    ByVal Reserved As Long, ByVal lpClass As String, ByVal dwOptions As Long, _
    ByVal samDesired As Long, ByVal lpSecurityAttributes As Long, _
    phkResult As Long, lpdwDisposition As Long) As Long
Declare Function RegOpenKeyEx Lib "advapi32.dll" Alias _
    "RegOpenKeyExA" (ByVal hKey As Long, ByVal lpSubKey As String, _
    ByVal ulOptions As Long, ByVal samDesired As Long, phkResult As Long) _
    As Long
Declare Function RegQueryValueExString Lib "advapi32.dll" Alias _
    "RegQueryValueExA" (ByVal hKey As Long, ByVal lpValueName As String, _
    ByVal lpReserved As Long, lpType As Long, ByVal lpData As String, _
    lpcbData As Long) As Long
Declare Function RegQueryValueExLong Lib "advapi32.dll" Alias _
    "RegQueryValueExA" (ByVal hKey As Long, ByVal lpValueName As String, _
    ByVal lpReserved As Long, lpType As Long, lpData As Long, _
    lpcbData As Long) As Long
Declare Function RegQueryValueExNULL Lib "advapi32.dll" Alias _
    "RegQueryValueExA" (ByVal hKey As Long, ByVal lpValueName As String, _
    ByVal lpReserved As Long, lpType As Long, ByVal lpData As Long, _
    lpcbData As Long) As Long

```

LISTING 18.8 Continued

```
Declare Function RegSetValueExString Lib "advapi32.dll" Alias _
    "RegSetValueExA" (ByVal hKey As Long, ByVal lpValueName As String, _
        ByVal Reserved As Long, ByVal dwType As Long, ByVal lpValue As String, _
        ByVal cbData As Long) As Long
Declare Function RegSetValueExLong Lib "advapi32.dll" Alias _
    "RegSetValueExA" (ByVal hKey As Long, ByVal lpValueName As String, _
        ByVal Reserved As Long, ByVal dwType As Long, lpValue As Long, _
        ByVal cbData As Long) As Long
Declare Function RegQueryInfoKey Lib "ADVAPI32" Alias _
    "RegQueryInfoKeyA" (ByVal hKey As Long, ByVal lpszClass As String, _
        ByRef lpcchClass As Long, ByRef lpdwReserved As Long, _
        ByRef lpcSubKeys As Long, ByRef lpcchMaxSubkey As Long, _
        ByRef lpcchMaxClass As Long, ByRef lpcValeus As Long, _
        ByRef lpcchMaxValueName As Long, ByRef lpcbMaxValueData As Long, _
        ByRef lpcbSecurityDescriptor As Long, _
        ByRef lpftLastWriteTime As FILETIME) As Long
Declare Function RegEnumKeyEx Lib "ADVAPI32" Alias _
    "RegEnumKeyExA" (ByVal hKey As Long, ByVal iSubkey As Long, _
        ByVal lpszName As String, ByRef lpcchName As Long, _
        ByVal lpdwReserved As Long, ByVal lpszClass As String, _
        ByRef lpcchClass As Long, ByRef lpftLastWrite As FILETIME) As Long
Declare Function RegDeleteKey Lib "advapi32.dll" Alias "RegDeleteKeyA" _
    (ByVal hKey As Long, ByVal lpSubKey As String) As Long
Declare Function RegDeleteValue Lib "advapi32.dll" Alias _
    "RegDeleteValueA" (ByVal hKey As Long, ByVal lpSubValue As String) As Long
```

NOTE

More API declarations are included than are used for this example. This is to save you the trouble of hunting them down yourself.

Table 18.3 lists the wrapper functions created and used in the sample application.

TABLE 18.3 Even the Wrapper Functions Call Other Functions

Wrapper Routine	Purpose
CreateNewRegKey	Deletes a key or value from an application
DeleteAllRegEntriesForKey	Deletes all entries for a given key
GetRegistrySubKeys	Retrieves all subkeys for the given Registry key
QueryRegValue	Sets up variables and calls QueryValueEx

TABLE 18.3 Continued

<i>Wrapper Routine</i>	<i>Purpose</i>
QueryValueEx	Returns the value from the Registry; called by QueryRegValue
SetRegKeyValue	Set up variables and calls SetValueEx
SetValueEx	Actually sets the Registry value; called by
SetRegKeyValue	

A couple of these routines have already been discussed, and more will be shown as we move on.

Getting a Subkey List for a Registry Key

The last task performed in the routine in Listing 18.4 hasn't yet been discussed. The last task is to populate the `lstBackends` list box with a list of back ends from the Registry, as shown in this code line:

```
Me.lstBackends.RowSource = _
    GetRegistrySubKeys(HKEY_CURRENT_USER, MyAppBEsKey)
```

This code calls the `GetRegistrySubKeys()` wrapper function to perform this task. This function iterates through a given key's subkeys, concatenating them into a string separated by semicolons.

Well, that's it for the first command button. Fortunately, this is the most complicated command button as well. By looking at the form's `Load` event, you can see that the only task that occurs is the loading of the list box:

```
Private Sub Form_Load()
    Me.lstBackends.RowSource = GetRegistrySubKeys(HKEY_CURRENT_USER, MyAppBEsKey)
End Sub
```

Linking the Tables

The next routine to look at is the `cmdLink` command button's click event (see Listing 18.9).

LISTING 18.9 Chap18.mdb: Setting the Current Back End and Linking the Tables

```
Private Sub cmdLink_Click()
    If IsNull(Me.lstBackends) Then
        MsgBox "Please select a backend to link", vbInformation, ap_AppTitle()
        Me.lstBackends.SetFocus
    Else
        SetCurrentBE Me.lstBackends
        LinkTables CurrBENAMEAndPath()
```

LISTING 18.9 Continued

```

        DoCmd.Close acForm, Me.Name
    End If
End Sub

```

The `cmdLink_Click` routine first checks to see whether an item is selected in the list box. If it is, three routines are then called, one of which, `CurrBENameAndPath()`, is passed as a parameter.

Here is the first routine called at this point, `SetCurrentBE`:

```

Sub SetCurrentBE(strCurrBE As String)
    SetRegKeyValue HKEY_CURRENT_USER, MyAppBEsKey, "CurrentBE", _
        strCurrBE, REG_SZ
End Sub

```

This routine, in the `modBERegistryRoutines` module, sets the `CurrentBE` value to the path specifying where the current back end is linked. This way, you can later refer to this value whenever you need to access any other values for the current back end.

The next routine, `LinkTables`, really has nothing to do with Registry entries, but is shown in Listing 18.10. This routine can be found in `modUtilityRoutines`.

LISTING 18.10 Chap18.mdb: Creating or Reconnecting Links

```

Public Sub LinkTables(strDataMDB As String)
    Dim catCurr As New ADOX.Catalog
    Dim rstSharedTables As New ADODB.Recordset
    Dim intTotalTbls As Integer
    Dim intCurrTbl As Integer
    Dim strCurrTable As String
    On Error GoTo Err_LinkTables

    rstSharedTables.Open "tblLinkedTables", CurrentProject.Connection, _
        adOpenStatic, Options:=adCmdTableDirect

    '-- Get the total number of linked tables, then display the progress meter.
    rstSharedTables.MoveLast
    intTotalTbls = rstSharedTables.RecordCount
    rstSharedTables.MoveFirst

    SysCmd acSysCmdInitMeter, "Linking Tables...", intTotalTbls

    catCurr.ActiveConnection = CurrentProject.Connection

```

LISTING 18.10 Continued

```
intCurrTbl = 1
Do Until rstSharedTables.EOF      '-- Update the progress meter
    SysCmd acSysCmdUpdateMeter, intCurrTbl
    '-- Attempt to open the current link
    On Error Resume Next

    strCurrTable = rstSharedTables!TableName
    catCurr.Tables.Delete strCurrTable
    catCurr.Tables.Refresh

    On Error GoTo Err_LinkTables

    Dim tblCurr As New ADOX.Table
    Set tblCurr.ParentCatalog = catCurr

    tblCurr.Name = rstSharedTables!TableName
    tblCurr.Properties("Jet OLEDB:Link Datasource") = strDataMDB
    tblCurr.Properties("Jet OLEDB:Create Link") = True
    tblCurr.Properties("Jet OLEDB:Remote Table Name") = _
        rstSharedTables!TableName

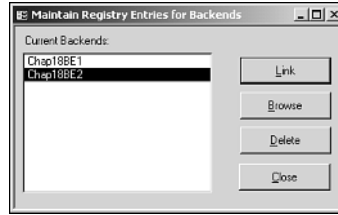
    catCurr.Tables.Append tblCurr
    Set tblCurr = Nothing
    rstSharedTables.MoveNext
    intCurrTbl = intCurrTbl + 1
Loop

Exit_LinkTables:
    SysCmd acSysCmdRemoveMeter
    Exit Sub

Err_LinkTables:
    Resume Exit_LinkTables

End Sub
```

The idea behind this routine is to take the string of the full path and filename of the back end, and then delete and re-create the links to the shared tables. This routine also updates the Access status bar by using the SysCmd statement. Figure 18.12 shows the LinkTable routine in action.

**FIGURE 18.12**

The `LinkTable` routine links the current back-end database while displaying a status bar.

Deleting a Back End from the Registry

Finally, let's see how to delete a back-end entry from the Registry. Note that the application deletes the Registry *entry*, not the file itself. Listing 18.11 shows the code for `cmdDelete_Click`.

LISTING 18.11 Chap18.mdb: Deleting a Back End from the Registry

```
Private Sub cmdDeleteBEReg_Click()
    If IsNull(Me!lstBackends) Then
        MsgBox "Please select a backend to delete the registry entry.", _
            vbInformation, ap_AppTitle()
        Me!lstBackends.SetFocus
    Else
        DeleteBERegEntry Me!lstBackends
        Me!lstBackends.RowSource = _
            GetRegistrySubKeys(HKEY_CURRENT_USER, MyAppBEsKey)
        Me!lstBackends.Requery
    End If
End Sub
```

This routine is similar to the `cmdLink_Click` routine because it ensures that you've chosen a value from the list box before trying to delete the routine. The routine calls `DeleteBERegEntry`, located in the `modBERegistryRoutines` module:

```
Sub DeleteBERegEntry(strBEToDelete As String)
    DeleteAllRegEntriesForKey HKEY_CURRENT_USER, MyAppBEsKey & "\" & _
        strBEToDelete
End Sub
```

Taking the back-end name to delete, this routine calls the `DeleteAllRegEntriesForKey` routine, which is located in the `modRegistryAPIRoutines` module and is an API wrapper routine.

The last control was the `cmdClose` button, which simply uses the `DoCmd` statement to close the form.

Summary

Quite a lot was covered in manipulating the Registry using API calls and VBA commands. The main thing to remember is to take advantage of the routines in the sample app so that you don't have to reinvent the wheel. Although it looks intimidating, after you get a handle on it, it's quite logical in the way that most calls require the predefined key, the path to the key, and maybe the value you're wanting to work with. Keep working on it and you will become quite proficient in no time.

For more information on API calls and linking databases, see these chapters:

- Chapter 15, “Extending the Power of Access with API Calls,” shows how to use a number of different API calls and where to find more.
- Chapter 25, “Startup Checking System Routines Using ADO,” shows how to link tables using ADO at startup.

Using Access with the Internet

CHAPTER

19

IN THIS CHAPTER

- What's New in Access 2002 for the Internet? 574
- Using the Access Hyperlink Features 574
- Easily Importing and Exporting Access Objects to HTML Documents 587
- Publishing to Other Web File Formats 592

Because the Internet is still one of the hottest topics in the computer industry today, it makes sense that Access, as well as all the other Office products, has features that take advantage of the Internet.

This chapter is actually bigger than the Internet, however—if you can believe that. Some features that have sort of “fallen off” the Internet you can now use inside your applications on either an intranet (a companywide Internet), among applications on your system, or even within your Access application itself.

What’s New in Access 2002 for the Internet?

Several new Internet features have come about in Access 2002 and are also discussed to a greater degree in other parts of the book:

- **XML.** The hot topic for this version, XML enables various systems to share data in a common way, with all the systems knowing the standard for accessing the data. Although XML gives you a great method for passing data back and forth over the Internet, a lot of use can also come about over intranets and between systems on a LAN. I’ve added a whole chapter to cover this new feature: Chapter 6, “Using XML with Access 2002.”
- **Enhanced Data Access Pages.** Data Access Pages—new as of Access 2000—allow you to create a new type of form that also has report features and can be directly brought over to the Internet. The source for Data Access Pages is HTML. This feature has been greatly enhanced in Access 2002 and is no longer considered a “version 1”-type feature. Data Access Pages are covered in Chapter 12, “Working with Data Access Pages.”
- **Forms and reports.** You can now save forms and reports as Data Access Pages, thus allowing you to create “live Web reports” and interfaces that can more closely match between LAN solutions and Web solutions. This feature is also discussed in Chapter 12, “Working with Data Access Pages.”
- **Enhanced Office Web Components.** Also introduced in Access 2000, the Office Web Components—Spreadsheet, Pivot Chart, and Pivot Table—have all been enhanced. You can now open the standard Access objects (and SQL Server) such as tables, queries, forms, and stored procedures in these views. They have robust object models including properties, methods, and events. These components are discussed in greater detail with examples in Chapter 10, “Expanding the Power of Your Forms with Controls.”

Now look at other Internet features found in Access, starting with the hyperlink.

Using the Access Hyperlink Features

Hyperlinks have been around for quite a while in one form or another. In help systems, hyper-text is used to create “hot spots” that move users to information applying to the subject they clicked.

The Internet uses hyperlinks to move from one HTML page to another, or to move from one URL to another. HTML (Hypertext Markup Language) is the standard language used for creating Web pages. Although it sounds like something college students do after a Friday night, *URL (uniform resource locator)* is the standard method for addressing various locations on the Web.

You can use hyperlinks in various forms in Access:

- As an unbound control on a form linking to objects in the current database, current system, networked computer, intranet address, or Internet address
- As a field in a table, with a special data type, performing the same links as just mentioned

NOTE

Regardless of what you use to call a hyperlink and what kind of call it was, the Web toolbar appears.

You can see various examples for hyperlinks on the frmHyperLinkExamples form (see Figure 19.1), found in Chap19.mdb. This database can be found on the book's Web page at www.sampublishing.com.

LastName	FirstName	FavoriteHomePage
Barker	Steven	Microsoft's Home Page
Anders	Maria	http://www.sampublishing.com/
Barker	Diana	ap_SystemUtilities
Hughes	Joram	Applications Plus
Hardy	Thomas	

FIGURE 19.1

This form and subform contain bound and unbound hyperlink controls.

Working with Unbound Hyperlink Controls

You can hyperlink-enable three different Access control types: label, image, and command button. Each control has two hyperlink properties—Hyperlink Address and Hyperlink SubAddress—which you can see in the property sheet (see Figure 19.2) or the Edit Hyperlink dialog (shown later in Figure 19.3).

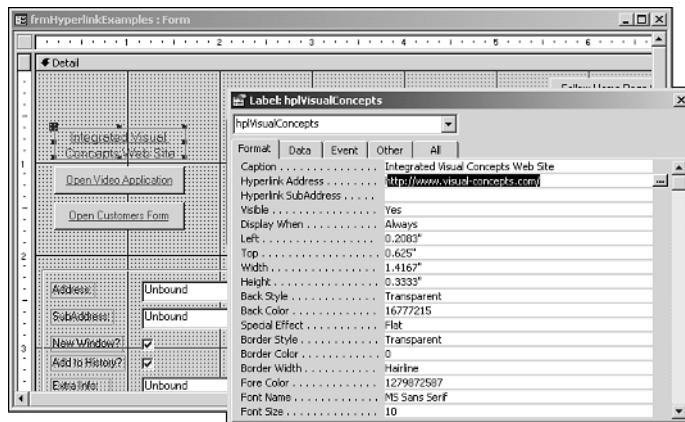


FIGURE 19.2

By filling in the Hyperlink Address property, you can link to an URL or a path and filename of another application, as well as other protocols.

Using the Hyperlink Address and Hyperlink SubAddress Properties

The Hyperlink Address is where you place the address of what you want linked. This address can be any of the following protocols:

<code>\\</code> (UNC path)	<code>mailto:</code>	<code>pnm://</code> (Real Audio Media)
<code>cid:</code>	<code>mid:</code>	<code>prospero:</code>
<code>file:</code>	<code>mms://</code> (Microsoft Media Server media)	<code>rlogin:</code>
<code>ftp:</code>	<code>msn:</code>	<code>telnet:</code>
<code>gopher:</code>	<code>news:</code>	<code>tn3270:</code>
<code>http:</code>	<code>nntp:</code>	<code>wais:</code>

NOTE

The *UNC (Universal Naming Convention)* path is the absolute path used with Windows NT/2000 network servers. Suppose that the VideoApp(ADO).mdb database is in the \Examples folder and assigned to the logical H: drive, which is assigned the share \\Abednego\Root. The UNC path would be

\\Abednego\Root\Examples\VideoApp(ADO).mdb

Use the Hyperlink SubAddress property to specify the object in the file you want to use. An example here could be a form within VideoApp(ADO).mdb—in this case, ap_SystemUtilities. For this example, the Hyperlink SubAddress would be

Form ap_SystemUtilities

Editing the Hyperlink Properties

You can edit the Hyperlink Address and Hyperlink SubAddress properties in several ways. The easiest way is to right-click the hyperlink control you're interested in editing; then choose Hyperlink, Edit Hyperlink. The Edit Hyperlink dialog opens for the label control hp1VisualConcepts (see Figure 19.3).

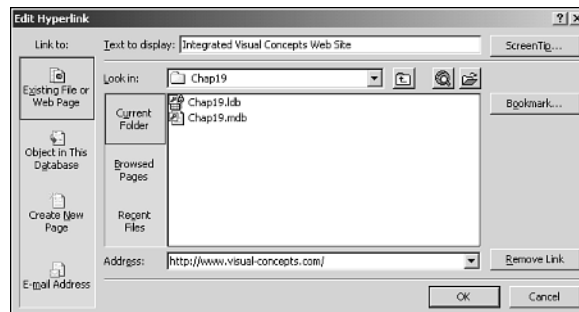


FIGURE 19.3

I typed this URL for the Integrated Visual Concepts home page on the Internet.

The first item in the dialog is the Text to Display text box. Next is the button that leads to ScreenTips. These controls are the same regardless of which type of hyperlink you are creating.

To work through the rest of this dialog, look at the Link To list along the left side of the dialog:

- *Existing File or Web Page* specifies a Web page (including World Wide Web sites) or a physical file. This is the Hyperlink Address. If the address is a file, click the Open File button to find the file on the machine's hard drive or on the network (File) or a Web Page.

When specifying a subaddress such as a form within another database, separate the address from the subaddress with the pound sign. For instance, to open the `ap_SystemUtilities` form within `VideoApp.mdb`, the `address#subaddress` would be `..\VideoApp(ADO).mdb#Form ap_SystemUtilities`

- *Object in This Database* allows you to specify an object in the current Access database. When this Link To option is selected, you will see the Hyperlink SubAddress—actually, a tree that displays all the objects (see Figure 19.4).

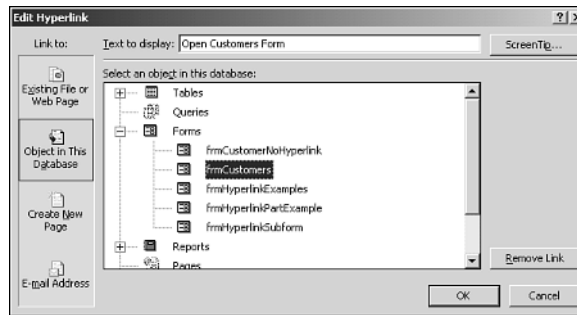


FIGURE 19.4

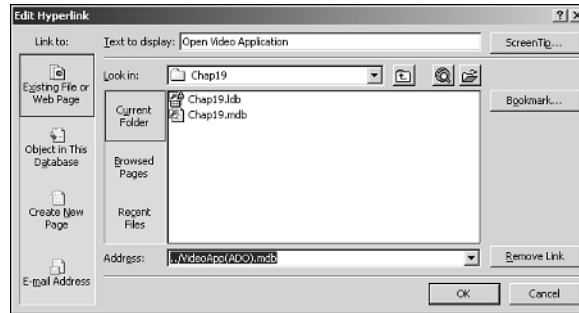
Here, clicking *Object in This Database* creates a hyperlink in which the address is the current application.

- *Create New Page* allows you to create a new Web page and point to it, or just specify a Web page and actually create it later.
- *E-mail Address* is great for creating *PIMS (Personal Information Systems)*. This allows you to easily specify an e-mail address for a field.

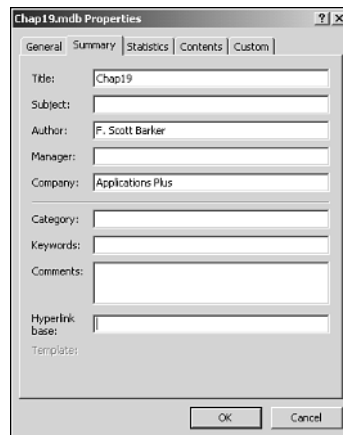
In the Edit Hyperlink dialog, you need to use only the `..` to specify that you want an address to be relative to the application (see Figure 19.5). Use the `..` whenever you have hyperlinks to subfolders beneath your application. That way, if you move your application, Access can still find the path relative to the new location.

Maintaining a Hyperlink Base for a Database

You can also have your address relative to another path rather than your application. To change the Hyperlink Base property, choose Database Properties from the File menu. Click the Summary tab to see the Hyperlink Base property at the bottom (see Figure 19.6).

**FIGURE 19.5**

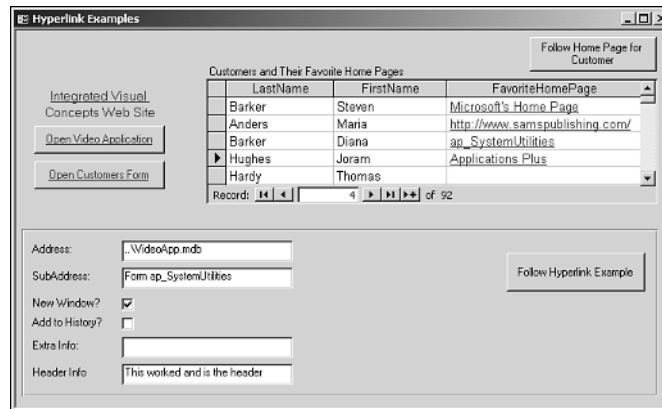
This hyperlink uses the path of the address relative to the location of the current application.

**FIGURE 19.6**

Set Hyperlink Base when you want to make your hyperlinks relative to another location.

Looking at the Hyperlink Data Type

In addition to using unbound controls, there's a data type of Hyperlink. The Hyperlink data type, in the list of data types in Table Design view, lets you store hyperlinks within a table and lets users go to the address stored in the Hyperlink data type field. To see this, look at the frmHyperlinkExamples form in Form view (see Figure 19.7). Notice that in the Datasheet view of the subform (on the right side of the form), the first record displays the text Microsoft's Home Page, and the second displays an URL for Sams Publishing's home page.

**FIGURE 19.7**

You can create captions for hyperlinks even with the Hyperlink data type's Table field, as seen for Joram Hughes's Favorite Home Page (Applications Plus). The hyperlink is <http://www.AppsPlus.com>.

To display the caption versus the actual hyperlink address when you're adding the hyperlink to the table, place the cursor in the Hyperlink Type field and then press F2. You can also add it by using the Text To Display text box in the Edit Hyperlink dialog.

CAUTION

When adding a hyperlink to a table, don't click an existing hyperlink because you'll end up calling it. This causes problems if you aren't hooked up to the Web.

The other way to get into a hyperlink's Edit mode is to right-click the field, choose Hyperlink, and then choose Select Hyperlink. You aren't taken to another dialog, but you see the actual entry into the field, separated by a pound sign (see Figure 19.8). Another way to see the entire hyperlink text is to press F2 when on the table entry.

**FIGURE 19.8**

You can see the entire hyperlink address and caption for Application Plus's Web site.

The Hyperlink type field is divided into four parts, separated by a pound sign (#):

- Display text or caption
- Address, using either a UNC or an URL, as described earlier in “Editing the Hyperlink Properties”
- SubAddress, an object in a file—again, as described in “Editing the Hyperlink Properties”
- ScreenTip, the ToolTip that appears when users hover the mouse pointer over the hyperlink

This was done for Microsoft’s home page by typing the following into the FavoriteHomePage field:

```
Microsoft's Home Page#http://www.msn.com#
```

If you’re using the ap_SystemUtilities form in VideoApp.mdb, you type

```
Utilities Form#..\VideoApp.mdb#Form ap_SystemUtilities#
```

Using the IsHyperlink Property to Add Hyperlinks to Your Interface

You might find yourself using a back-end database, such as SQL Server, that doesn’t support Access’s Hyperlink data type. Wouldn’t it be great if you could still use bound hyperlinks on your forms? In Access you can. Access text boxes and combo boxes each have the IsHyperlink property on the Format tab. If you set this property to Yes, Access treats the data in the field like a hyperlink—it uses the hyperlink display format: You can edit the field with the Edit Hyperlink dialog, and you can follow the link with a single click.

To see the IsHyperlink property in action, select the tblCustomersNoHyperlink table in the Database window. This table stores the FavoriteHomePage in a memo field rather than a hyperlink. With this table selected, choose AutoForm from the Insert menu. Access creates and opens the form in Figure 19.9.

The FavoriteHomePage field now displays the raw, #-delimited data stored in the field.

Another feature is that you can modify many form properties in Page view. You might already see the form property sheet, but if not, choose Properties from the View menu. Put the cursor in the FavoriteHomePage field. On the Format page, change IsHyperlink to Yes. The raw text format is changed to a hyperlink, with only the display text visible (see Figure 19.10). You can now click and follow this link.

The screenshot shows a Microsoft Access form titled 'tblCustomerNoHyperlink'. The form contains several text boxes for customer information: CustomerID (1), LastName (Barker), FirstName (Steven), Address (455 Anystreet), City (Woolville), State (WA), ZipCode (98933), PhoneNo ((206) 555-3994), and FavoriteHomePage (Microsoft's Home Page#http://www.home.msn.com/#). The FavoriteHomePage field is displayed as plain text. At the bottom, a status bar indicates 'Record: 1 of 92'.

FIGURE 19.9

The FavoriteHomePage field appears as plain text by default because it's stored in a Memo data type.

This screenshot is identical to the previous one, but the 'FavoriteHomePage' field now displays the text 'Microsoft's Home Page' with a mouse cursor hovering over it. Below the text, the URL 'http://www.home.msn.com/' is visible, indicating that the field is now functioning as a hyperlink. The status bar at the bottom still shows 'Record: 1 of 92'.

FIGURE 19.10

The FavoriteHomePage field now looks and behaves like a hyperlink.

Programmatically Using Hyperlinks with the Follow, FollowHyperlink, and HyperlinkPart Methods

When you want to go to a hyperlink's address in code, you can use one of two methods: Follow or FollowHyperlink.

Using the Follow Method

You use the Follow method when a hyperlink control or field already exists. You will use the method off the actual (hyperlink) object. An example of using Follow might be when a customer is using the system and requests the system to go to his recorded home page.

Listing 19.1 shows an example of using the Follow method. This code is attached to the OnClick event of a new command button, cmdFollowHomePage, which was added to the frmHyperlinkExamples form.

LISTING 19.1 Chap19.mdb: Firing Off a Hyperlink Programmatically with the Follow Method

```
Private Sub cmdFollowHomePage_Click()  
    On Error GoTo cmdFollowHomePage_Error  
  
    If Len(Me!HyperLinkExamplesSub!FavoriteHomePage.Hyperlink.Address) = 0 Then  
        Beep  
        MsgBox "This Customer doesn't have a home page!", vbCritical, _  
            "Error Occurred"  
    Else  
        Me!HyperLinkExamplesSub!FavoriteHomePage.Hyperlink.Follow  
    End If  
  
Exit Sub  
  
cmdFollowHomePage_Error:  
    MsgBox Err.Description  
Exit Sub  
  
End Sub
```

Notice that the `Address` property and `Follow` method are off the `Hyperlink` property for the specific control being used—in this case, `FavoriteHomePage`. The syntax for the `Follow` method is as follows:

```
object.Hyperlink.Follow(newwindow,addhistory,extrainfo,method,headerinfo)
```

The `Follow` method uses these optional arguments:

- The Boolean *newwindow* argument specifies whether to open the document in a new window. The default is `false`.
- The Boolean *addhistory* argument specifies whether to add the hyperlink to the history folder. The default is `true`.
- The *extrainfo* information can be a parameter for the address, provided that it takes one.
- *method* is one of two constants: `msoMethodGet` (*extrainfo* is appended to the address and is a string) and `msgMethodPost` (*extrainfo* is posted as a string or an array of type byte).
- The *headerinfo* string argument lets you specify what you want displayed in the header. By default, the string is zero-length.

In some cases, you need to allow users to specify a hyperlink on-the-fly. In that situation, use the `FollowHyperlink` method off the `Application` object.

Using the FollowHyperlink Method

When using the FollowHyperlink method, you supply the Address and SubAddress information as well as some of the information used for the Follow method.

For the example here, a number of controls are on the frmHyperlinkExamples form so users can specify the parameters used with the FollowHyperlink method. You can see these controls on the lower half of the frmHyperlinkExamples form in Figure 19.11. Listing 19.2 shows the code for following a hyperlink.

FIGURE 19.11

The controls in the lower part of the form will be used with the FollowHyperlink method off the Application object.

LISTING 19.2 Chap19.mdb: Following a Hyperlink

```
Private Sub cmdFollowHyperlink_Click()
    On Error GoTo cmdFollowHyperlink_Error

    If IsNull(txtExtraInfo) Then
        If IsNull(txtSubAddress) Then
            Application.FollowHyperlink Me!txtAddress, , _
                Me!chkNewWindow, Me!chkAddHistory, , , Me!txtHeaderInfo
        Else
            Application.FollowHyperlink Me!txtAddress, Me!txtSubAddress, _
                Me!chkNewWindow, Me!chkAddHistory, , , Me!txtHeaderInfo
        End If
    Else
        If IsNull(txtSubAddress) Then
            Application.FollowHyperlink Me!txtAddress, , Me!chkNewWindow, _
                Me!chkAddHistory, Me!txtExtraInfo, , Me!txtHeaderInfo
        End If
    End If
End Sub
```

LISTING 19.2 Continued

```

Else
    Application.FollowHyperlink Me!txtAddress, Me!txtSubAddress, _
        Me!chkNewWindow, Me!chkAddHistory, Me!txtExtraInfo, , _
        Me!txtHeaderInfo
End If
End If

Exit Sub

cmdFollowHyperlink_Error:
MsgBox Err.Description
Exit Sub

End Sub

```

In cases such as `txtSubAddress` and `txtExtraInfo`, you need to check for null values because the `FollowHyperlink` method isn't very forgiving when dealing with null values.

Using the HyperlinkPart Method

Sometimes you might find it necessary to break down raw, #-delimited hyperlink data into its component address, subaddress, display text, and ScreenTip parts. You can easily do this with the Access Application object's `HyperlinkPart` method. The `frmHyperlinkPartExample` form in `Ch19.mdb` (see Figure 19.12) uses the `HyperlinkPart` method to perform such a breakdown.

The screenshot shows a form titled "Customers" with a data entry grid. The grid contains the following fields and values:

CustomerID	
LastName	Barker
FirstName	Diana
Address	Avda. de la Constitución 2222
City	Woodville
State	WA
ZipCode	98072
PhoneNo	(206) 555-3433
FavoriteHomePage	ap_SystemUtilities

Below the data grid, the `HyperlinkPart` method is demonstrated with the following fields and values:

Full Address:	..\VideoApp(ADO).mdb#Form ap_SystemL
Address:	..\VideoApp(ADO).mdb
Sub Address:	Form ap_SystemUtilities
Display Text:	ap_SystemUtilities
Displayed Value:	ap_SystemUtilities
Screen Tip:	

At the bottom of the form, a status bar indicates "Record: 14 of 92".

FIGURE 19.12

The parts of the `FavoriteHomePage` field are separated in the `frmHyperlinkPartExample` form.

In the example, the hyperlink part fields are recalculated in the form's Current event. Listing 19.3 shows the code.

LISTING 19.3 Chap19.mdb: Using the HyperlinkPart Method

```
Private Sub Form_Current()  
    Dim strHyperlink As String  
    strHyperlink = Nz(Me.FavoriteHomePage)  
    'Use the HyperlinkPart function to decompose the hyperlink  
    txtAddress = HyperlinkPart(strHyperlink, acAddress)  
    txtDisplayedValue = HyperlinkPart(strHyperlink, acDisplayedValue)  
    txtDisplayText = HyperlinkPart(strHyperlink, acDisplayText)  
    txtFullAddress = HyperlinkPart(strHyperlink, acFullAddress)  
    txtScreenTip = HyperlinkPart(strHyperlink, acScreenTip)  
    txtSubAddress = HyperlinkPart(strHyperlink, acSubAddress)  
End Sub
```

Most intrinsic constants accepted as the optional second argument to the HyperlinkPart method are self-explanatory. The default value, acDisplayedValue, gets you the string that Access will display in the user interface. The acFullAddress value just returns the address and subaddress separated by a # sign.

Working with Hyperlink Options

You've noticed that hyperlinks on the Internet change color after they're selected. This is true in Access as well. To look at the options available, choose Options from the Tools menu, and then click the Web Options button at the bottom of the General page. Figure 19.13 shows the defaults for the Web Options dialog.

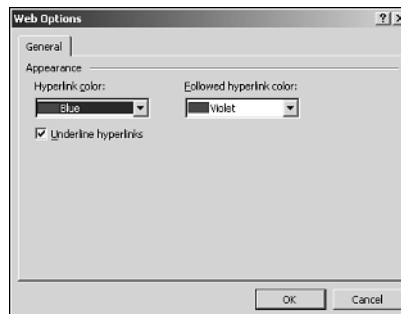


FIGURE 19.13

You can specify how you want hyperlinks to be displayed.

These options are available for hyperlinks:

- *Hyperlink Color* sets the color of hyperlinks for the application.
- *Followed Hyperlink Color* sets the color of hyperlinks for the application that have been followed or selected.
- *Underline Hyperlinks* determines whether to display hyperlinks with underlines.

Easily Importing and Exporting Access Objects to HTML Documents

As mentioned at the start of this chapter, HTML is a language used for creating Web pages used with the Internet and intranets. In Access, you can create HTML files from the various Access objects.

You can export objects in four ways: with an HTML, ASP, IDC, or XML file type.

Exporting to HTML

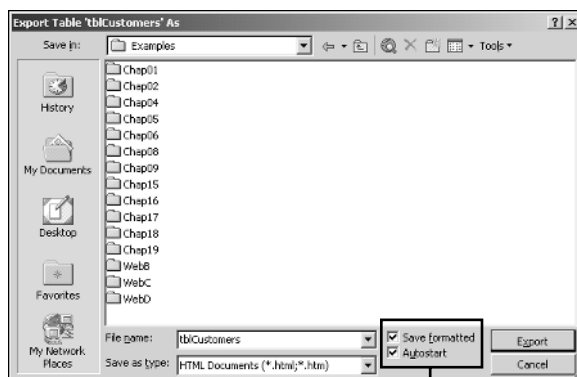
The most expedient way to export an object to HTML format is to highlight the object to export—in this case, the `tblCustomers` table—and then choose Export from the File menu. In the standard Export As dialog, select HTML Documents from the Save as Type drop-down list. Using the Export command in this way creates a separate, static Web document.

A couple of extra options become available when you choose HTML Documents (see Figure 19.14):

- *Save Formatted* allows you to save the headings and table order into the HTML file.
- *Autostart* starts your default Web browser for you automatically, after exporting the object.

Select these two options and then click Export. Access creates a datasheet type view when the HTML file (with an `.htm` extension) is viewed in page format. This `.htm` file is placed in your default folder and displayed in the Web document's address. The next three figures show

- The original table, `tblCustomers`, found in `Chap19.mdb` (see Figure 19.15).
- The `tblCustomers` page view in IE (see Figure 19.16).
- The HTML source file. To get to this code while in IE, choose Source from the View menu. IE uses Notepad (or WordPad for larger files) to display the file (see Figure 19.17).



HTML options

FIGURE 19.14

Get ready to save a table as an HTML document.

tblCustomers: Table							
CustomerID	LastName	FirstName	Address	City	State	ZipCode	Phone
1	Barker	Steven	465 Anystreet	Woodville	WA	98072	(206) 555-0000
2	Anders	Maria	Obere Str. 57	Woodville	WA	98072	(206) 555-0001
3	Barker	Diana	Ayda, de la Con	Woodville	WA	98072	(206) 555-0002
4	Hughes	Joram	Mataderos 231	Woodville	WA	98072	(206) 555-0003
5	Hardy	Thomas	120 Hanover Sq	Woodville	WA	98072	(206) 555-0004
6	Berglund	Christina	Berguvsvägen 8	Woodville	WA	98072	(206) 555-0005
7	Moos	Hanna	Forsterstr. 57	Woodville	WA	98072	(206) 555-0006
8	Citeaux	Frédérique	24, place Kléber	Woodville	WA	98072	(206) 555-0007
9	Sommer	Martin	C/ Araquil, 67	Woodville	WA	98072	(206) 555-0008
10	Lebihan	Laurence	12, rue des Bou	Woodville	WA	98072	(206) 555-0009
11	Lincoln	Elizabeth	23 Tsawassen E	Woodville	WA	98072	(206) 555-0010
12	Ashworth	Victoria	Fauntleroy Circl	Woodville	WA	98072	(206) 555-0011
13	Simpson	Patricio	Cerrito 333	Woodville	WA	98072	(206) 555-0012
14	Chang	Francisco	Sierras de Gran	Woodville	WA	98072	(5) 555-0013
15	Wang	Yang	Hauptstr. 29	Woodville	WA	98072	0452-0014
16	Afonso	Pedro	Av. dos Luslada	Woodville	WA	98072	(11) 55-0015
17	Brown	Elizabeth	Berkeley Garde	Woodville	WA	98072	(71) 55-0016
18	Ottlieb	Sven	Walsenweg 21	Woodville	WA	98072	0241-0017
19	Labruno	Janine	67, rue des Cini	Woodville	WA	98072	40.67.10018
20	Devon	Ann	35 King George	Woodville	WA	98072	(71) 55-0019
21	Mendel	Roland	Kirchgasse 6	Woodville	WA	98072	7675-30020
22	Cruz	Aria	Rua Orós, 92	Woodville	WA	98072	(11) 55-0021

FIGURE 19.15

Here you have the original Access table to export.

FIGURE 19.16

When you export to HTML, Access fires off whatever Web browser you happen to be using—in this case, Microsoft's IE.

FIGURE 19.17

This HTML source file for a Web document was created from Access data.

You can retrieve information from an HTML file as well. Also remember that you can export other objects, such as reports, forms, and queries.

Importing and Linking to HTML Files

Access includes a new file type for importing and linking files. To link a file of the HTML type, follow these steps:

1. From the main database window, choose Get External Data from the File menu, and then choose Link Tables. The Link dialog appears.
2. From the Files of Type drop-down list, select HTML Documents (see Figure 19.18). Then choose the name of the .htm file to link to (in this case, tblCustomers.htm) and click Link.

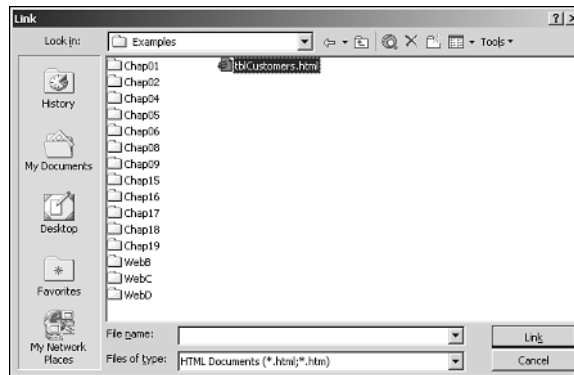


FIGURE 19.18

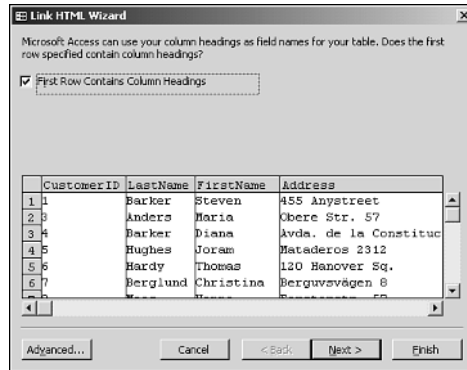
Access seems to add file types to import and link with each version.

The Link HTML wizard opens. As with wizards that help you import and export file formats, the Link HTML wizard walks you through linking to an HTML file. First, it displays the HTML file to be linked and asks whether the First Row Contains Column Headings (see Figure 19.19).

NOTE

The Advanced button in the dialog's bottom-left corner leads you to a link specification for the file. This is actually the same specification dialog used for importing and exporting files.

3. Check that the first row does in fact contain column headings, as you can see in Figure 19.19. Then click Next.

**FIGURE 19.19**

As with importing text files and spreadsheets, the Link HTML wizard steps you through linking to an .htm file.

4. You're given the opportunity to change the Field Name and Data Type of each field. You can also specify whether to skip the field altogether, with the Do Not Import Field (Skip) check box (see Figure 19.20).

**FIGURE 19.20**

Through this dialog, you can set how each column should appear in Access, if at all.

5. Click Next after you modify the fields. The last dialog of the HTML Link Wizard lets you name the link to the file anything you want. It uses the name part of the file as the default.
6. Type 1 after the tblCustomers table name in the Link Table Name text box. Then click Finish. You get a message saying that the link was successful.

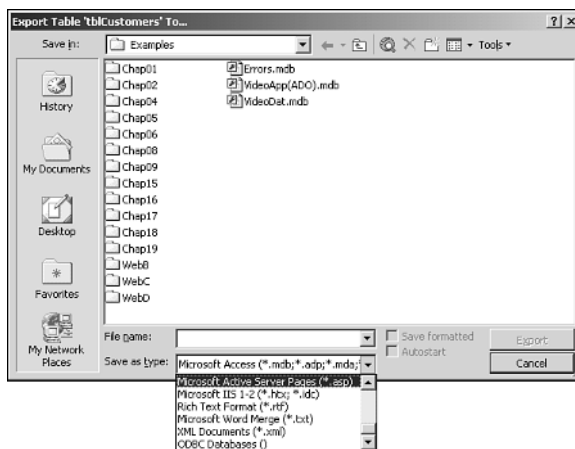
You then see the link listed in the current table listing with an Internet Explorer icon beside it. When you double-click the table, you see the recordset the table consists of.

NOTE

Linked HTML tables are static only. To update an HTML file, you need to import the table by following the steps just given but by choosing Import instead of Link Tables in step 1. Then update the data in the new table. Finally, use the Export command (discussed earlier in the section “Exporting to HTML”) to re-create the .htm file.

Publishing to Other Web File Formats

By using Export from the File menu, you can also publish Access objects to other Web file formats—specifically, ASP, IDC, and XML—by selecting these types from the Save as Type field (see Figure 19.21).

**FIGURE 19.21**

You can select Web file format options from the Save as Type list.

NOTE

Because some features mentioned from here on in this chapter require specific Web servers and it would be unreasonable to assume that you have them all, the features aren't covered in great detail.

You are then presented with the Microsoft Active Server Pages Output Options dialog, and can fill it in (see Figure 19.22).

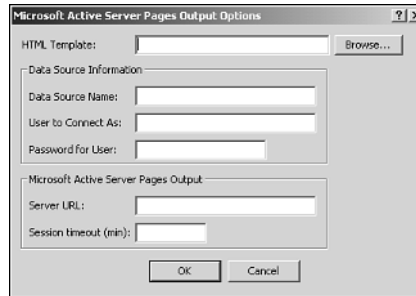


FIGURE 19.22

You can create an ASP file format from an Access object.

If you had chosen the HTX/IDC format, you would be presented with the dialog in Figure 19.23.

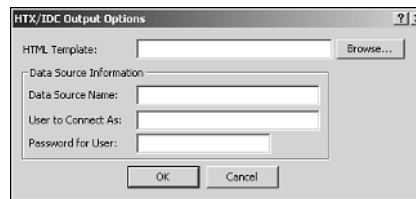


FIGURE 19.23

Fill in the information for HTX/IDC.

This is a quick-and-dirty example of moving a publication out onto the Web. To get into it more deeply, you really need access to a Web server. Access has many powerful features for moving your data onto the Internet or intranet.

NOTE

The final Internet related file option, XML, was covered in detail in Chapter 6, "Using XML with Access 2002."

Summary

Access gives you a number of ways to use hyperlinks to your advantage and to use data on Web pages. You can incorporate some of the innovations introduced for the Internet in your Access applications. For more information on Internet features found in Access, see these chapters:

- Chapter 6, “Using XML with Access 2002,” discusses how to take advantage of a new standard for passing data between various systems both on and off the Net.
- Chapter 10, “Expanding the Power of Your Forms with Controls,” goes over the Office Web components, which you can use either as controls on forms, reports, and Data Access Pages, or by themselves to give a different view of tables, queries, and forms.
- Chapter 12, “Working with Data Access Pages,” covers exciting features found in the development and use of Data Access Pages to get your information onto the Internet. Also, converting forms and reports to Data Access Pages is covered.

Managing Databases

PART IV

IN THIS PART

- 20 Securing Your Application 597
- 21 Handling Multiuser Situations 651
- 22 Welcome to the World of Database Replication 691
- 23 Moving Workgroup Applications to Client/Server 741
- 24 Developing SQL Server Projects Using ADPs 779

IN THIS CHAPTER

- Understanding the Purpose of Securing Applications 598
- Understanding Access Security 599
- Looking at the Security User Interface 608
- Making Life Easier with Access Security Tools 625
- Avoiding Common Pitfalls Found in Access Security 626
- Managing Security Through Code 630
- Using the Secured Sample Database: Chap20s.mdb 647

One of the most important tasks you need to perform as an Access developer before distributing an application is to secure the database. Access provides one of the most sophisticated, integrated security models in the PC database world. Although the Access security model is somewhat daunting at first, by the end of this chapter you should feel quite comfortable with it. In fact, with the information provided in this chapter, you'll have the tools necessary to deliver secured applications.

Within Access 2002 is the Security Wizard, along with virtually the same user interface for maintaining security. These tools, plus the programmatic access to security through Data Access Objects (DAO), ActiveX Data Objects (ADO), or SQL statements, give you everything you need when it comes to delivering secured Access applications.

NOTE

Trying to cover all three programmatic possibilities would take a whole book. Because this chapter deals with only Access objects, and not SQL Server, I will use DAO with VBA to show how to deal with Access objects.

Understanding the Purpose of Securing Applications

Obviously, there's little reason to secure the recipe database you created for your mother or that wedding database you whipped up for your daughter. You're probably not being paid for those databases. However, when a customer has a business need, you have to be able to secure the application. The two main reasons to secure an application are to protect you as the developer and to protect your clients.

Protecting Sensitive Data: The Client's Perspective

If you're creating a business solution, your client probably doesn't want just anybody thumbing through company files looking at sensitive data. If you don't secure your application correctly, anybody with a copy of Access could quite easily look at all your client's data.

For example, you might be developing a Human Resources application in which one employee field contains salary information. It would be unacceptable to deliver an application that allowed any employee in the company with Access to look at the salaries of all their co-workers. By implementing Access security on your application, you can prevent this sort of breach from happening.

Protecting Code: The Developer's Perspective

When creating database applications, you probably spend a lot of effort creating neat little finishing touches. Maybe you worked for several weeks getting a really slick feature to work just right. Or maybe you got the forms and reports set up perfectly, and you don't want others messing around with them, wrecking all the work you did to get that last pixel just right. For either reason, the only way you can protect your code is through the Access security model.

NOTE

Access doesn't compile its databases into executables as other database products can. Two ways to protect code are by creating an .mde file or by using a VBA password (discussed later in the section "Securing Modules in the VBE").

Understanding Access Security

Access has one of the most advanced security models in the PC database industry. This chapter will help you understand Access security by describing the security model, explaining the user interface, discussing common problems, and providing useful code examples.

To understand the Access security model, you must first understand the difference between share-level security and user-level security. The easiest way to envision the difference between share-level and user-level security is to make some analogies to some popular network operating systems.

Share-level security (where *share* refers to a network share such as \\Abednego\\public) is the simpler of the two security models. It's simply a password-protection scheme. If you know the password to the resource, you're provided access to that resource. Each resource has its own password in the share-level model. In the networking model, this can become quite cumbersome if you have to remember the name of each share and its password. If you need access to three different file servers, two SQL servers, and five printers, you need to remember 10 passwords! Share-level security is what you find in most applications that provide password protection.

User-level security is an advanced security model adopted from popular network operating systems. In the user-level model, an administrator creates predefined users and groups. Each user gets a unique logon ID and password. The administrator then assigns permissions to each user. To help ease the process of assigning permissions to multiple users, groups can be created and permissions then assigned to groups. Novell NetWare and Microsoft Windows NT networks use this method of security.

The benefit of user-level security is clear: The end users need to remember only their login names and passwords. From there, they either have permissions to a resource or they don't. The downside is that you need to have a knowledgeable administrator to create, modify, and remove user accounts and permissions. This quite often can become an entire group of positions in large corporations.

Share-Level Security: The Database Password

In Access 95, Microsoft added the most requested security feature from its end-user community: the database password. All passwords within Access, whether database passwords or user passwords, are case sensitive and up to 14 alphanumeric characters long.

To set the database password, you must have permissions to open the database exclusively. (For more information, see the later section “Setting Database Permissions.”) Assuming that you have permissions, follow these steps with the database open:

1. From the Tools menu, choose Security and then Set Database Password.
2. In the Set Database Password dialog (see Figure 20.1), type the database password in the Password text box.



FIGURE 20.1

The Set Database Password dialog lets end users password-protect their databases.

3. Reenter the password in the Verify text box.
4. Click OK to accept the new password or Cancel to abort database password protection.

The next time this database is opened, the user is prompted to enter a database password.

CAUTION

If you forget the password, there's no way to reset it. You'll be locked out of the database and lose access to all your data. Although some Web sites provide code to do so, I can't refer you to any of them here.

Remember that password protection is good only for preventing somebody from opening the database. When a user gets into a password-protected database, there's no other security unless

you set up user-level security. If you haven't set up user-level security, anybody could change any of the objects and the data freely. Using just a database password isn't the preferred method for securing a full-fledged application.

User-Level Security: The Real Security System of Access

Since day one, Access has had a user-level security model built into the product. Surprisingly, few people have taken the time to master it. If you take the time to properly learn and implement Access user-level security in your applications, you'll surely be rewarded for these efforts.

Recall that the user-level security model is based on permissions assigned to users and groups. Each user has a unique logon name and password.

TIP

Access security is always turned on. If the Admin user account doesn't have a password, the system will automatically log a person on as the user Admin. This works great when you aren't using security because everyone then has rights to all objects in all versions of Access. In the beginning of the development cycle, this makes it easy to send your application for review without having to worry about security right away.

To see what user you're logged on as, follow these steps:

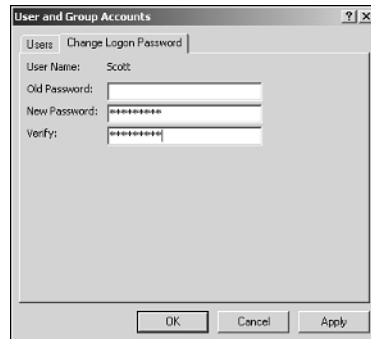
1. From the Tools menu, choose Security and then User and Group Accounts.
2. Click the Change Logon Password tab.

The current username is displayed at the top of the dialog. In Figure 20.2, you can see that user Scott is now logged on to this session of Access.

Users and Groups

By default, Access ships with one user, Admin, and two groups, Admins and Users. The Admin user and Users group accounts are the same across all versions of Access. This is by design so that everybody who ever bought a version of Access can share their databases if they don't want to implement security. Remember, Access security is always active; you just don't see it when you're logging on as Admin with no password.

Versions of Access before Office 95 had two additional default accounts: Guest and Guests. In Access 95, these accounts were removed.

**FIGURE 20.2**

You can always find out who you're logged on as in the User and Group Accounts dialog, or by typing `?_CurrentUser()` in the Debug window.

NOTE

Only members of the Admins group can create users and groups. What's more, only members of the Admins group and the database creator can change permissions on the database object and encrypt the database.

CAUTION

All user accounts are members of the Users groups; there's no way around this in the user interface. However, in code you can make some very serious mistakes because there's no security error checking, per se. Through code, you can create users that belong to no groups. This isn't advised because everyone should belong to at least one group.

You can never remove the three default accounts created by Access. The Jet database engine will protect these accounts. Furthermore, at least one user must always be in the Admins group. This user can be one that you, as the developer, create. It doesn't have to be the default Admin account. In fact, a database isn't secured if the default Admin account is a member of the Admins group.

For details about removing users from the Admins group, see the later section "Removing Users and Groups." For more information about creating users and groups through the user

interface, see the section “Looking at the Security User Interface.” For details about creating users and groups through code, see the section “Managing Security Through Code.”

Permissions

Access security is based on a database object granularity scale. This means that you can assign different security permissions on a per-object basis. You can set security permissions on a form in the database window, but you can’t set security permissions on a form’s combo box object.

Each user and group can be assigned various permissions for each database object. Table 20.1 shows the permissions that you can assign based on the object type.

TABLE 20.1 Object Permissions

<i>Permission</i>	<i>Description</i>
<i>Table and Query Object Types</i>	
Read Design	Allows for viewing the Design view
Modify Design	Allows for modifying the object
Read Data	Allows for viewing the data
Update Data	Allows for changing existing data
Insert Data	Allows for the addition of data
Delete Data	Allows for the deletion of data
Administer	Allows for full access of the object and assigning permissions
<i>Form, Report, and Macro Object Types</i>	
Open/Run	Allows for execution of the object
Read Design	Same as tables and queries
Modify Design	Same as table and queries
Administer	Same as table and queries

NOTE

To see a table’s or query’s data, select Read Data and Read Design. If you select Read Data through the user interface, Access automatically selects Read Design. However, if you’re assigning permissions through code, be sure to assign both.

NOTE

Also, there is no security (and therefore no permissions) on Data Access Pages. Modules are protected in the Visual Basic Editor with one password for all code in the VBA project, so no permissions are assigned using Access security.

Access has additional permissions that you can set at the database level:

<i>Permission</i>	<i>Description</i>
Open/Run	Allows for opening of the database
Open Exclusive	Allows for opening the database exclusively
Administer	Allows for setting the database password, replicating a database, and changing startup properties

For assigning permissions through the user interface, see the later section “Looking at the Security User Interface.” For examples of how to set permissions programmatically, see the section “Managing Security Through Code.”

Do I Have Permissions?

Within Access is the concept of implicit and explicit permissions. Users automatically get all the permissions assigned to them for each object. These permissions are known as *explicit*. Also, users get all permissions assigned to any group they belong to. These permissions are *implicit* because they’re tacitly granted to users by their belonging to those groups.

When an inconsistency exists between explicit (user) permissions and implicit (group) permissions, Access grants the least restrictive permissions. To put it another way, the permissions are additive. The following examples help illustrate this point.

Assuming that the Users group has no permissions to the Customers table, Jennifer might have been given explicit permissions to read the table. That is, her account has both Read Data and Read Design permissions. Jennifer is also a member of the Employees group, which has permissions to Update Data and Insert Data. So Jennifer would be able to read, update, and insert data. Because Jennifer is a member of only the Employees group and no other, however, she can’t delete data from the Customers table.

Ryan might have no permissions to the Customers table explicitly assigned to his account. However, Ryan is a member of the Managers group, which has permissions to Read Data (and Read Design), Update Data, Insert Data, and Delete Data. So Ryan can read, update, insert, and delete data in the Customer table.

By looking at these examples, you can see why assigning users to groups is beneficial. If you know you will have 30 employees, three managers, and 100 objects in your database, you really don't want to assign 3,300 different sets of permissions when you could assign 200.

Ownership

Another characteristic of Access security to be aware of is the concept of *ownership*. The creator of an object is known as that object's owner. The owner of an object has some special privileges that can't be removed unless a member of the Admins group changes the object's ownership. (For information about changing an object's ownership, see the later section "Looking at the Security User Interface.")

Owners can always assign permissions to other users and groups for the objects they created. Also, owners can always assign permissions back to themselves if a member of the Admins group removes their permissions. Again, to help illustrate the point, an example is in order.

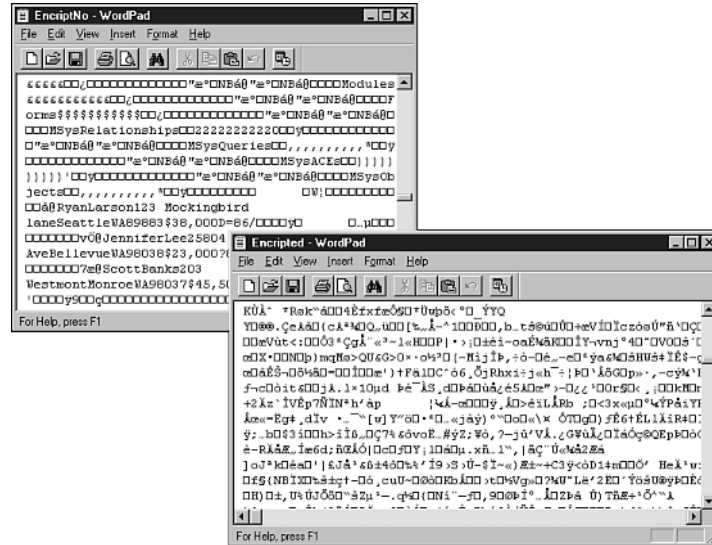
Suppose that Jennifer creates a new report named rptMonthlyVideoSales. She is the owner of the rptMonthlyVideoSales report. As you recall from the example in the preceding section, Jennifer is a member of the Employees group only; she isn't a member of the Admins group. Again, assume that the default Users group has no permissions to any objects. However, she can assign permissions to rptMonthlyVideoSales for other users. And if a member of the Admins group removes all permissions from rptMonthlyVideoSales, she can assign full permissions back to herself. The only way a member of the Admins group can keep Jennifer from reassigning herself permissions is to change the ownership of the rptMonthlyVideoSales report object from Jennifer to someone else.

Database Encryption for Added Protection

Encryption is the process of scrambling, or encoding, the database file so that normal word processors or file-viewing utilities can't view any of the data or definitions within a database. Figure 20.3 shows the differences between an encrypted database and an unencrypted database.

NOTE

Access uses RSA's RC4 encryption technology to scramble the database. RC4 is an industry-standard encryption algorithm.

**FIGURE 20.3**

Which would you rather have: a fairly easy-to-read unencrypted database (top), or an encrypted database (bottom)?

As you can see in Figure 20.3, viewing sensitive data is easy. You can see the salary and address data for three employees in the top window. If user Ryan looks at this database with any text editor, he can see where his salary is in relation to his name. Ryan knows he makes \$38,000, so he could quickly deduce that his boss, Scott, makes \$45,500, and that Jennifer makes \$23,000. Worse yet, nothing can stop him from modifying data.

Any copy of Access can read an encrypted database from another copy of Access. So encryption as the only method of security won't protect you from another user with Access. What encryption does protect you from is the file-viewing utilities.

CAUTION

Encryption doesn't come without a price. Encrypting a database decreases performance about 5% to 15%, depending on the database structure and contents. Also, compression utilities don't affect encrypted databases. Compression utilities look for common patterns in a file to reduce its size. Encryption, by its very nature, creates randomness, thus preventing compression algorithms from doing their jobs.

For details about how to encrypt a database, see the later sections “Looking at the Security User Interface” and “Managing Security Through Code.”

The System.mdw File

Now that the concepts of users, groups, permissions, ownership, and encryption have been covered, it's time to see how Access implements these theories.

Access uses a two-database security model. Permissions are stored in the same database as the objects they pertain to. Users, groups, and passwords are stored by default in a special database named System.mdw. Passwords are stored encrypted. Also, System.mdw stores individual user settings and group membership.

The System.mdw file is known by many interchangeable names among Access developers. Some other names for it are System.mda, SystemDB, *system database*, *workgroup file*, and *workgroup information file*. The Access 2002 documentation refers to it as the latter.

NOTE

If you're familiar with 16-bit Access versions, you might remember this file having the common Access database library extension of .mda. This was changed in Access 95 to prevent confusion with other libraries. Now the extension is .mdw, for Microsoft Database Workgroup.

Regardless of the name it's known by, it's a very important file. In fact, before Access 2000, if you didn't have a valid workgroup information file, Access wouldn't start. As of Access 2000, if the mdw file isn't found, Installer will prompt you to repair the application. And this doesn't happen until you open a Jet database. You can technically start Access without an mdw file.

In previous 16-bit versions, Access found the workgroup information file by looking at the SystemDB line in the [Options] section of the Msaccess.ini (v1.x) or Msacc20.ini (v2.0) file. Now the path to the workgroup information file is stored in the Windows Registry. Developers can find the path to SystemDB under the following Windows 2000 Registry key:

```
\HKEY_USERS\Software\Microsoft\Office\10.0\Access\Jet\4.0\Engines
```

In Windows 9x, it can be found the Registry at

```
\HKEY_LOCAL_MACHINE\Software\Microsoft\Office\10.0\Access\Jet\4.0\Engines
```

You can find out which workgroup information file you're using in several different ways:

- Choose Security and Workgroup Administrator from the Access Tools menu. Before, the Workgroup Administrator program (Wrkgadm.exe) used to be accessible only in the

Access program folder; in Access 2002, Microsoft made it easier to reach by putting it off the Tools menu.

- Use the Registry Editor to navigate the tree given in the preceding path (see Figure 20.4).
- Call a system command from VBA to return the path of the workgroup information file. The system command is in the `ap_CanNotCreateDatabase()` function found at the end of this chapter in the section “Denying Users the Ability to Create Databases.”

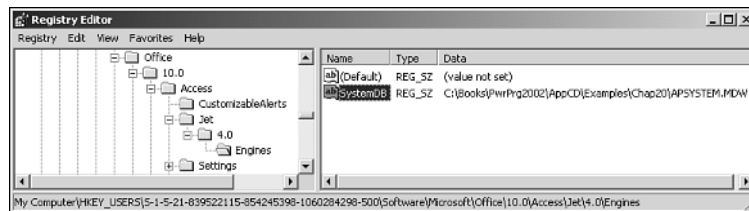


FIGURE 20.4

You can look up the workgroup information file key in the Windows Registry Editor.

Looking at the Security User Interface

You’ve already seen how to add a password to a database through the user interface. The following sections explain how to create new users and groups, assign permissions for the database and its objects, change the owner of an object, encrypt a database, and—finally—create a new workgroup information file.

Working with PIDs, SIDs, WIDs, and Passwords

Before learning how to create a user or a group, you need to understand one additional fundamental security concept. When you create a user or a group (or a workgroup information file, in some sense), what you’re actually doing is creating a *security identifier (SID)*. A SID is an encrypted string used to identify a user or group.

When creating a user or group, you must enter a name and a *personal identifier (PID)*. These two strings are combined and encrypted to form the SID, as in the following formula:

$$(Name + PID)Encrypted = SID$$

Name can be a user or a group.

The SIDs for the Users group and Admin user are the same across all versions of Access. This way, security can run invisibly without getting in the way of the millions of users who don’t want to deal with security.

User Name Criteria

Names can be 1 to 20 alphanumeric characters long, including accented characters, numbers, spaces, and symbols. The exceptions are that names can't begin with spaces, or control characters and the reserved characters (" / \ [] : | < > + = ; , ? *) can't be used. Also, because of the way Access stores users and groups internally, you can't define a user and a group with the same name.

TIP

Come up with a naming convention for your users and groups. Some examples that work are first name, last initial (ScottB) or first initial, last name (SBarker) when creating your users. This will help you to create unique users every time.

Personal Identifier Criteria

PIDs can be 4 to 20 alphanumeric characters long. The PID part of a SID is case sensitive. PIDs follow the same character limitations that user names do.

Workgroup Identifiers

You use *workgroup identifiers* (WIDs) when you create a new workgroup information file. When creating a workgroup information file, you must fill out three fields instead of the traditional two fields. Name, Company, and Workgroup ID are all required to create a workgroup information file.

In creating a workgroup information file, you're actually creating a file that will function as your system database. The default system database is named System.mdw. As this file is created, it also creates the SID for the Admins group. It's your Admins group that keeps your system secure.

So unlike the other two default accounts (Admin and Users), which are the same across all versions of Access, the Admins group is unique to your workgroup information file.

Re-Creating Accounts

Suppose that you create a user with the name MyNewUser and a PID of ThisIsMyPID999. If you have to re-create that user by using MyNewUser and ThisIsmyPID999, you didn't re-create the proper SID. This is one letter off (the *m* in the PID isn't the same), and the encryption algorithm will create different SIDs.

CAUTION

It can't be stressed enough how important your names and PIDs are in Access security. If you ever have to re-create your accounts because of a lost or corrupted workgroup information file, you need to reenter the strings exactly as before.

SIDs are so important because they're used to compare permissions with objects. If a SID matches the object's permissions, Access allows the user to perform an action on an object. Without the proper SID, you might find yourself without any access to any objects, including the database object.

Passwords

User passwords serve as an extra step of security within Access. Passwords aren't necessarily needed for the Access user-level security model to work properly; they're simply used for additional protection.

NOTE

PIDs are *not* passwords! Too many times developers have created new users by providing a name and a PID, only to try to log on as that new user with the name and use the PID as a password. When you log on as a newly created user, you have no password. You can add a password to a user only after you log on.

Passwords verify a user. If a password is created, it and the name are compared to ensure that the login is valid. From there, the SID is used to verify permissions.

Passwords can be up to 14 alphanumeric characters long, with the same limitations on special characters that user and group names have.

Creating a New User

In the Access user-level security model, users must be created. You can log on only as a user, such as Ryan; you can't log on as a group, such as Managers. To create a new user through the Access user interface, follow these steps:

1. From the Tools menu, choose Security and then User and Group Accounts.
2. On the Users page, click New.

3. In the New User/Group dialog, enter a Name and Personal ID.
4. Click OK to save the user.

Figure 20.5 shows the creation of the user Scott with a PID of B1a2R3k4E5r.

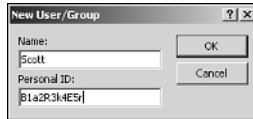


FIGURE 20.5

A new user account called Scott is being created.

NOTE

Because the logon process isn't case sensitive, you can't create a user named Scott and another user named scott, even if you give them different PIDs.

Creating a New Group

Creating new groups is almost identical to creating new user accounts. In fact, you'll see that they share a common interface in Figure 20.6, which shows the creation of a group named Managers with the PID S1e2C3r4E5t.

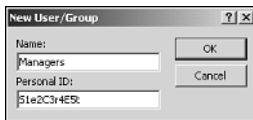


FIGURE 20.6

Create a new group account called Managers.

To create a new group account, follow these steps:

1. From the Tools menu, choose Security and then User and Group Accounts.
2. On the Groups page, click New. (This page is unavailable unless you're logged on as a member of the Admins group.)
3. In the New User/Group dialog, enter a Name and a Personal ID.
4. Click OK to save the group.

Removing Users and Groups

Removing users and groups is just as easy as creating them. To remove a user or group, follow these steps:

1. From the Tools menu, choose Security and then User and Group Accounts.
2. Select the Users or Groups tab, depending on whether you want to remove a user or a group.
3. Select the appropriate user or group from the Name combo box.
4. Click Delete.
5. At the prompt to verify the deletion, click OK for the user or group to be deleted, or click Cancel if you don't want to delete the user or group.

Remember, you can't delete the default accounts—Admin, Admins, and Users—from the system, but you can remove their permissions.

Adding a User to a Group

After you create your users and groups, the next logical step is to add the users to their appropriate groups. Figure 20.7 shows user Ryan being added to the Managers group.



FIGURE 20.7

User Ryan is being added to the Managers group.

Only members of the Admins group can add users to groups. If you're logged on as a non-Admins group member, the Groups tab will be disabled, as will the Add and Remove buttons in the Group Membership section of the Users page.

You can add users to groups through the same dialog used for creating new users. Follow these steps to add a user to a group:

1. From the Tools menu, choose Security and then User and Group Accounts.
2. On the Users page, select the appropriate user from the Name combo box.
3. In the Group Membership section, select the appropriate group from the Available Groups list.
4. Click Add. The group you selected is transferred from the Available Groups list to the Member Of list.

NOTE

Only users can be added to groups; groups can't be added to other groups.

Adding a Password to a User Account

For added protection, you can add a password to a user account. Through the user interface, you must log on as that user to add a password to the account. Follow these steps to add a password to a user account:

1. Start Access, logging on as the user you want to give a password to.
2. From the Tools menu, choose Security and then User and Group Accounts.
3. On the Change Logon Password page, leave the Old Password text box blank (refer to Figure 20.2). Enter the new password in the New Password and Verify text boxes. (Remember that passwords are case sensitive.)

NOTE

The first time you add a password, you don't have a previous password. If you're changing a password, provide the old password.

4. Click OK to accept this new password and exit this dialog. Click Apply to accept this new password and remain in this dialog. Click Cancel if you don't want to save this password.

Removing a Password from a User Account

You can remove a password from a user account in two ways: through the user or through a member of the Admins group.

To remove a password from a user account if you're logged on as that user, simply go to the User and Group Account dialog's Change Logon Password page, enter the old password in the Old Password text box, and leave the New Password and Verify text boxes blank. Click OK or Apply to clear the password.

If a user forgets his password, any member of the Admins group can clear the password on the Users page of the User and Group Accounts dialog by selecting the user's name in the Name combo box and clicking the Clear Password button.

Setting Permissions on Objects

After your users and groups are created, you're ready to assign permissions to the objects you've created. Figure 20.8 shows the Managers group getting Read Design, Read Data, Update Data, Insert Data, and Delete Data permissions for the Categories, Customers, Order Details, and Orders tables.

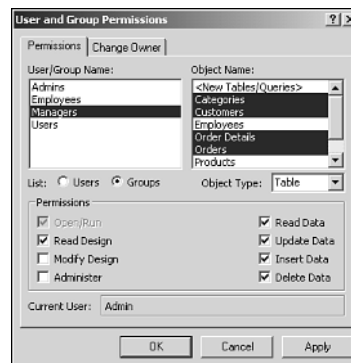


FIGURE 20.8

You can assign permissions to multiple objects.

To assign permissions to objects within the database, follow these steps:

1. Open the database on which you want to change object permissions.
2. From the Tools menu, choose Security and then User and Group Permissions.
3. On the Permissions page, choose whether you want to see a list of Users or Groups from the List options.
4. Choose which database object type you want to modify from the Object Type combo box.
5. Select the user or group from User/Group Name list that you want to assign permissions for. (This is a *single-select list box*; you can choose only one user or group.)

6. Select the object(s) you want to assign permissions to in the Object Name list box. (This is a *multi-select list box*; select as many objects as you want by Shift+clicking or Ctrl+clicking.)

TIP

If you know that all the objects of a certain type will have the same permissions, you can save yourself a lot of time by assigning the appropriate permissions to the default object for the specific object type. To do this, select *New ObjectType* (where *ObjectType* is either Tables/Queries, Forms, Reports, or Macros) from the Object Name list box in step 6. Then all newly created objects of that type will have the default permissions you set.

7. Select the permissions you want to assign to the selected object(s) by selecting and deselecting the check boxes in the Permissions section. (Grayed-out fields don't apply to the object type you've selected.)
8. Click OK to accept the permissions and exit this dialog, click Apply to accept the permissions and remain in this dialog, or click Cancel to exit this dialog without making changes to any permissions.

Changes to permissions take effect immediately, except for objects already opened by users. Their permissions aren't marked until the next time they're opened.

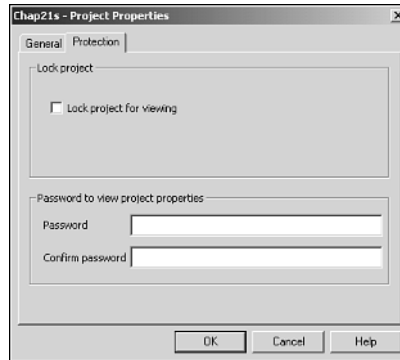
Only the owner of the object, members of the Admins group, and users with the Administer permission for that object can assign permissions to other users and groups.

Securing Modules in the VBE

Access 200x and VBA 6 have you secure your modules by using the Visual Basic Editor. To do this, in Chap20.mdb, follow these steps:

1. Open a module.
2. Choose Chapter 20's Project Properties from the Tools menu.
3. On the Protection page (see Figure 20.9), click the Lock Project for Viewing check box.
4. Supply the password, confirm it, and then click OK.

After closing and reopening Access, you are asked to supply a password when you next try to edit a module.

**FIGURE 20.9**

Modules can now be secured through the VBE.

Setting Database Permissions

Setting database permissions is very similar to setting object permissions. To set permissions for a database, follow these steps:

1. Open the database on which you want to change permissions.
2. From the Tools menu, choose Security and then User and Group Permissions.
3. On the Permissions page, choose whether you want to see a list of Users or Groups from the List options.
4. Choose the Database object from the Object Type combo box.
5. Select from the User/Group Name list the user or group that you want to assign permissions for.
6. Select the permissions you want to assign to the database by selecting and deselecting the check boxes in the Permissions section.
7. Click OK to accept the permissions and exit this dialog. Click Apply to accept the permissions and remain in this dialog. Click Cancel to exit this dialog without changing any permissions.

Setting database permissions separately from the other objects is stressed for two reasons:

- These permissions are often overlooked and not used.
- You should take advantage of these very powerful permissions.

Recall that the three database properties you can set are Open/Run, Open Exclusive, and Administer. Each one yields significant security strengths:

- **Open/Run** is perhaps the most powerful property. By clearing it for the Users group and Admin user, you can effectively require users to utilize your workgroup information file to log on. Your database can be copied around the world, but if they don't have your workgroup information file or the names and PIDs of your accounts so that they can be re-created, nobody can get into the database.
- **Open Exclusive** is useful for database developers/administrators operating in multiuser environments. By clearing this property for all users and groups (except for your developer/administrator accounts), you can prevent users from inadvertently locking other users out of the database. Remember, however, that to compact and repair a database, you need to be able to open the database exclusively.
- **Administer** grants users the ability to change the startup properties, change the database (share-level) password, and create replicas of the database.

Changing the Owner of an Object

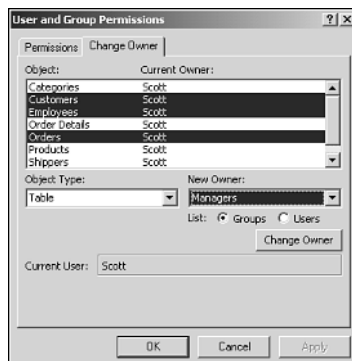
Sometimes changing the owner of an object is necessary. You can do this in two ways through the user interface:

- The easiest way doesn't even require accessing the Tools menu's Security submenu. If you have Modify Design permissions, you can simply cut (or copy) the object from the database window and paste it back in. After pasting, you might need to delete the old object and rename your new pasted object to the original object's name. Similarly, you can import or export an object to become its owner.

CAUTION

This first method won't work with queries that have their Run Permissions property set to Owner's. To change the owner of such a query, either temporarily set the property to User's and change it back after the change of ownership, or re-create the entire query. For more information about this property, see the later section "Running with Owner's Permissions."

- Use the security user interface to transfer ownership. Figure 20.10 shows three table objects (Customers, Employees, and Orders) with ownership created by user Scott and transferred to the Managers group.

**FIGURE 20.10**

The ownership of three tables is changed from user Scott to the Managers group.

NOTE

Groups can't create objects because you can't log on as a group account. Groups, however, can *become* the owner of objects through the User and Group Permissions dialog's Change Owner page.

To change the owner of an object, follow these steps:

1. Open the database in which you want to change object owners.
2. From the Tools menu, choose Security and User and Group Permissions.
3. On the Change Owner page, select which database object type you want to modify from the Object Type combo box.

NOTE

You can't change the owner of a database. Selecting the Database object in step 3 disables the Change Owner button. To change the owner of a database, create a new database while logged on as that user and then import all the database objects.

4. Select the object(s) you want to change the owner of from the Object list box.
5. Choose whether you want to see a list of Groups or Users from the List options.
6. Select the appropriate user or group from the New Owner combo box.

7. Click the Change Owner button to transfer ownership.
8. Click OK to accept the change and exit, or Cancel just to exit this dialog.

Encrypting a Database

Earlier in this chapter, you learned about encryption and how it prevents unwanted eyes from viewing your data through a text editor by encoding how it stores information. To encrypt a database through the user interface, follow these steps:

1. If it's not already started, start Access, but don't open a database. If Access is running, close any databases you might have open.
2. From the Tools menu, choose Security and then Encrypt/Decrypt Database.
3. From the Encrypt/Decrypt Database dialog (a standard file dialog), select the database you want to encrypt and click OK.
4. At the prompt to encrypt the database as another name, choose a new filename and click Save. Access closes the dialog and proceeds with the encryption.

NOTE

To encrypt or decrypt a database, you must be the owner (creator) of the database or a member of the Admins group of the workgroup information file used when the database was created.

To decrypt a database, follow these same steps. When you select a database in step 3, Access determines whether it's encrypted and, if so, prompts you to decrypt it as a different name.

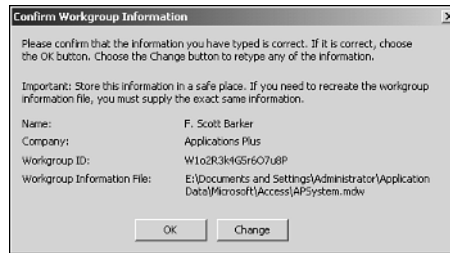
Creating a Workgroup Information File

The workgroup information file is one of the most important aspects of Access security. Creating a new workgroup information file is what makes your database secure—it's the basis for your Admins group SID and the storage of your users, groups, and passwords.

Before Access 2002, you had to run a separate executable (Wrkgadm.exe) outside the Access program to create a new workgroup information file. Access 2002 makes it much easier to access the Workgroup Administrator; simply choose Security and then Workgroup Administrator from the Tools menu. Then, follow these steps:

1. Choose Create in the Workgroup Administrator dialog.
2. Enter a Name, Organization, and Workgroup ID in the Workgroup Owner Information dialog, and then click OK.

3. Choose a Database filename and location in the Workgroup Information File dialog and click OK.
4. Confirm the options you've selected in the Confirm Workgroup Information dialog (see Figure 20.11) by clicking OK. Or click Change to return to step 2.

**FIGURE 20.11**

Keep track of the information about this dialog in a safe place; you might need it later.

5. The Workgroup Administrator creates your new workgroup information file and updates the Windows Registry to point to the newly created file. Click OK in the success dialog that appears.
6. Click Exit in the Workgroup Administrator dialog.

NOTE

To join a different workgroup information file, follow the same steps as explained, but at step 1, choose Join instead of Create. This will prompt you with the same dialog as step 3. Choose Browse to bring up the Select Workgroup Information File dialog. Find the workgroup information file you want to join and click Open. At the successful join message, click OK and Exit.

You can't create a workgroup programmatically. Everything else done up to this point—users, groups, passwords, permissions, ownership, and encryption—can all be performed programmatically. (For examples of how to perform these functions programmatically, see the later section "Managing Security Through Code.")

Manually Securing a Database

Now that you've gone through many of the details on how to create new users and groups, set permissions, change ownership, and the like, step back and review the entire process of

securing your database in Access. The following steps, if followed in order, will ensure that your database is secured to the maximum degree that Access allows:

1. Create a new workgroup information file. By doing so, you in turn create a secured Admins SID.
To create a new workgroup information file, run the Wrkgadm.exe file in the \Access folder. Click Create and provide a new Name, Company, and Workgroup ID. Save the file as something other than System.mdw. One suggestion is to keep it the same name as your applications, except with an .mdw extension—for example, MyApp.mdw.
2. Start Access. Don't open a database.
3. Add a password to the Admin user account. From the Tools menu, choose Security and then User and Group Accounts. On the Change Logon Password page, leave the Old Password text box blank; add and verify the new password in the text boxes provided.
4. Create a new Admin account. This new user will be the owner of the database and all its objects. To create a new Admin account, from the Tools menu choose Security and then User and Group Accounts. On the Users page, click the New button. Provide a new Name and Personal ID. (Remember, the longer the name and PID, the better the encryption.) For this example, use MyAdmin for the Name.
5. Add the new Admin to the Admins group. To do this from the same dialog and the Users page, select your new user's name from the Name combo box. (In this example, select MyAdmin.) In the Group Membership section, highlight Admins and choose Add to move the Admins account to the Member Of section.
6. Exit and restart Access. In the Logon dialog, enter the new user's name in the Name text box. For this example, type **myadmin** (case isn't sensitive for logging on). Do *not* enter a password; currently, this user has no password.
7. For added protection, add a password to your new administrator's account. To do this, follow the directions in step 3.
8. Remove Admin from the Admins group to ensure that if somebody does find out the password to the Admin user, it won't have any permissions. To do this, follow the same directions as in step 5, *except* make sure that you select Admin for the user, highlight Admins from the Member Of section, and choose Remove.
9. Restart Access to ensure that the Admin user is removed from the Admins group, and the only user left in your Admins group is your new Admin. When prompted to log on, enter your new Admin's name and any password you might have created in step 7.
10. Create a new, secured database and name it MyApp.mdb for this example. To create a new database, either choose Blank Database from the Access startup screen, or choose New Database from the File menu and then choose Blank Database, followed by OK.

11. Encrypt the database to protect you from others snooping through your database with a word processor or some other file-utility viewer. To encrypt your database, close the database, and from the Tools menu choose Security and then Encrypt/Decrypt Database.
12. Create any application-specific users and groups (optional).

Remember, security shouldn't be an afterthought; if it is, it's a lot more work for the developer. If you don't know all your users and groups at this point, you can add them later, but if you add them now, it will save time and could possibly prevent some security holes. At this point, the groups are the most important accounts. Because traditionally you'll be assigning users to groups and that's where they will be getting their permissions, groups should be created here.
13. Set up default permissions. By establishing default permissions up front, you'll save yourself a ton of work later. If you know you don't want any groups except the Admins group (or only your new Admin account) to have access to your forms, remove all permissions from all the groups (especially the Users group) from the New Forms object. To do this, open your newly secured, encrypted database; then from the Tools menu, choose Security and then User and Group Permissions. On the Permissions page, choose Form from the Object Type combo box. With New Forms selected in the Object Name list, clear all the check boxes under the Permissions section.
14. Create (or import) all your objects to make your new Admin the owner of all objects in the database.

CAUTION

If you're importing objects from another database, make sure that all queries have the Run Permissions property set to User's. If they don't, they might not run when imported to the new secured database. When they're imported, you can safely reset the property back to Owner's. For more information about the Run with Owner's Permission query property, see the later section "Running with Owner's Permissions."

15. Set database permissions. Remove the Open Exclusive permission from all your users and groups, except users who'll need to compact or repair the database. Remove the Administer permissions from all users and groups except your database Administrator user. Remove Open/Run permissions from any users and groups you don't want accessing your database.
16. Run the `ap_CanNotCreateDatabase()` and `ap_CanNotCreateTables()` functions (optional, but highly recommended). `ap_CanNotCreateDatabase()`, discussed later in the section "Denying Users the Ability to Create Databases," prevents users of your

workgroup information file from creating new databases. `ap_CanNotCreateTables()`, discussed in the section “Denying the Creation of Table and Query Objects,” prevents users from creating new table and query objects in your database.

17. Open the VBE. Choose *Databasename* Properties from the Tools menu. On the Protection page, click the Lock Project for Viewing check box. Finally, add and confirm a password.

Provided that you follow all these steps in order, your database will be as secure as the Access environment can make it. You and your client can feel good knowing that unwanted eyes can't see the data or the code.

Which Permissions Should I Set?

Nobody can tell you exactly which permissions to set on your database to make it secure. Each database is different and fulfills different needs for different users. With the creation of multiple users and groups, the management of security becomes even more of a task. The following are a few good basic guidelines:

- Never assign permissions to individual user accounts.
- Use as few groups as possible, but do use them.
- Assign permissions only to groups.
- Remove all permissions from the Users group and Admin user accounts.
- Always remove the Admin user from the Admins group.
- Create all objects (including the database) with one user who's a member of the Admins group.
- Keep as few users as possible in the Admins group. If you can stick with just one, great!
- Be sure to set the database properties.

For a database with only one type of user (all users requiring the same permissions), create a group called MyUsers and then use the permissions listed in Table 20.2 as a base.

TABLE 20.2 Suggested Default Permissions on a Simple Application with Only One Group

<i>Object</i>	<i>Permissions</i>
Tables	No permissions for any users or groups.
Queries	No permissions for Admin or Users; Read Definition, Read Data (and Update/Insert/Delete Data where applicable), and RWOP (Read With Owner's Permission) query properties set on all queries for the MyUsers group.

TABLE 20.2 Continued

<i>Object</i>	<i>Permissions</i>
Forms	No permissions for Admin or Users; Open/Run permissions for MyUsers group.
Reports	Same as Forms. Make sure that all reports have their printer property set to the default printer.
Macros	Same as Forms.
Modules	Protected as outlined earlier in the section “Securing Modules in the VBE,” where a password is specified for the VBA code in the project and the password is given only to developers and admins.
Database	Open/Run permissions only for MyUsers group and owner account. Open Exclusive only for the owner’s account. Don’t use Administer for any groups.

Steps to Unsecure a Database

Because databases can be secured, you might also need to know how to unsecure a database. One reason it’s good to know how to unsecure a database is if you need to upgrade your version of Access. To unsecure a database, follow these steps:

1. Start Access, logging on as a member of the Admins group.
2. Change any queries that have their Run Permissions property set to Owner’s to User’s.
3. Give the Users group full permission on all objects in the database.
4. Exit Access.
5. Restart Access, logging on as Admin.
6. Create a new database.
7. Import all the objects from your originally secured database.

This new database is completely unsecure. Anybody with a copy of Access has full permissions to this database and all its objects.

TIP

Another way to unsecure a database is to make sure that the Admin user has Read Design and Read Data permissions on all the objects in the database, and then run the User-Level Security Wizard while logged on as Admin. The following section covers using the Security Wizard.

Making Life Easier with Access Security Tools

Access security is a complex topic, but luckily some development tools are available to help ease the pain of implementing security in your applications. The following sections discuss the tools available to help you handle security: the Security Wizard and the Security white paper.

Using the Security Wizard

The Security Wizard that ships with Access is officially known as the User-Level Security Wizard. This is the fifth generation for the Security Wizard. The original Security Wizard was nothing more than a giant SendKeys function.

The User-Level Security Wizard allows you to choose the database objects types you want to secure, and it imports relationships, toolbars, and import/export specifications. It also automatically encrypts your newly secured database. If you try to run the User-Level Security Wizard while logged on as the default Admin user, it will require you to create a new workgroup file.

To run the User-Level Security Wizard, follow these steps:

1. Start Access, logging on as the user who you want as the new owner of the database and all its objects.
2. Open the database you want to secure.
3. From the Tools menu, choose Security and then User-Level Security Wizard.
4. Follow the dialogs until your database is secure.

There is no step in securing your application that can't be performed by using the wizard, including adding users and groups.

Printing Users and Groups from Access

Although the User-Level Security Wizard takes the prize for being the most useful security tool available, you shouldn't overlook another tool that's provided for you in the Access user interface. You can get a detailed report of the users and groups from the workgroup information file. To print a detailed report listing all the users and groups (or just the users or groups), follow these steps:

1. Start Access, logging on as a member of the Admins group.
2. From the Tools menu, choose Security and then User and Group Accounts.
3. On the Users page, click the Print Users and Groups button.
4. Choose Both Users and Groups, Only Users, or Only Groups.

Reading the Security White Papers

Several papers have been written on Access security. By far the most popular is the Security white paper, a must-read for any developer planning on using security in an Access database.

In addition to the main Security white paper, a Programming Jet Security white paper is available. This paper goes into the programming intricacies that are touched on at the end of this chapter.

Both papers have been presented at a number of conferences and are updated for each version of Access.

Using Other Security Resources

In addition to the traditional documentation that comes with the product and Microsoft white papers, other resources are available. Just about every conference that features Access has at least one session devoted to security. Finally, a few of the advanced third-party developer books are devoting a chapter or two to security. Wherever you are, some help is available when it comes to implementing security in Access.

Avoiding Common Pitfalls Found in Access Security

The following sections discuss a handful of issues that developers new to security might often encounter. Some topics covered include planning security, running permissions on queries, securing attached tables, and dealing with life after the Security Wizard.

Planning Security

You should plan your security model at the same time you plan your database design. Planning ahead and implementing security along the way gives you less chance for leaving security holes in your application. Often when a project gets big and hundreds of objects are created, there's a great possibility that an object or two might be overlooked and not be secured properly if you wait until the last minute to secure the database.

One technique that's a good practice to follow is setting up all your users and groups first, before creating any objects. Then use the default permissions for new objects settings for each object.

Creating Objects with Default Accounts

It can't be stressed enough that any objects—including database objects—created with the default user Admin account are never secure. Furthermore, you should always create a

brand-new workgroup information file before starting that new project that must be secured. It's of the utmost importance that your Admins group SID is unique—that's what makes security work. (Recall that the only way to create a new SID for the Admins group is to use the Workgroup Administrator.) For more information about how to create a new workgroup information file, see "Creating a Workgroup Information File" earlier in this chapter.

After the new workgroup information file is created, you can proceed to create a new developer account, add that account to the Admins group, log on with the new account, and create the application—of course, making sure that you remove the Admin account from the Admins group.

Securing Linked Tables in a Multiuser Environment

Often, Access developers find it beneficial to create a multiple-database system. Splitting the code from the data allows the code to run locally on each machine and the data to reside on a server. This situation gives you performance gains and makes it easier to manage updates. But how do you set up security in this type of situation?

One suggestion is to totally lock up the data database. The only things that should be in the data database (as opposed to the code database) are tables and system relationships. Remove all permissions from all users and groups. Also remember to encrypt the database. Because the database will be up on a file server where many people have access to it, you need to ensure that they can't read the database if they copy it to their local machine; encryption will solve this problem. Because data will be added, deleted, and modified, you need to leave the network operating system permissions set to Read, Write, and Update on the data file.

Now that Windows will allow writes to the file, you need to set up Access permissions so that the code database can manipulate the data database. Next, create links to each table, and then remove all permissions to all users and groups for those linked table objects. Any time you need to access the data through a form or a report, base the form's or report's record source to a query with the Run Permissions property set to Owner's. This ensures the most protection of your sensitive data in the data database and keeps people from looking at the tables through the code database, except through the mechanisms (forms and reports) that you provide.

For ease of user administration, it's suggested that you keep your workgroup information file (System.mdw) on the file server with your data database. This is especially important if you're doing a lot of user administration, such as creating users and groups or moving users between groups. This is also very helpful if you let users change their own passwords.

Running with Owner's Permissions

Perhaps the most complex mechanisms built into the Access security model is the Run Permissions query property. If you can master the concept of the `WITH OWNERACCESS OPTION`

SQL clause for queries, you'll have no problem implementing the most advanced features of Access security.

The purpose for the Owner access option is to let people with no permissions for a table look at a limited set via a query. Typically, you need to have at least Read Data permission to view data from a table. The problem arises when you don't want to allow viewing of the full table. For example, you might have an Employees table with a salary field. You might want to use this table to create a phone book listing for a form by showing the FirstName, LastName, and PhoneExt fields in a query. To execute the query, you must have Read Data on the table, thus the dilemma. You need to give partial access to the table. (This is the functional equivalent of SQL Server column-level security.)

Recall the two users from earlier in the chapter, Ryan the manager and Jennifer the employee. As a member of the Managers group, Ryan might have full access to update the Employees table. You don't, however, want to give Jennifer access to the table because she shouldn't be able to view everybody else's salary. On the other hand, that's your only Employees table and you've created a query that returns employee names and phone extensions from that table.

What Access lets you do in this situation is create the table and remove all permissions from the Employees and Users groups. Then, if you made the table and the query, you can set the Run Permissions property to Owner's. Doing this grants the user executing the query access to the base table for just that query. By setting the query's permissions to Read Design and Read Data only for the Employees group, all employees can look at the phone listing without having access to the base table.

It's suggested that you use this method on all linked tables when you're using a two-database (code plus data) system.

CAUTION

Changing ownership of a query with its Run Permissions property set to Owner's could potentially render the query useless. To properly change a query's owner, make sure that the new owner has the appropriate permissions on the table object. Save the query with Run Permissions set to User's. Change the owner by copying and pasting, or by importing/exporting the query object and then resetting the Run Permissions property to Owner's.

TIP

You can change the default Run Permissions property for new queries from User's to Owner's from the Tools menu by choosing Options, selecting the Tables/Queries tab, and selecting from the Run Permissions section.

All queries with their Run Permissions property set to Owner's will end with the `WITH OWNERACCESS OPTION` statement in their SQL clause.

To view the Run Permissions property of a query, follow these steps:

1. Open the query in Design view.
2. From the View menu choose Properties.
3. Double-click in the upper half of the query design grid so that the property sheet displays the query properties. The Run Permissions property is a toggle field six rows down the property sheet.

Using Security in a Replication Environment

You need to be aware of several issues when dealing with security in a replication environment. First, database passwords can't be used on a replicated database. If you want to implement security, full user-level security must be used.

Never replicate the workgroup information file (System.mdw). Serious repercussions can result if a workgroup information file has a conflict during synchronization.

Distributing Secured Applications with the Microsoft Office Developer

The biggest thing to remember when you're going to distribute an application is that the `/runtime` command-line argument is no substitution for full user-level security.

If you rely only on the runtime environment (that is, no Design mode or database container) for protection of your application, it can be broken into with any copy of a full version of Access. Remember, Access can't compile its applications into executables to protect code and data.

Distributing Secured Applications Through an .mde File

In Access, you can create an .mde file for distribution. If your database contains VBA code, saving your database as an .mde file compiles all modules, removes all editable source code, and compacts the destination database. Because the editable code is removed, memory is saved, thus increasing performance.

CAUTION

Before you create an .mde file, save the database to another name. You need to have a clean copy saved, otherwise you can't un-MDE the file and you won't be able to convert it to future versions of Access.

To save a file as an .mde file, choose Database Utilities from the Tools menu. Then choose Make MDE File. You're presented with the Database to Save As MDE dialog. Because this dialog is one of the common file dialogs, simply locate the file and then click Make MDE. When creating the MDE for a ADP, the extension will be ADE.

Managing Security Through Code

Theory has been covered, as have user interfaces, tools, and several pitfalls. Now it's time to start manipulating security through code. The rest of this chapter is dedicated to providing you with several code examples that you can use to help manipulate Access security.

Programming Security with DAO

When it comes to programming security, you must understand the relative position of security in the Data Access Objects (DAO) hierarchy. If you have a good handle on DAO, creating a new user programmatically is just as easy as creating a recordset programmatically. Appendix C, "Working with Data Access Objects," discusses DAO in detail. (This appendix is available on this book's Web page at www.sampublishing.com.) Figure 20.12 shows the Users and Groups as they are in position in the Data Access Objects object model.

Access didn't provide programmatic access to security until version 2.0. Before DAO, there were SendKeys and DoMenuItems. (As I mentioned earlier, the very first Security Wizard was a giant SendKeys looping structure.) Access has come a long way in providing programmatic security features for power programmers.

Data Access Objects Object Model

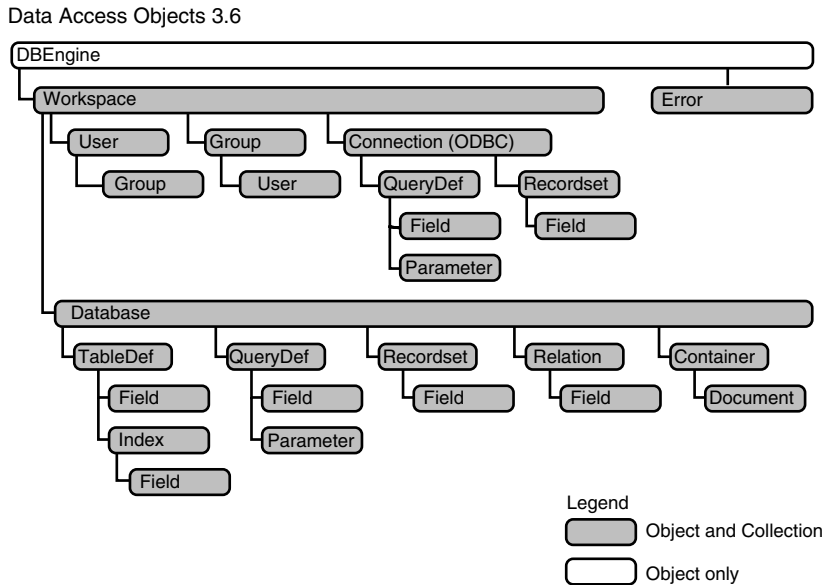


FIGURE 20.12

Notice the Users and Groups collections within the DAO hierarchy.

The User and Groups Collections

The Users and Groups collections reside off the Workspace object. It's at this level that the following functions can be performed:

- Creating and deleting users
- Creating and deleting groups
- Adding, removing, and changing passwords
- Adding and removing users in groups

Permissions and the Documents Collection

Permissions are stored for each object in the Documents collection of a container object. It's at this level that the following functions can be performed:

- Changing the owner of an object
- Setting database permissions
- Adding, removing, and changing object permissions

NOTE

You must be a member of the Admins group to create/delete users and groups. When it comes to setting permissions, you must be a member of the Admins group or the owner of the object.

Creating a New User Through Code

Listing 20.1 shows how to programmatically create a new user by using the `ap_CreateUser()` function. This function accepts two or three parameters: Name, PID, and optionally Password.

NOTE

This function includes code to automatically add the user to the Users group. This doesn't happen automatically through code; you must append the user to the Users group. If you don't do this, the user won't have access to the Utility.mda file and won't be able to start Access.

LISTING 20.1 Chap20.mdb: Creating a User

```
Function ap_CreateUser(strUserName, strUserPID, Optional strUserPassword)
'Note that the password parameter is optional.
  On Error GoTo ErrCreateUser
  Dim usrNewUser As User
  Dim wspNewWorkspace As Workspace

  Set wspNewWorkspace = DBEngine(0)

  'The next line creates the user account with the parameters passed in.
  Set usrNewUser = wspNewWorkspace.CreateUser(strUserName, strUserPID, _
    strUserPassword)

  'The next line makes the user available this line is required.
  wspNewWorkspace.Users.Append usrNewUser

  'This next line is optional but is STRONGLY suggested
  'to follow the Microsoft Access User Interface model
  'of adding all new users to the Users group.
  usrNewUser.Groups.Append wspNewWorkspace.CreateGroup("Users")
```

LISTING 20.1 Continued

```
'Function successful
Exit Function

ErrCreateUser:
  MsgBox "Function 'ap_CreateUser' did not complete successfully."
  Exit Function

End Function
```

NOTE

All the sample code from this chapter is in the Chap20.mdb database, which you can find on this book's Web page at www.sampublishing.com.

As you can see from Listing 20.1, you have two choices when you call this function. For example, you can pass a name and a PID parameter:

```
=ap_CreateUser("Chris", "123abc")
```

Or you can provide the optional password parameter:

```
ap_CreateUser("Jonathan", "999zzz888YYY", "MySecret")
```

The first example creates a user named Chris with no password; the second example creates a user named Jonathan with a password of MySecret.

Deleting a User Through Code

Listing 20.2 shows how to programmatically delete an existing user. This function accepts one parameter: the name of the user you want to delete.

LISTING 20.2 Chap20.mdb: Deleting a User

```
Function ap_DeleteUser(strUserName)
  On Error GoTo errDeleteUser
  Dim strUser As String
  Dim wspNewWorkspace As Workspace

  Set wspNewWorkspace = DBEngine(0)

  'The next line gets the name of the user passed in as a parameter.
  strUser = wspNewWorkspace.Users(strUserName).Name
```

LISTING 20.2 Continued

```
'The next line actually deletes the user.
wspNewWorkspace.Users.Delete strUser

'Function successful
Exit Function

errDeleteUser:
  MsgBox "Function 'ap_DeleteUser' did not complete successfully."
  Exit Function

End Function
```

You would use the code in Listing 20.2 to delete a user in the following fashion:

```
=ap_DeleteUser("Ryan")
```

Notice that no PID or password is required when deleting a user. However, you have to be logged on as a member of the Admins group to delete a user. This is also true for creating a user.

Setting the Database Password Through Code

Listing 20.3 shows how to programmatically set the database password for a share-level security model. This function accepts two parameters: the new password and the previous password.

LISTING 20.3 Chap20.mdb: Setting a Database Password

```
Function ap_DatabasePassword(strDBPassword, strDBPrevPass)
' strDBPassword is the new password for the database.
' strDBPrevPass is the previous password for the database.
  On Error GoTo errDatabasePassword
  Dim db As DATABASE
  Set db = CurrentDb()

  'This line adds the new password to the database.
  db.NewPassword strDBPrevPass, strDBPassword

  'Function successful
  Exit Function
```

LISTING 20.3 Continued

```
errDatabasePassword:
    MsgBox "Function 'ap_DatabasePassword' did not complete successfully."
    Exit Function

End Function
```

In Listing 20.3, if you're setting the database password for the first time, you pass in an empty string for the previous password parameter—for example,

```
=ap_DatabasePassword("MyNewPass", "")
```

This function is also useful if you want to clear the database password. You simply pass in, through the function's arguments, the proper previous password and the empty string for the new password—for example,

```
=ap_DatabasePassword("", "OldPassword")
```

NOTE

This code is great to use if you want to create your own menu item (or toolbar button) that lets users change the database password.

Creating a Group Through Code

Listing 20.4 shows how to programmatically create a new group. This function accepts two parameters: the name of the group and a PID.

LISTING 20.4 Chap20.mdb: Creating a Group

```
Function ap_CreateGroup(strGroupName, strGroupPID)
'Pass in the group name and group PID as string parameters.
    On Error GoTo errCreateGroup
    Dim usrNewGroup As Group
    Dim wspNewWorkspace As Workspace

    Set wspNewWorkspace = DBEngine(0)

    'The next line creates the group account with the parameters passed in.
    Set usrNewGroup = wspNewWorkspace.CreateGroup(strGroupName, strGroupPID)
```


LISTING 20.4 Continued

```
'The next line makes the user available this line is required.
wspNewWorkspace.Groups.Append usrNewGroup

'Function successful
Exit Function

errCreateGroup:
MsgBox "Function 'ap_CreateGroup' did not complete successfully."
Exit Function

End Function
```

This function is very similar to the `ap_CreateUser()` function in Listing 20.1, except that you don't need to worry about appending it to the Users collection. Also, there are no passwords for groups. Here is an example of how this function might be used to create a group named Accountants:

```
=ap_CreateGroup("Accountants", "$Money$")
```

Deleting a Group Through Code

Listing 20.5 shows how to programmatically delete an existing group. This function accepts one parameter: the name of the group you want to delete.

LISTING 20.5 Chap20.mdb: Deleting a Group

```
Function ap_DeleteGroup(strGroupName)
'This function takes one parameter, the name of the group to be deleted.
On Error GoTo errDeleteGroup
Dim strGroup As String
Dim wspNewWorkspace As Workspace

Set wspNewWorkspace = DBEngine(0)

'The next line gets the name of the group passed in as a parameter.
strGroup = wspNewWorkspace.Groups(strGroupName).Name

'The next line actually deletes the group.
wspNewWorkspace.Groups.Delete strGroup

'Function successful
Exit Function
```

LISTING 20.5 Continued

```
errDeleteGroup:
    MsgBox "Function 'ap_DeleteGroup' did not complete successfully."
    Exit Function

End Function
```

This function has almost the exact same syntax as deleting a user. You just provide it a group name, and it will delete the group. Remember that you must be a member of the Admins group for this code to run correctly.

Here's an example of how to call this function:

```
=ap_DeleteGroup("Managers")
```

Adding a User to a Group Through Code

Listing 20.6 shows how to programmatically add a user to a group. This function accepts two parameters: the user's name and the name of the group you want to add the user to.

LISTING 20.6 Chap20.mdb: Adding a User to a Group

```
Function ap_AddUserToGroup(strUserName, strGroupName)
    'Two parameters; the user name and the group name
    On Error GoTo errAddUserToGroup
    Dim grpGroup As Group
    Dim wspWorkspace As Workspace

    Set wspWorkspace = DBEngine(0)
    Set grpGroup = wspWorkspace.Groups(strGroupName)

    'This line actually adds the user to the group.
    grpGroup.Users.Append grpGroup.CreateUser(strUserName)

    'Function successful
    Exit Function

errAddUserToGroup:
    MsgBox "Function 'ap_AddUserToGroup' did not complete successfully."
    Exit Function

End Function
```

This function can be called as follows:

```
=ap_AddUserToGroup("Jennifer", "Employees")
```

In this example, user Jennifer would be added to the group Employees. Both Jennifer and Employees must exist for this function to complete successfully.

Removing a User from a Group Through Code

Listing 20.7 shows how to programmatically remove a user from a group. This function accepts two parameters: the user's name and the name of the group you want to remove the user from.

LISTING 20.7 Chap20.mdb: Removing a User from a Group

```
Function ap_RemoveUserFromGroup(strUserName, strGroupName)
'Two parameters, a user and a group
  On Error GoTo errRemoveUserFromGroup
  Dim strGroup As String
  Dim usrUser As User
  Dim wspWorkspace As Workspace

  Set wspWorkspace = DBEngine(0)
  Set usrUser = wspWorkspace.Users(strUserName)

  'The following two lines perform the deletion of the user from the
  'group given the passed in parameters.
  strGroup = usrUser(strGroupName).Name
  usrUser.Groups.Delete strGroupName

  'Function successful
  Exit Function

errRemoveUserFromGroup:
  MsgBox "Function 'ap_RemoveUserFromGroup' did not complete successfully."
  Exit Function

End Function
```

In Listing 20.7, when you pass in the user and group as parameters, both must previously exist or the function won't complete successfully. Here's an example of a call to this function:

```
=ap_RemoveUserFromGroup("Ryan", "Managers")
```

CAUTION

Be careful when using this function. Remember, through code you can remove a user from the Users group; however, doing so doesn't conform to Access security standards.

Changing the Owner of an Object Through Code

The code in Listing 20.8 programmatically changes an object's owner. This function accepts three parameters: the new owner name, the object name, and the object type. The new owner name can be a user or a group. Valid object types are tables, queries, forms, and reports.

LISTING 20.8 Chap20.mdb: Changing the Owner of an Object

```
Function ap_ChangeOwner(strNewOwner, strObjectName, strObjectType)
'This function requires an object type to be passed in because more than one
'object can have the same name (i.e. a Form and a Table named Employees).
'Valid object types include: Tables, Queries, Forms, and Reports.
  On Error GoTo errChangeOwner
  Dim db As Database
  Dim con As Container
  Dim doc As Document

  'The next three lines of code takes the object type and
  'object name parameters to find the object.
  Set db = CurrentDb
  Set con = db.Containers(strObjectType)
  Set doc = con.Documents(strObjectName)

  'Test to see if the user has permissions to perform this operation;
  'if true, then changes ownership
  If (doc.AllPermissions And dbSecWriteOwner) <> 0 Then
    doc.Owner = strNewOwner
  Else
    MsgBox "You don't have permissions to change the" _
      & " ownership of object: " & strObjectName
    Exit Function
  End If

  'Function successful
  Exit Function
errChangeOwner:
```

LISTING 20.8 Continued

```

errChangeOwner:
    MsgBox "Function 'ap_ChangeOwner' did not complete successfully."
    Exit Function

End Function

```

Many things can be noted about Listing 20.8. First, remember that groups can own objects; they just can't create them. So the parameter `strNewOwner` can be an existing user or group.

Also note that it uses the `AllPermissions` property and the `dbSecWriteOwner` security constant to determine whether the user has permissions to perform this operation. (For more information about the `AllPermissions` property and security constants, consult the Answer Wizard in online help.)

Finally, look at the function's third parameter. If you want to change the owner of a macro, you must use the keyword `Scripts`.

Setting Permissions for an Object Through Code

Listing 20.9 shows how to programmatically set permissions for an object. This function accepts three parameters: the name for which you want the permissions set, the object name, and the object type. The name can be a user or a group. Valid object types are tables, queries, forms, and reports.

LISTING 20.9 Chap20.mdb: Setting the Permissions of an Object

```

Function ap_AssignPermission(strUserName, strObjectName, strObjectType, _
    flgPermission)
    'This function requires an object type to be passed in because more than one
    'object can have the same name (i.e. a Form and a Table named Employees).
    'Valid object types include: Tables, Queries, Forms, and Reports.

    'strUserName can be any valid user or group.
    'flgPermission can be any of security permission constants

    On Error GoTo errAssignPermission
    Dim db As DATABASE
    Dim con As Container
    Dim doc As Document

    'The next three lines of code takes the object type and
    'object name parameters to find the object.
    Set db = CurrentDb

```

LISTING 20.9 Continued

```

Set con = db.Containers(strObjectType)
Set doc = con.Documents(strObjectName)

'The next line of code sets the user (or group).
doc.UserName = strUserName

'The next line of code does the actual sending of the permission.
doc.Permissions = doc.Permissions Or flgPermission

'Function successful
Exit Function

errAssignPermission:
  MsgBox "Function 'ap_AssignPermission' did not complete successfully."
  Exit Function

End Function

```

The `ap_AssignPermissions()` function takes four parameters. The first parameter is the user name you want to assign permissions to, followed by the object name, the object type, and the security permission constant you want to assign. Here's an example:

```
=ap_AssignPermission("Scott", "[Chapter 20]", "Forms", acSecModWriteDef)
```

In this example, the user Scott would get Write Definition permissions on the Chapter 20 form.

Checking Permissions Through Code

Listing 20.10 shows how to programmatically check to see whether a user has permissions to a specific object. This function accepts three parameters: the user name, the object name, and the object type. Valid object types are tables, queries, forms, and reports.

LISTING 20.10 Chap20.mdb: Checking Permissions

```

Function ap_CheckPermissions(strUserName, strObjectName, strObjectType, _
    flgPermission)
'The next line of code sets the user (or group).
doc.UserName = strUserName

'The next line of code does the actual sending of the permission.
doc.Permissions = doc.Permissions Or flgPermission

'Function successful
Exit Function

errCheckPermissions:
  MsgBox "Function 'ap_CheckPermissions' did not complete successfully."
  Exit Function

End Function

```

LISTING 20.10 Continued

```
'The next three lines of code takes the object type and
'object name parameters to find the object.
Set db = CurrentDb
Set con = db.Containers(strObjectType)
Set doc = con.Documents(strObjectName)

'The next line of code sets the user
doc.UserName = strUserName

' Check if flag against the AllPermissions property.
If (doc.AllPermissions = flgPermission) Then
    ap_CheckPermissions = True
Else
    ap_CheckPermissions = False
End If

'Function successful
Exit Function

errCheckPermissions:
MsgBox "Function 'ap_CheckPermissions' did not complete successfully."
Exit Function

End Function
```

The code in Listing 20.10 takes four parameters to allow this function to be as flexible as possible. You can check different permissions for different users on different objects. Here's an example of how this function might be called:

```
=ap_CheckPermissions("Ryan", "[Chapter 20]", "Forms", dbSecFullAccess)
```

In this example, the function would return a True or False if the user Ryan had Administer permissions on the Chapter 20 form.

TIP

You can use this code in a startup switchboard to hide and show certain buttons that would take users into different areas of the application based on their job positions. For example, the Managers group might have permissions to view several types of reports that members of the Employees group couldn't see.

Determining Who You're Logged On As Through Code

Often, developers want to know who they're logged on as. This is extremely useful information when you're creating a secured application and testing many of the different user accounts. The following code shows how to programmatically detect what user you're logged on as by using VBA's `CurrentUser()` function:

```
Function ap_WhoAmI()
    Dim Title As String
    Title = "Who Am I?"
    MsgBox "You are logged in as: " & CurrentUser(), , Title
End Function
```

This little code example is handy to have around. Again, it's nothing that you can't do through the Access user interface: You can always see who you're logged on as from the Tools menu by choosing Security and User and Group Accounts and then clicking the Change Logon Password tab.

Denying Users the Ability to Create Databases

Listing 20.11 is one of the most powerful coding examples in all of security. This function can be performed only through code; Access doesn't provide a way through the user interface to perform this operation. The code shows how to programmatically remove a user's ability to create a database. This function takes one parameter: the name of the user (or group) you want to deny database creation permissions.

LISTING 20.11 Chap20.mdb: Denying Database Creation Permissions

```
Function ap_CanNotCreateDatabase(strName)
'For this function you can pass in any valid User or Group Name.
    On Error GoTo errCanNotCreateDatabase
    Dim dbSystemDB As Database
    Dim conContainer As Container
    Dim strSysPath As String

    'The next line gets the path to the Workgroup Information File
    strSysPath = SysCmd(acSysCmdGetWorkgroupFile)

    'The next two lines open the Workgroup Information File
    'and grab the Databases object within the Containers collection
    Set dbSystemDB = DBEngine(0).OpenDatabase(strSysPath)
    Set conContainer = db.Containers!Databases

    'The next line of code sets the user or group name from the
    'passed-in parameter
```


LISTING 20.11 Continued

```
conContainer.UserName = strName
'The next line of code removes the permissions to create a database
conContainer.Permissions = conContainer.Permissions And Not dbSecDBCCreate

'Function successful
Exit Function

errCanNotCreateDatabase:
MsgBox "Function 'ap_CanNotCreateDatabase' did not complete successfully."
Exit Function

End Function
```

Listing 20.11 shows the use of the `SysCmd()` function, which performs a multitude of tasks. In this example, `SysCmd()` is retrieving the path to the workgroup information file. For this function to work, it must modify permissions in the workgroup information file, not in the current database. You might also notice the use of another security constant, `dbSecDBCCreate`.

The code in Listing 20.11 works only for your current workgroup information file. If users create their own or join a different workgroup information file, their user accounts in those workgroup information files will be able to create databases.

NOTE

For this function to be effective, you must apply it to the user and all groups the user is a member of. That would probably mean passing it to the Users group also.

Denying the Creation of Table and Query Objects

Often, you don't want users to create new table objects in your databases and fill them with unnecessary data. Listing 20.12 shows you how to programmatically deny users the ability to create new table and query objects in your databases. This function takes one parameter: the name of the user who's being denied creation rights.

LISTING 20.12 Chap20.mdb: Denying Table and Query Creation

```
Function ap_CanNotCreateTables(strName)
'For this function you can pass in any valid User or Group Name.
'It denies the user from creating Tables and Queries all at once.
```

LISTING 20.12 Continued

```
On Error GoTo errCanNotCreateTables
Dim db As DATABASE
Dim con As Container

Set db = CurrentDb
Set con = db.Containers("Tables")

con.UserName = strName

'This next line of code removes the table (and query) creation
'privilege from the specified user.
con.Permissions = con.Permissions And Not dbSecDBCCreate

'Function successful
Exit Function

errCanNotCreateTables:
MsgBox "Function 'ap_CanNotCreateTables' did not complete successfully."
Exit Function

End Function
```

The code in Listing 20.12 uses the database constant `dbSecDBCCreate` to perform its task.

This function can be performed only through code; there's no equivalent Access user interface option. Remember that to make this fully effective, you must run it on the user account and all the group accounts to which that user belongs. Here's how this function might be used:

```
=ap_CanNotCreateTables("Jennifer")
```

Compacting, Encrypting, or Decrypting a Database Through Code

Listing 20.13 shows how to programmatically compact, encrypt, or decrypt a database. This function accepts three parameters: the current name of the database to be compacted, encrypted, or decrypted; the new name of the database; and the flag to tell it to encrypt or decrypt. The flag parameter is optional; if it's left blank, a compact will take place. The two valid options for the flag parameter are `dbEncrypt` and `dbDecrypt`.

LISTING 20.13 Chap20.mdb: Compacting or Encrypting a Database

```
Function ap_CompactEncrypt(strCurrentDBName, strNewDBName, Optional flgEncDec)
'Use dbEncrypt or dbDecrypt for the optional flag parameters
'Or leave blank for a compact.
```

LISTING 20.13 Continued

```
On Error GoTo errCompactEncrypt

DBEngine.CompactDatabase strCurrentDBName, strNewDBName, "", flgEncDec

'Function successful
Exit Function

errCompactEncrypt:
MsgBox "Function 'ap_CompactEncrypt' did not complete successfully."
Exit Function

End Function
```

You can compact a database, for example, in the following manner:

```
=ap_CompactEncrypt("c:\Olddb.mdb", "Newdb.mdb")
```

This example takes the existing Olddb.mdb file and creates a new, totally compacted file named Newdb.mdb.

NOTE

A requirement would be Exclusive access to the database.

Also, because this isn't necessarily an action from within the database, if that database is secured with an .mdw, you need to make sure that the same .mdw is associated with the database you are running from when you call this function. It can become difficult if you are trying to do this to different databases with different .mdws.

Disabling the Bypass Key Through Code

Listing 20.14 shows how to programmatically disable the Shift key during startup of your Access application. This very important security function can be done only through code; there's no equivalent user interface option.

LISTING 20.14 Chap20.mdb: Disabling the Shift Bypass Key

```
Function ap_DisableShift()
'This function will disable the shift at startup causing
'the Autoexec macro and Startup properties to always be executed
```

LISTING 20.14 Continued

```

On Error GoTo errDisableShift
Dim db As DATABASE
Dim prop As Property
Const conPropNotFound = 3270

Set db = CurrentDb()

'This next line disables the shift key on startup.
db.Properties("AllowByPassKey") = False

'Function successful
Exit Function

errDisableShift:
'The first part of this error routine creates the AllowByPassKey
'property if it does not exist.
If Err = conPropNotFound Then
    Set prop = db.CreateProperty("AllowByPassKey", _
        dbBoolean, False)
    db.Properties.Append prop
    Resume Next
Else
    MsgBox "Function 'ap_DisableShift' did not complete successfully."
    Exit Function
End If

End Function

```

Notice the error-handling routine in Listing 20.14, starting with the line

```
errDisableShift:
```

If the AllowByPassKey property doesn't exist—which it doesn't by default—the routine creates it and sets it to False.

Using the Secured Sample Database: Chap20s.mdb

The Chap20.mdb file provided on this book's Web page is a totally unsecured database with all the code examples in this chapter. There's also a totally secured version of this database, Chap20s.mdb, provided for your use. To access this database, you need to join the Apsystem.mdw workgroup information file also provided on the Web page.

PART IV

The following users and groups have been created in Apsystem.mdw and are referenced here so that you can re-create the accounts:

<i>User</i>	<i>PID</i>	<i>Password</i>
Jennifer	L1a2B3r4I5e	<none>
Ryan	jS6uP9nU2e	MySecret
Scott	B1a2R3k4E5r	Developer
<i>Group</i>	<i>PID</i>	
Managers	S1e2C3r4E5t	
Employees	A1c2C3e4S5s	

In addition to these users and groups, the workgroup information file, Apsystem.mdw, was created with the following settings:

Name: F. Scott Barker

Company: Applications Plus

Workgroup ID: W1o2R3k4G5r6O7u8P

The original Admin that came with the workgroup information file has the following settings:

User: Admin

PID: <global across all versions of Access>

Password: a

Try opening Chap20s.mdb while logged on to your own workgroup information file and see what happens. Then try joining Apsystem.mdw and logging on as one of the provided users. Notice which users can see the code.

Summary

Security in Access can either drive you crazy or be your savior. If you understand and follow the rules for setting up and maintaining security, it can be a great tool to work with. But if you don't pay attention to details, problems can arise when you try to access objects in your database. Access gives you a number of ways in code to solve various situations that can arise when using security.

NOTE

Almost all this material applies only to Jet/MDBs. For SQL Server/ADP security, you should make an ADE file and use integrated SQL Server or Windows NT/2000 security to protect your data.

- Chapter 5, “Introducing ActiveX Data Objects,” goes into detail on working with the various ADO collections.
- Chapter 21, “Handling Multiuser Situations,” shows you how to optimize your database for handling various multiuser issues.
- Chapter 22, “Welcome to the World of Database Replication,” can help you update your systems without user intervention. Here, you also learn how to perform database replication manually and programmatically.
- Appendix C, “Working with Data Access Objects,” goes into detail on working with the various DAO collections. You can find this appendix on this book’s Web page at www.sampublishing.com (enter this book’s ISBN in the Search field).

Handling Multiuser Situations

CHAPTER

21

IN THIS CHAPTER

- Understanding Multiuser Terminology 652
- Understanding Multiuser Handling in Access 653
- One or Two Database Containers: Knowing Where to Put the Pieces 657
- Looking at the Built-In Locking Modes 661
- Working in VBA with Unbound Forms 666
- Coding for Multiuser Error Handling 685

You can take Microsoft Access out of the shrink-wrap, place it on a file server, and use it in a multiuser environment with very little programming. However, a more efficient and robust application requires more work.

Placing a single Access database that uses standard bound forms and programming on a network isn't really recommended. Performance can decrease, and users can become frustrated due to locking conflicts. A little information and minor programming can remedy this.

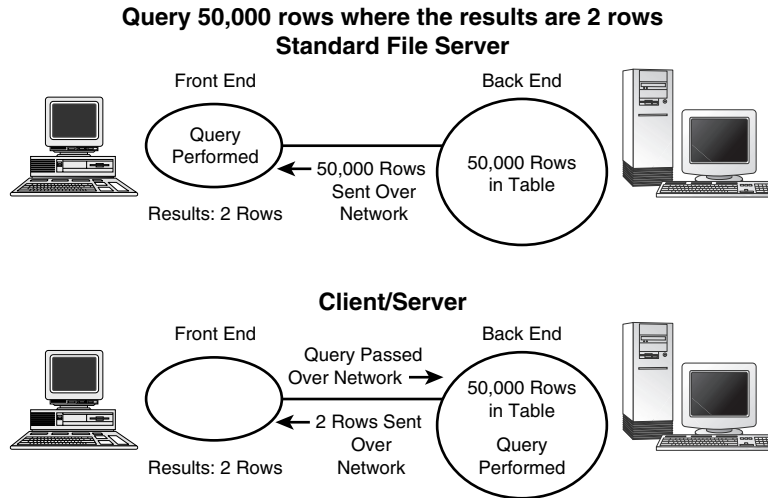
Understanding Multiuser Terminology

Before you see how Access generally handles multiuser issues, you need to understand the terminology used about the subject. The following words are commonly used in connection to networking and Access:

- **Front end, back end.** To achieve optimal performance on a network, *splitting* the application from the shared data is recommended. This means creating two databases. One is used locally on the workstation, called the *front end*; the other contains the shared data on the network, called the *back end*. Splitting a database is discussed in detail later in the section "One or Two Database Containers: Knowing Where to Put the Pieces."
- **Client/server.** Native Access uses file-server technology. When it performs requested work, such as a query, it does so by using the front end, thus passing over the network all the records used as the query's source. Access can also use a back end implemented in a client/server product, such as SQL Server. When this is the case, the query is passed through to the back end, on the server, and the work is performed there. The back end then passes only the answer over the network to the front end. Figure 21.1 diagrams the difference between file servers and client/servers. For more information on implementing Access with client/server, see Chapter 23, "Moving Workgroup Applications to Client/Server."
- **Shared, exclusive.** When a database is opened so that more than one person can access it at the same time, it's *shared*. When only one person at a time can get into the database, it's *exclusive*.

TIP

To improve performance when working with front- and back-end databases, open the front-end database *exclusively*. This is discussed in greater detail, including available command-line options, later in the section "One or Two Database Containers: Knowing Where to Put the Pieces."

**FIGURE 21.1**

You can retrieve results from large records much quicker with client/server.

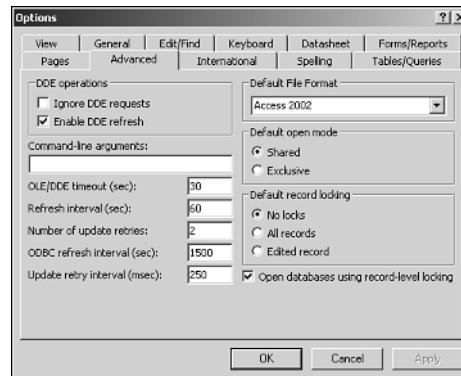
- **Local area network (LAN).** This type of network is set up in a limited area, such as in an office or business.
- **Wide area network (WAN).** This type of network usually consists of several LANs hooked together over a great distance. In any particular location, you might have only one station or many.

Understanding Multiuser Handling in Access

Now, the best way to get going with Access's multiuser features is to look at some of the options available. You can set up certain Access options for working on a network. To get started, open the Options dialog (choose Options from the Tools menu) and click the Advanced tab (see Figure 21.2). The following sections discuss each Advanced page option that deals with multiuser issues.

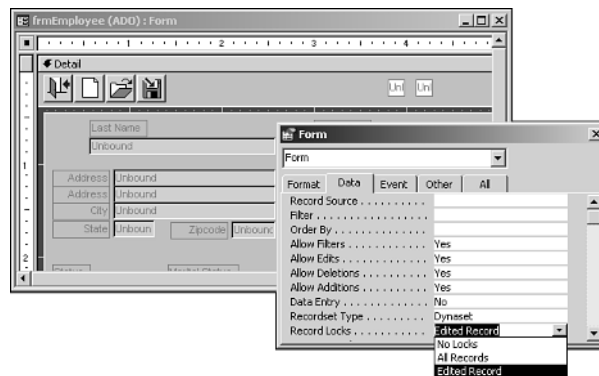
Default Record Locking

Access provides true record-level locking for forms, datasheets, and VBA code dealing with recordsets. Bulk query operations still require page-level locking, which used to be all there was. Both locking types are discussed later in the section "Record-Level Versus Page-Level Locking." When you open the Options dialog to the Advanced page, the Open Databases Using Record-Level Locking check box should be selected by default.

**FIGURE 21.2**

The Advanced page of the Options dialog lets you set multiuser options as well as other options.

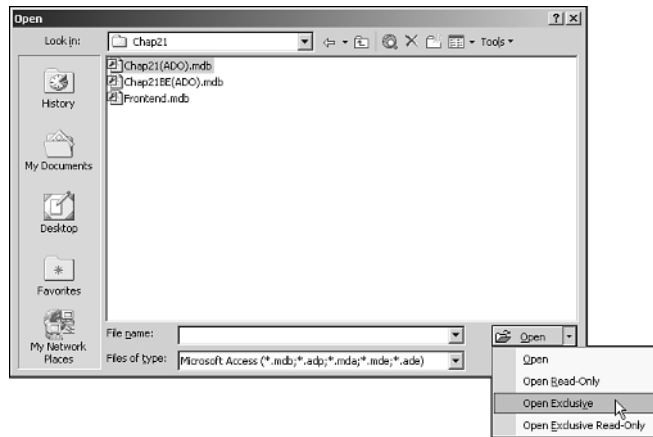
The options in the Default Record Locking section set the default mode for record locking: No Locks (the default), All Records, and Edited Record. (Later, the section “Looking at the Built-In Locking Modes” fully discusses the purpose of each option.) When you set Default Record Locking, all new forms will have this setting as their default for record locks. Figure 21.3 shows the record locks on the frmEmployee form.

**FIGURE 21.3**

Set Default Record Locking to reflect what you want the default to be on forms.

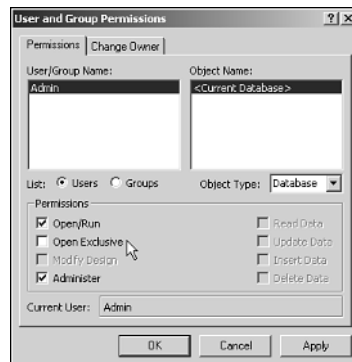
Default Open Mode: Shared Versus Exclusive

The Default Open Mode section specifies just that—it’s either shared use or exclusive use when opening a database through the Open dialog (see Figure 21.4). Open (shared) is the default.

**FIGURE 21.4**

The Open dialog reflects setting the Default Open Mode to Exclusive.

A user sometimes accidentally opens a database exclusively. If the database is on the network, this can cause problems for other users. To keep users from opening a database exclusively, you can set their permissions on the database object's Open Exclusive property to false. Figure 21.5 shows the User and Group Permissions dialog, with the Object Type set to Database. (To open this dialog, choose Security from the Tools menu, and then User and Group Permissions.) For more information on Access security, see Chapter 20, "Securing Your Application."

**FIGURE 21.5**

Notice the Open Exclusive check box available in the User and Group Permissions dialog.

Number of Update Retries

The Number of Update Retries option in the Options dialog sets the number of retries when Access is trying to save a changed record that's locked by another user. After a number of retries is reached, an error message appears. The default setting is 2; the maximum number of retries is 10.

You can trap this error in code with error handling. This and other multiuser errors, as well as how to handle them, are covered later in the section “Coding for Multiuser Error Handling.”

ODBC Refresh Interval

The *Open Database Connectivity (ODBC)* technology connects Access to other file-management systems. The most common use for ODBC is allowing Access to be used as a front end with SQL Server as a back end.

The ODBC Refresh Interval (Sec) setting indicates how often Access refreshes whatever recordsets are set up between the front end and the ODBC back end. Information in the refreshed recordsets are data changes, insertions, and deletions. The default interval is 1,500 seconds; the maximum is 3,600 seconds.

Refresh Interval

The Refresh Interval (Sec) setting is the amount of time between refreshes of recordsets on networks between the front and back ends. The default is 60 seconds; the maximum is 32,766.

TIP

If an application needs data refreshed up to the second, you can set this interval to be smaller. Remember, however, that refreshing more often increases network traffic, which can result in greater degradation for the network overall. If you start with the default, you can adjust the interval and monitor performance to get the optimal setting. It's different for various network setups.

Update Retry Interval

The Update Retry Interval is just that: the time (in milliseconds) between successive retries to update a record. The default is 250; the maximum is 1,000 milliseconds.

One or Two Database Containers: Knowing Where to Put the Pieces

When first starting with Access, most developers create a single database for all objects used with Access. This is very convenient because you have to update only one file when updating a system. This works great when it's a single-user system, but not for most multiuser systems. Performance takes a big hit when you have all your applications on the network.

Knowing What Should Go Where: An Overview

If you're working in a purely Jet environment, the ideal way to arrange your system on a network is to have your application components—queries, forms, reports, macros, and modules—in a local database. (Other components that you can keep locally for better performance are discussed shortly.) After splitting a database, you have to relink the tables, which is covered in detail in Chapter 25, “Startup Checking System Routines Using ADO.”

Shared data in tables is stored in a shared folder on the network. Figure 21.6 shows what should go where when dealing with splitting the database.

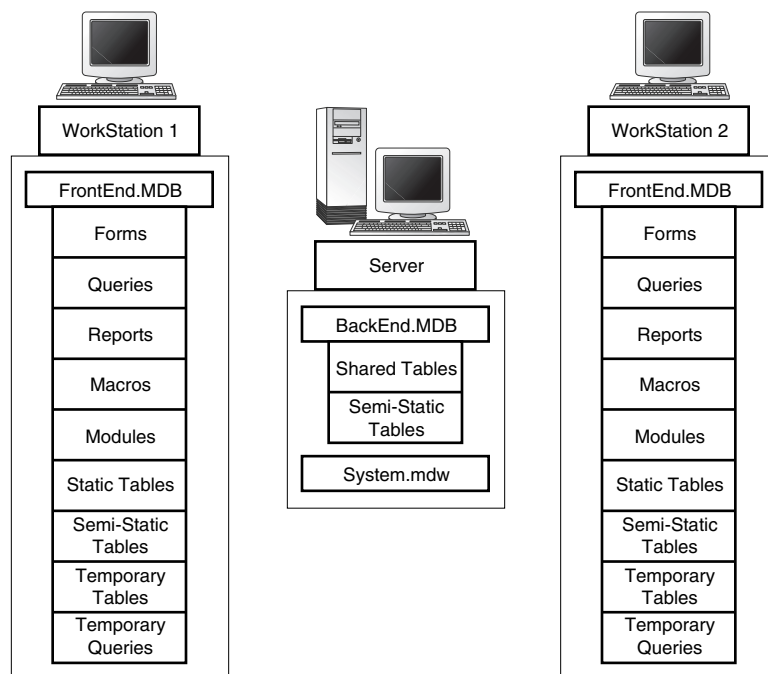


FIGURE 21.6

This diagram shows the locations of specific elements of a multiuser Access application that uses MDBs.

NOTE

Along with the database file containing the shared data, the System.mdw file should also be stored on the server. This file contains various information for a workgroup, including user and group information dealing with security. For more information about this file and security, see Chapter 20, “Securing Your Application.”

Now that you’ve seen what an Access application looks like when it’s split, look at the advantages and disadvantages, so that you have the whole picture. Overall, the benefits to splitting databases overshadow any extra tasks that arise.

Advantages to Splitting Databases

Storing certain objects of an application locally, rather than on the server, tends to increase performance. How much performance improves depends on workstation hardware, the type of network, and the performance of the file server. The following objects would be best kept locally:

- *Application objects* make up the application portion of the project: forms, queries, macros, and modules.
- *Static objects* are tables that are never updated, such as code tables. Generally, this data is updated only by developers and administrators, and then sent out with an update of the front end.
- *Semi-static objects* are infrequently updated tables, such as user tables, that developers replicate from back end to front end. More explanation and code for this technique are provided in Chapter 26, “Creating Maintenance Routines.”
- *Temporary objects* are queries and tables created on-the-fly but then deleted when the tasks are completed. Keeping these objects on the front end saves the hassle of worrying about other users overwriting your temporary tables and coming up with a naming scheme for them. Be sure to have enough local hard disk space to handle the largest size a temporary table will likely become.

NOTE

As mentioned in the preceding section, splitting the front end from the back end works best for a pure Jet environment, where the front- and back-end databases are both .mdb. When working with an ADP and client/server environment, queries likely will be stored as views or stored procedures on the server. Also, you won’t store temporary tables in the ADP, but instead will use persisted recordsets or temporary views. To find out more about this setup, see Chapter 24.

When you're working remotely and need to update the application, just sending the application database file is more efficient than sending both the application and all the data. The application likely will stay under a couple of megabytes in size when compressed, whereas the data portion will continue to grow.

You also can provide users with new versions of the application database without affecting the shared database. By updating only the application database, the data is unaffected and the update process is much smoother than a single database application.

Disadvantages to Splitting Databases

One possible disadvantage to splitting a database is that tables have to be automatically relinked. You need to include a routine to do so in your startup system checking (covered in Chapter 25). Figure 21.7 shows the dialog that appears when the back end is moved and the shared tables must be relinked.

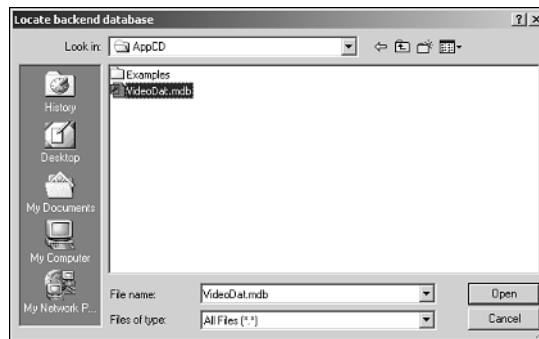


FIGURE 21.7

This routine for relinking tables opens an open file dialog to help locate the back end.

To create an ADO reference to link tables, use the following code:

```
Dim cnnCurr As New ADODB.Connection, rstCust As New ADODB.Recordset
cnnCurr.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data " & _
    "Source=E:\Net\backend.mdb;"
rstCust.Open "Customers", cnnCurr
```

Another disadvantage could be that you need to update two separate files—the front end and back end.

Splitting Databases

You can split a database in more than one way. Here's one way to do so manually:

1. Create the database as a normal single-station system with all the elements in the same database container.

2. Create a second database with the back-end name you want to use.
3. Open the original table and export the tables you want shared into the new database.
4. Delete the exported tables from the original database.
5. Link all the shared tables from the back end to the front end.

You then just have to maintain links if they get broken by a back end moving.

You also can split a database by using the Database Splitter wizard. Starting out, you perform step 1 as explained previously. To demonstrate how to use this wizard, open the database named `FrontEnd.mdb`, which contains two tables and two forms named `frmCustomers` and `frmEmployees`. You can find `FrontEnd.mdb` on the book's Web page at www.sampublishing.com.

To split this database after you open it, follow these steps:

1. From the Tools menu, choose Add-Ins and then Database Splitter. The first dialog (in Figure 21.8) warns you to back up the database and that this process could take a while.

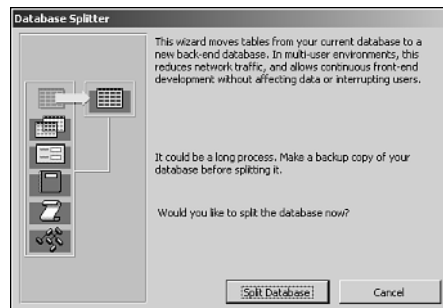


FIGURE 21.8

Heed the warnings given on this first dialog of the Database Splitter wizard.

2. Click Split Database. The Create Back-end Database dialog appears.
3. Type **backend.mdb** in the File Name text box (see Figure 21.9) and click Split. Access does some work and splits the database for you.
4. A dialog appears to tell you that the database is split. Click OK.

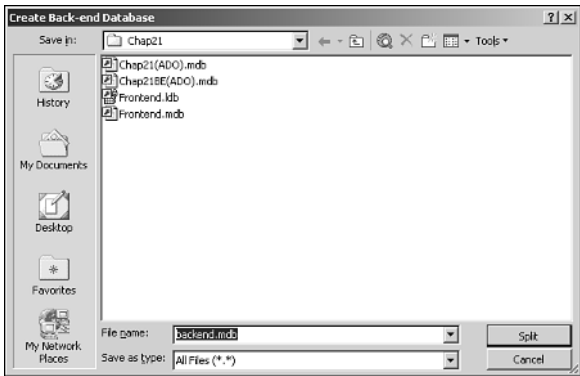


FIGURE 21.9
Use the filename backend.mdb for the new shared database.

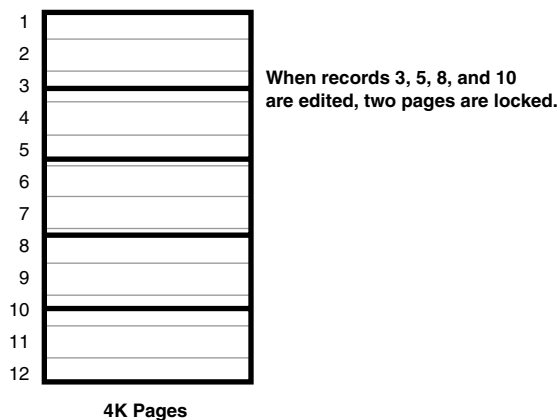
Looking at the Built-In Locking Modes

Access provides three alternative locking modes that you can use on forms, reports, and queries: No Locks, All Records, and Edited Record. The following table shows the objects and which modes work with them. You can set these choices globally on the Options dialog’s Advanced page, as explained earlier in the section “Default Record Locking.”

<i>Object Type</i>	<i>Locking Modes Used</i>
Form	No Locks
	All Records
	Edited Record
Report	No Locks
	All Records
Query	No Locks
	All Records
	Edited Record

Understanding Row-Level Versus Page-Level Locking

Before Access 2000 (Jet 4.0), Jet didn’t use row-level locking. Instead, it used what’s referred to as *page-level locking*. This means that depending on the locking scheme used, when a record is edited, a 4KB page surrounding the record is locked as well. Figure 21.10 shows what a 4KB page would look like. As of Access 2000, you can use either page-level or record-level locking.

**FIGURE 21.10**

The numbers shown here are merely for diagram purposes; Access doesn't use record numbers.

As you can see in Figure 21.10, some issues come along with page locking. For example, when a record falls on more than one page, both pages are locked. Issues are more pronounced when using some locking schemes rather than others.

NOTE

With Jet 4.0, the page size increased to 4KB from 2KB.

All this discussion of page locking deals with Access native tables only. Dealing with other products for the back end depends on the product. Other ISAM products, such as Microsoft FoxPro and Paradox, use their own locking methods. SQL Server lets Access perform all necessary locking.

Using row-level locking (particularly Edited Record mode) not only is more convenient, but also increases performance.

Using the Built-In Locking Modes

Most developers work with the three built-in Access locking modes when dealing with multi-user programming. That Access has no problem adding records and locking out people also helps quite a bit. For some developers, however, it's just not quite good enough—they choose to use their own ways to handle record locking, discussed later in the section “Using Alternative Locking Schemes.”

Locking All Records Mode

When using the All Records locking mode, you are essentially locking all records in a recordset. This means that other users who try to access the same records can only read those records, not update them. Using All Records means that other users are also excluded from adding or deleting records.

CAUTION

All Records mode is pretty extreme and should be used with care. A multiuser system isn't very "multiuser" friendly if only one person can update records in a whole table or recordset at a time.

In some cases, you want to use this mode. For example, if you're creating a long report, you want to make sure that the data doesn't change while you're working on it, so you set the report's Record Locks mode to All Records.

Locking Edited Records Mode

Edited Record mode uses *pessimistic* locking, which locks the row or page as soon as you start editing a record. This mode has its advantages and disadvantages.

One advantage is that you can ensure that only one user at a time is editing a record. As soon as a user presses a key, that page is locked, and any other users who happen to be on that page receive an international "not" symbol on their record selector. Figure 21.11 shows a locked page.



FIGURE 21.11

In Edited Record locking mode, only one user at a time can edit records in the same row or page.

You can tell in Figure 21.11 that the Customers form (bottom) is being edited because of the pencil symbol on the record selector along the left side. You know the `tblCustomers` table (top) can't be edited because of the "not" symbol on the left side.

Now, the bad news for page-level locking: Notice that although the record for Maria Anders is next to the one being edited in the Customer form, it's the one with the "not" symbol. This happens when you have Access use page locking rather than record locking. Because Access locks pages of 4KB, a number of records can be locked, depending on their size. This can confuse users, and more than once you'll get a call saying that Access is locking records other than the one being edited.

No Locks Mode

The No Locks mode is referred to as *optimistic* locking because the lock is in place only at the time the update is performed, and it causes an error only if the same record is edited by two different people at the same time. This kind of error—a *write conflict*—is one of the biggest disadvantages to using optimistic locking. A write conflict can occur as follows:

1. A person starts editing a record.
2. Another person starts to edit the same record.
3. The first person tries to save the record, thus causing a write conflict (see Figure 21.12).



FIGURE 21.12

Write conflicts can be somewhat confusing for users to deal with.

The choices on the Write Conflict dialog are as follows:

- *Save Record* saves the record over whatever changes were made by the other user.
- *Copy to Clipboard* allows the user to copy the information for the record to the Clipboard and then to append it to the end of the recordset. This choice is highly confusing and not one that users should be encouraged to use.
- *Drop Changes* allows users to simply forgo the changes altogether.

Although the write-conflict error is confusing, the No Locks mode results in the best performance because a conflict occurs only when two users are working on the same record.

Using Locking Modes in VBA

In VBA, the default mode of record locking is pessimistic. To set it to optimistic by using DAO, you must set the `LockEdits` property of the recordset to false. This sample code does just this:

```
Set dynSample = dbLocal.OpenRecordSet("Customers", dbOpenDynaset)
dynSample.LockEdits = False
```

For ADO, use the recordset's `LockType` property, which has four settings: `adLockReadOnly` (the default), `adLockPessimistic`, `adLockOptimistic`, and `adLockBatchOptimistic`. The first three settings correspond with All Locks, Edited Records, and No Locks, respectively. The fourth VBA setting, used in the following code snippet, handles batched records:

```
rstCust.Open "Customers", cnnCurr
rstCust.LockType = adLockOptimistic
```

Using Alternative Locking Schemes

A couple of different schemes for working with Access locking mechanisms include rolling your own record locking and using unbound forms to control locking through code.

Using the “Roll-Your-Own” Scheme

The idea behind rolling your own record locking is to create a record-level locking scheme for your application. This can be done with tables to store the record that's being edited at any given time. Then some code needs to maintain the locks and keep everything updated.

This scheme isn't recommended and is unnecessary with Access 2002, because it can take quite a bit of code to maintain all the necessary pieces. I mention it only for those of you who like to take care of record locking yourselves.

It's also not recommended because other applications might get into the system through VBA and Automation or ODBC. These applications won't necessarily be up to the task of keeping the record locks in place. For example, if a program is using Visual Basic to look at the data, it has to know exactly what you did—and have the routines—to perform the same type of locking scheme.

Using the Unbound Forms Scheme

The unbound forms scheme is popular. You have to do some work up front for one form, and then—if done properly—the other forms you create can use the same functions created for the first one. This scheme is good because you can have very fast forms that maintain their speed, even when used on a WAN with systems all over the country.

NOTE

Forms bound to a record source can greatly decrease editing and appending performance when a large data set is used over a large network. Again, wise programming can overcome most hindrances before having to turn to client/server solutions.

The major benefits of using this scheme are that the locks are all performed within code at the time the updates are performed, and can be controlled with error handling. The tough part of using this scheme is that a bit of code is involved in getting the routines set up to handle all the work that's performed automatically when you use bound forms.

The following steps overview what needs to take place when you create unbound forms (the next section presents the actual details):

1. Create your forms just as you would regular bound forms, with the `RecordSource` property set to whatever source you normally use, such as `Customers`, and with bound fields.
2. Remove the control sources from the fields for the form, and leave the name reflecting the field from the original record source, but with the fields unbound.
3. Remove the record source from the form.
4. Set up command buttons to add new records, open existing records with lookup, save records, and perform any other tasks you need. These commands look up the information in the original recordset, but you control it.

Step 4 is by far the most complicated. It also turns off most developers before even starting. The cool part is that step 4—the writing of the VBA code—has been done for you generically, and can then be used on other forms with very little coding of your own. All you need to do is remember which pieces go where and what's already created. The following sections discuss this as well as other coding issues.

Working in VBA with Unbound Forms

Potentially, additional coding needs to be done for multiuser programming. Again, it depends on how robust and powerful you want your application to be. What's meant by *powerful* in this case is the following:

- How fast do you want the application to be?
- How easy to use do you want it to be for users?
- How complete do you want to make the error handling for multiuser situations?

CAUTION

This discussion isn't for the faint-hearted when it comes to code. Although these routines are somewhat complicated, most of them need to be created only once and then called from forms thereafter. Remember that this code is a great place to start; however, not every scenario that arises when using unbound forms can be covered in this single discussion.

Creating the Routines for Handling Unbound Forms

The big part of coming up with useful routines is to make them as generic as possible so that you can use them repeatedly. Most of the code discussed in this chapter has been placed in a global module so that it can be created once and reused. You'll also want to simply copy and paste the form for the next one you want to create because the routines, as well as the command buttons needed, are the same. Figure 21.13 shows the sample form with the added command buttons.

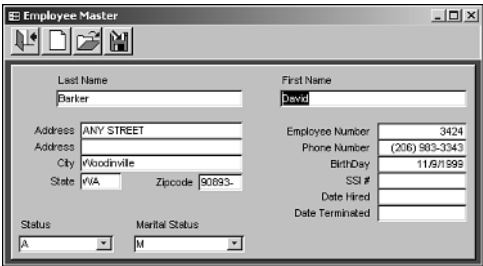


FIGURE 21.13

The Employee Master form in Chap21(ADO).mdb is a good example of unbound form techniques.

Before you get started, it would be good to lay out all the objects that will be used (see Table 21.1).

TABLE 21.1 Objects Used in the ADO Unbound Form Example

Object Name	Type	Location
tblEmployee	Table	Chap21BE(ADO).mdb
qryEmployee	Query	Chap21BE(ADO).mdb
qryEmployeeValueLocate	Query	Chap21BE(ADO).mdb

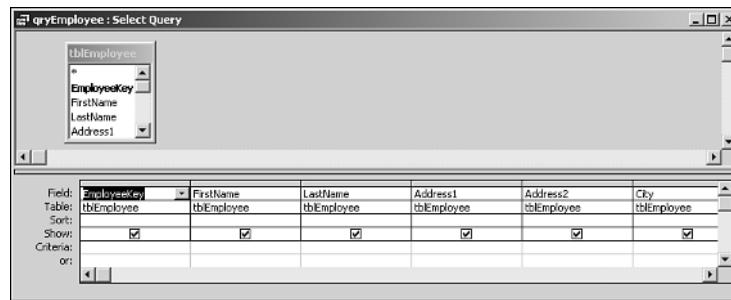
TABLE 21.1 Continued

<i>Object Name</i>	<i>Type</i>	<i>Location</i>
frmEmployee(ADO)	Form	Chap21(ADO).mdb
modCommonRoutines	Module	Chap21(ADO).mdb
modUnboundForms(ADO)	Module	Chap21(ADO).mdb

Using the Sample Form, Step by Step

The following steps are intermixed with the pieces from the part of the example that sets up a form to use as an unbound form. At the end of this section, the steps for using the completed pieces are laid out as a checklist when dealing with unbound forms.

1. Create the form by first binding it to a recordset. In this case, the recordset is named qryEmployee and is assigned as the form's record source. Figure 21.14 shows the query in Design view.

**FIGURE 21.14**

Each field is laid out in the qryEmployee query.

TIP

Laying out each field individually in the query comes in handy later when the various routines compare the form's controls to the query's fields.

2. Place each field on the frmEmployee(ADO) form from the field list, just as you would create any other form.
3. Create the command buttons necessary to control adding, editing, and saving information. (The code for this information is discussed shortly.)

4. After the form is set up with the field information, unbind it by deleting the `ControlSource` properties on the controls, and then deleting the form's `RecordSource` property. Figure 21.15 shows the unbound `frmEmployee(ADO)` form and the property sheet for the `LastName` field.

NOTE

In addition to removing the `ControlSource` on each control, be sure to turn off the record selectors and navigation buttons to control how users get to the various records.

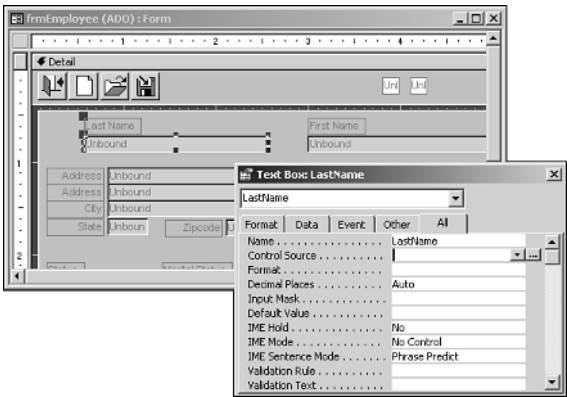


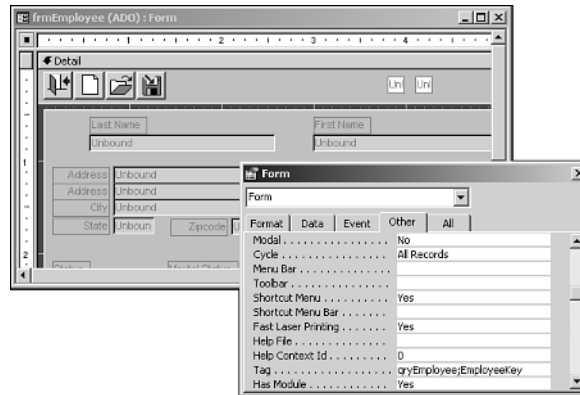
FIGURE 21.15

Although the controls' `ControlSource` properties are cleared, the `Name` properties remain.

NOTE

The `qryEmployee` query can now be put onto the back end by exporting and then deleting it from the front end. It will be used later in the example by opening it through ADO. Most of the time, you will do this when you use unbound forms, such as with client/server.

5. Place the name of the `RecordSource` with the name of the key field (usually the Primary Key field) for the form into the `Tag` property of the `frmEmployee(ADO)` form (see Figure 21.16).

**FIGURE 21.16**

Notice the RecordSource and key field placed in the Tag property of the Employee form.

Now that the form itself is created and the objects are defined, look at the code attached to the command buttons.

Using Support Routines

Before going over the specific tasks, look at a couple of support routines that help retrieve the RecordSource and key field name from the form's Tag property. These two routines and most of the rest of the functions are located in the `modCommonRoutines` standard module.

The first function, `ap_GetRecordSource()`, returns the RecordSource name (see Listing 21.1). The current form—in this case, `frmEmployee(ADO)`—is passed to both functions.

LISTING 21.1 Chap21(ADO).mdb: Getting the Form's Record Source

```
Function ap_GetRecordSource(frmCurr As Form) As Variant
    On Error GoTo Error_ap_GetRecordSource

    '-- Returns the name of the record source stored in the
    '   Tag property of the form
    If InStr(frmCurr.Tag, ";") <> 0 Then
        ap_GetRecordSource = Left$(frmCurr.Tag, InStr(frmCurr.Tag, ";") - 1)
    Else
        ap_GetRecordSource = frmCurr.Tag
    End If

    Exit Function
Error_ap_GetRecordSource:
    Exit Function
```

LISTING 21.1 Continued

```
Error_ap_GetRecordSource:
    MsgBox Err.Description
    Exit Function

End Function
```

The next function, `ap_GetKey()`, retrieves the key field name (see Listing 21.2).

LISTING 21.2 Chap21.mdb: Getting the Key Field from the Form's Tag Property

```
Function ap_GetKey(frmCurr As Form) As Variant
    On Error Resume Next
    '-- Returns the Key Field stored in the Tag property of the form
    ap_GetKey = Mid$(frmCurr.Tag, InStr(frmCurr.Tag, ";") + 1)
End Function
```

The first task to look at is how to add a new record.

Adding a Record on the Unbound Form

To add a new record, the code in the `cmdNew_Click` event procedure in Listing 21.3 first offers to save the current record (if required), and then clears the form by using the `ap_ADONewRecord()` function (see Listing 21.4).

LISTING 21.3 Chap21(ADO).mdb: Checking Whether to Save the Current Record, and Then Clear Fields for a New Record

```
Private Sub cmdNew_Click()
    Dim flgFormSaved As Boolean

    flgFormSaved = True

    If pflgFormEdited Then
        flgFormSaved = ap_ADOSaveRecord(Me, True)
    End If

    If flgFormSaved Then
        ap_ADONewRecord [Form], False
    End If

End Sub
```

LISTING 21.4 Chap21(ADO).mdb: Clearing the Form to Create a New Record

```

Function ap_ADONewRecord(frmCurr As Form, flgDisable As Boolean)
    Dim strConnect As String
    Dim cnnBE As New ADODB.Connection
    Dim rstCurr As New ADODB.Recordset
    Dim ctlCurrent As Control
    Dim strFirstControl As String

    On Error GoTo Error_ap_ADONewRecord

    '-- Open the ADO connection
    strConnect = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source="
    strConnect = strConnect & CurrentProject.Path & "\Chap21BE(ADO).mdb;"
    cnnBE.Open strConnect

    rstCurr.Open "Select * From " & ap_GetRecordSource(frmCurr) & _
        " Where False;", cnnBE, adOpenStatic

    '-- This flag is used at the end to place the cursor at the first control
    strFirstControl = ""

    '-- For each of the controls on the form
    For Each ctlCurrent In frmCurr
        '-- If an error occurs, handle it
        On Error Resume Next

        '-- Attempts to match a field in the recordset up with the
        '-- current control from the form.
        pvarDummy = rstCurr.Fields(ctlCurrent.Name).Name

        '-- If no error, that means the field exists
        If Err = 0 Then
            '-- If the current control is first in the index, record
            If ctlCurrent.TabIndex = 0 Then
                strFirstControl = ctlCurrent.Name
            End If
            On Error GoTo Error_ap_ADONewRecord

            '-- If the argument flgDisable is set to true, then after setting the
            '-- fields, disable the field unless it is a Lookup field as assigned
            '-- to the Tag property of the control.

            If flgDisable Then
                If IsNull(ctlCurrent.Tag) Or ctlCurrent.Tag <> "LookUp" Then
                    ctlCurrent.Enabled = False
                End If
            End If
        End If
    Next
End Function

```

LISTING 21.4 Continued

```

    ElseIf ctlCurrent.Tag = "LookUp" Then
        ctlCurrent.Enabled = True
    End If
    ctlCurrent.Value = Null
Else '-- Otherwise enable it unless told to DisplayOnly in the
    '-- Tag property of the control.
    If ctlCurrent.Tag <> "DisplayOnly" Then
        ctlCurrent.Enabled = True
    ElseIf IsNull(ctlCurrent.Tag) Then
        ctlCurrent.Enabled = True
    End If
    '-- Handle the default value assigned to the control.
    If IsNull(ctlCurrent.DefaultValue) Then
        ctlCurrent.Value = Null
    Else
        '-- Default value is a function.
        If Left$(ctlCurrent.DefaultValue, 1) = "=" Then
            ctlCurrent.Value = Eval(Mid$(ctlCurrent.DefaultValue, 2))
        Else
            '-- Default value is a string.
            If InStr(ctlCurrent.DefaultValue, "\"") <> 0 Then
                ctlCurrent.Value = Eval(ctlCurrent.DefaultValue)
            Else '-- Default value any other type'
                ctlCurrent.Value = ctlCurrent.DefaultValue
            End If
        End If
    End If
End If
End If
End If
Err = 0
Next

'-- Null out the key value for the form.
frmCurr(ap_GetKey(frmCurr)) = Null

'-- Set the flag for adding a new record on the form.
frmCurr!AddRec = -1

'-- Set the flag for allowing to look up based on the disabled flag.
'-- True means opening current record, False means new record
pflgPerformLookup = flgDisable

'-- Go to the first control in the Tab Index.
If Len(strFirstControl) > 0 And Not flgDisable Then

```

21

PART IV**LISTING 21.4** Continued

```
        frmCurr(strFirstControl).SetFocus
    End If

    Exit Function

Error_ap_ADONewRecord:
    MsgBox Err.Description
    Exit Function

End Function
```

In the first section of Listing 21.4, the Connect and Recordset objects are declared and assigned:

```
Dim strConnect As String
Dim cnnBE As New ADODB.Connection
Dim rstCurr As New ADODB.Recordset
Dim ctlCurrent As Control
Dim strFirstControl As String

On Error GoTo Error_ap_ADONewRecord

'-- Open the ADO connection
strConnect = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source="
strConnect = strConnect & CurrentProject.Path & "\Chap21BE(ADO).mdb;"
cnnBE.Open strConnect

rstCurr.Open "Select * From " & ap_GetRecordSource(frmCurr) & _
    " Where False;", cnnBE, adOpenStatic
```

TIP

Unlike DAO, in ADO there's no real clean way to examine a recordset's fields without opening it. In DAO, the QueryDef object would be opened and the Fields collection examined. In ADO, one trick—as performed here—is to open the recordset as false. That way, the engine returns no records fairly quickly and you can crack it open to see the Fields collection.

The main task of the ap_ADONewRecord() function is to go through each control on the form, as performed with the following code lines:

```
'-- For each of the controls on the form
For Each ctlCurrent In frmCurr
```

The next task is to check whether the current control name can be found in the form's recordset:

```
'-- If an error occurs, handle it
On Error Resume Next

'-- Attempts to match a field in the recordset up with the
'-- current control from the form.
pvarDummy = qdfCurr.Fields(ctlCurr.Name).Name

'-- If no error, that means the field exists
If Err = 0 Then
```

As you can see from Listing 21.4, the idea is to be as generic as possible. This function also handles the following details:

- If the `flgDisable` flag is true and the control's `Tag` property is set to `LookUp`, the field is enabled for searching for a record. (This is covered in greater detail in the next section.)
- The opposite is true if `flgDisable` is false and the `Tag` property is set to `DisplayOnly`. This handles those controls that you want to use only to display the information.
- Various `DefaultValue` possibilities are also covered, whether it's a function (`=Date()`), string (WA), or value (5).
- This function sets the focus on the first control in the tab index order.

Locating and Loading a Record with the Unbound Form

The next functions, starting with `ap_AD00openWithChoice()`, locate and load an existing record. Figure 21.17 shows the call for `ap_AD00openWithChoice()` attached to the `cmdOpen` command button's `OnClick` event; Listing 21.5 shows the function's code.

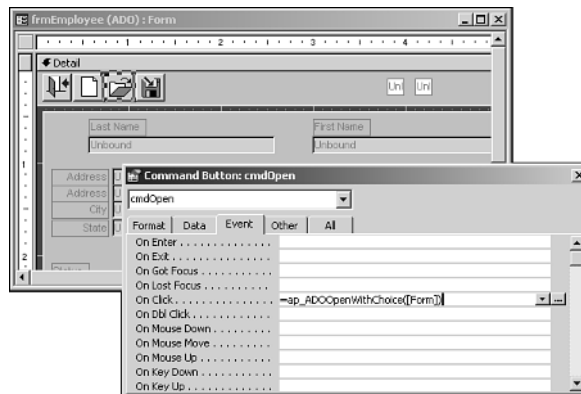


FIGURE 21.17

`ap_AD00openWithChoice()` is in the `modUnboundForms(ADO)` module.

LISTING 21.5 Chap21(ADO).mdb: Giving Users a Choice of Lookup Fields

```

Function ap_AD00openWithChoice(frmCurr As Form)
    Dim ctlCurrent As Control
    Dim flgFormSaved As Boolean

    On Error GoTo Error_ap_AD00openWithChoice

    '-- If the form has been edited, call the routine to save it.
    flgFormSaved = True

    If pflgFormEdited Then
        flgFormSaved = ap_AD0SaveRecord(frmCurr, True)
    End If

    '-- If the save was successfull, clear the controls on the form
    '-- then set the focus to the first lookup field.
    If flgFormSaved Then
        pflgPerformLookup = True
        pvarDummy = ap_AD0NewRecord(frmCurr, True)
        For Each ctlCurrent In frmCurr
            If ctlCurrent.Tag = "LookUp" Then
                ctlCurrent.SetFocus
            Exit For
        End If
    Next
    End If

    Exit Function

Error_ap_AD00openWithChoice:
    MsgBox Err.Description
    Exit Function

End Function

```

This function calls the save routine, if applicable. Then, if told to continue, it calls `ap_AD0NewRecord` to clear the values of the form's controls. Finally, it locates the first field with the text `LookUp` in the `Tag` property and sets the focus to that field (see Figure 21.18).

Next, look at the function called when the lookup field is updated with a value to locate. This function, `ap_AD0LookUpAfterUpdate()`, is located in the `modUnboundForms(ADO)` module (see Listing 21.6). Notice that two ADO object models are used: `ADODB` and `ADOX`.

FIGURE 21.18

The `LastName` field, because its `Tag` property is set to `LookUp`, has informed `ap_ADOOpenWithChoice()` to use it.

LISTING 21.6 Chap21(ADO).mdb: Opening and Executing a Query

```
Sub ap_ADOValueStandardSearch(frmCurr As Form)
    Dim strConnect As String
    Dim cnnBE As New ADODB.Connection
    Dim catCurr As New ADOX.Catalog
    Dim cmdCurr As New ADODB.Command
    Dim rstSearch As New ADODB.Recordset
    Dim strCriteria As String

    On Error GoTo Error_ap_ADOValueStandardSearch

    DoCmd.Hourglass True

    '-- Open the ADO connection
    strConnect = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source="
    strConnect = strConnect & CurrentProject.Path & "\Chap21BE(ADO).mdb;"
    cnnBE.Open strConnect

    catCurr.ActiveConnection = cnnBE

    Set cmdCurr = catCurr.Procedures(ap_GetRecordSource(frmCurr) & _
        "ValueLocate").Command

    cmdCurr.Parameters(Screen.ActiveControl.Name & "ToLocate") = _
        Screen.ActiveControl.Value

    '-- Open the recordset
    rstSearch.Open cmdCurr, , adOpenForwardOnly, adLockReadOnly, adCmdStoredProc

    DoCmd.Hourglass False
```

LISTING 21.6 Continued

```

If rstSearch.EOF Then
    MsgBox "This " & Screen.ActiveControl.Name & " was not found!", _
        vbCritical, "Search Error"
Else
    '-- If found, load up the controls
    ap_ADOLoadRecord rstSearch, frmCurr
End If

Exit Sub

Error_ap_ADOValueStandardSearch:
    DoCmd.Hourglass False
    MsgBox Err.Description
    Exit Sub

End Sub

```

One potentially confusing piece of Listing 21.6 is the following lines:

```

Set cmdCurr = catCurr.Procedures(ap_GetRecordSource(frmCurr) & _
    "ValueLocate").Command

```

These code lines combine the record source for the form, retrieved by the `ap_GetRecordSource()` function, with the literal `ValueLocate`. A command variable, `cmdCurr`, is set to reference the `qryEmployeeValueLocate` saved query.

NOTE

The `qryEmployeeValueLocate` query was created by taking the `qryEmployee` query and copying it, adding `ValueLocate` to the original name, and then adding the parameter mentioned in the next paragraph.

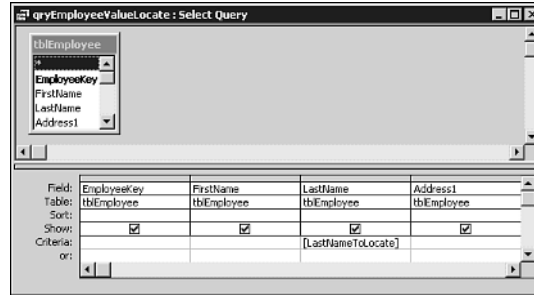
The next step in Listing 21.6 is to set the query's parameter by combining the name of the active control with the literal `ToLocate`. In this case, the parameter `LastNameToLocate` is set to the `LastName` control value:

```

cmdCurr.Parameters(Screen.ActiveControl.Name & "ToLocate") = _
    Screen.ActiveControl.Value

```

Figure 21.19 shows the `qryEmployeeValueLocate` query, located in `Chap21BE(ADO).mdb`.

**FIGURE 21.19**

Notice the `LastNameToLocate` parameter in the `qryEmployeeValueLocate` query.

If a record is found for the name, the values are loaded into the form's control. The lines that call the routine are as follows:

```
'-- If found, load up the controls
ap_ADOLoadRecord rstSearch, frmCurr
```

The recordset containing the record found and the current form are passed to `ap_ADOLoadRecord` as arguments. Listing 21.7 shows the code for `ap_ADOLoadRecord`.

LISTING 21.7 Chap21(ADO).mdb: Loading Existing Information

```
Sub ap_ADOLoadRecord(rstCurrent As ADODB.Recordset, frmCurr As Form)
    Dim ctlCurrent As Control

    On Error GoTo Error_ap_ADOLoadRecord

    '-- Turn the lookup feature off when a record is first loaded
    pflgPerformLookup = False

    '-- For all the controls on the form
    For Each ctlCurrent In frmCurr
        '-- If an error occurs, handle it
        On Error Resume Next

        '-- Since either data is now present, or new record allow editing
        ctlCurrent.Enabled = True

        '-- Attempts to match a field in the recordset up with the
        '-- current control from the form.
        pvarDummy = rstCurrent.Fields(ctlCurrent.Name).Name
```

LISTING 21.7 Continued

```

'-- If no error, that means the field exists
If Err = 0 Then
    On Error GoTo Error_ap_ADOLoadRecord
    '-- Store the recordset value into the form control
    ctlCurrent.Value = rstCurrent(ctlCurrent.Name).Value
End If
Err = 0
Next

'-- Since a record has been loaded, we are not adding a record.
frmCurr!AddRec = 0

Exit Sub

Error_ap_ADOLoadRecord:
MsgBox Err.Description
Exit Sub

End Sub

```

When the record loads, you see the information that was in the record, as in Figure 21.20.

The screenshot shows a Windows-style application window titled "Employee Master". It contains a form with two columns of data. The left column contains text boxes for "Last Name" (Sheeley), "Address" (123 ANYSTREET), "City" (Woodinville), "State" (WA), and "Zipcode" (90494). The right column contains text boxes for "First Name" (Steve), "Employee Number" (7887), "Phone Number" ((206) 887-2883), "BirthDay" (1/23/1960), "SSI #" (233-22-3322), "Date Hired" (12/12/1993), and "Date Terminated". At the bottom, there are two dropdown menus for "Status" (set to 'A') and "Marital Status" (set to 'M').

FIGURE 21.20

The information is loaded and ready for editing.

The code in Listing 21.7 is pretty straightforward in that it just whips through each control on the form, as in the other routines discussed already.

Saving a Record on the Unbound Form

The last task to look at when working with unbound forms is saving an updated record. The function that performs this is `ap_ADOSaveRecord()`. The code is shown in its entirety in Listing 21.8 and then broken up to explain individual sections.

LISTING 21.8 Chap21(ADO).mdb: Saving a Record

```

Function ap_ADOSaveRecord(frmCurr As Form, flgAskSave As Boolean) As Integer
    Dim ctlCurrent As Control
    Dim flgPerformSave As Integer
    Dim flgCheckField As Integer
    Dim strKeyName As String

    On Error GoTo Error_ap_ADOSaveRecord

    '-- If necessary to request whether the user wants to save the
    '   information, do so.
    If flgAskSave Then
        Beep
        flgPerformSave = _
            MsgBox("Would you like to save the information on '" & _
                frmCurr.Caption & "'?", 36, "Save Information?") = vbYes
    Else
        flgPerformSave = True
    End If

    If flgPerformSave Then
        Dim strConnect As String
        Dim cnnBE As New ADODB.Connection
        Dim rstCurr As New ADODB.Recordset
        Dim strCriteria As String

        '-- Open the ADO connection
        strConnect = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source="
        strConnect = strConnect & CurrentProject.Path & "\Chap21BE(ADO).mdb;"
        cnnBE.Open strConnect

        '-- Open the ADO recordset.
        rstCurr.Open ap_GetRecordSource(frmCurr), cnnBE, adOpenKeyset, _
            adLockOptimistic, adCmdTable

        On Error Resume Next

        '-- Get the name of the key field
        strKeyName = ap_GetKey(frmCurr)

        '-- If adding a record, perform the addnew command.
        If frmCurr!AddRec = -1 Then
            rstCurr.AddNew
        Else

```

21

LISTING 21.8 Continued

```

'-- If editing, find the record to update
strCriteria = "[" & strKeyName & "]" = " & frmCurr(strKeyName)
rstCurr.Find strCriteria

If rstCurr.EOF Then
    Beep
    MsgBox "An error has occurred locating the current record!", _
        vbCritical, "Locating Error"
    Exit Function
End If

For Each ctlCurrent In frmCurr
    pvarDummy = rstCurr(ctlCurrent.Name).Name
    If Err = 0 Then
        flgCheckField = True
        '-- If the current control is a keyfield and a counter,
        '    don't mess with it.
        If ctlCurrent.Name = strKeyName Then
            If rstCurr.Fields(ctlCurrent.Name).Properties("AutoIncrement") _
                = True Then
                flgCheckField = False
            End If
        End If
    End If

    If flgCheckField Then
        '-- If data has changed, then update it.
        If Nz(rstCurr(ctlCurrent.Name).Value) <> Nz(ctlCurrent.Value) Then
            rstCurr(ctlCurrent.Name).Value = ctlCurrent.Value
        End If
    End If

    Err = 0

Next

On Error GoTo Error_ap_ADOSaveRecord

'-- Grab the keyfield value from the recordset
frmCurr(strKeyName) = rstCurr(strKeyName)

```

LISTING 21.8 Continued

```

    '-- Update the recordset
    rstCurr.Update

    '-- Close the recordset and commit the transaction
    rstCurr.Close

    ap_ADOSaveRecord = True

Else
    ap_ADOSaveRecord = True
End If

'-- set the flag that the form is edited to false.
pflgFormEdited = False
frmCurr!AddRec = 0

Exit Function

Error_ap_ADOSaveRecord:
    MsgBox Err.Description
    rstCurr.CancelUpdate

    Resume 'Exit Function

End Function

```

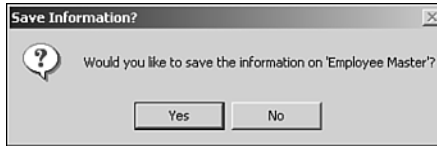
This function takes the following major actions:

1. If the `flgAskSave` argument is set to yes, users are prompted to save the information (see Figure 21.21). The code lines that perform this are as follows:

```

'-- If necessary to request whether the user wants to save
'   the information, do so.
If flgAskSave Then
    Beep
    flgPerformSave = _
        MsgBox("Would you like to save the information on '" & _
            frmCurr.Caption & "'", 36, "Save Information?") = vbYes
Else
    flgPerformSave = True
End If

```


**FIGURE 21.21**

The function prompts users about saving changes.

2. The function checks to see whether a new record is being added or a current record is being updated, and acts accordingly. (Note that with editing a record with ADO, you don't actually have to use an Edit method, as with DAO.)

```
'-- If adding a record, perform the addnew command.
If frmCurr!AddRec = -1 Then
    rstCurr.AddNew
Else
    '-- If editing, find the record to update
    strCriteria = "[" & strKeyName & "] = " & frmCurr(strKeyName)
    rstCurr.Find strCriteria

    If rstCurr.BOF Then
        Beep
        MsgBox "An error has occurred locating the current record!", _
            vbCritical, "Locating Error"
        Exit Function
    End If
End If
```

3. The last main section whips through all the controls on the form. If the control is in the recordset, the function checks to see whether it's an AutoIncrement field. If it is, the routine skips the control; otherwise, it checks to see whether the value has changed and handles it accordingly:

```
'-- If the current control is a keyfield and a counter,
'    don't mess with it.
If ctlCurrent.Name = strKeyName Then
    If rstCurr.Fields(ctlCurrent.Name).Properties("AutoIncrement") _
        = True Then
        flgCheckField = False
    End If
End If
```

```

If flgCheckField Then
    '-- If data has changed, then update it.
    If Nz(rstCurr(ctlCurrent.Name).Value) <> Nz(ctlCurrent.Value) Then
        rstCurr(ctlCurrent.Name).Value = ctlCurrent.Value
    End If
End If

```

Again, after these routines are created, you only have to cut and paste to new forms. The following steps do just that:

1. Copy the frmEmployee(ADO) form to your new form.
2. Delete all the controls that refer to fields from the new form, leaving the controls on the top of the form. Leave the AddRec and EmployeeKey fields in place for now, however.
3. Place the name of the record source in the form's RecordSource property.
4. Place the fields from the field list where you want them on the form.
5. After the fields are set up the way you want them, delete each field's control source.
6. Highlight all fields that aren't lookup and display only. Then type `=ap_FormEdited()` in the AfterUpdate event of the multiple selection.
7. Highlight all fields and set the Enable property to false.
8. In the field on which you want to perform a lookup, set the Tag property to Lookup. Then place `=ap_ADOLookupAfterUpdate([Form])` in the field's AfterUpdate event.
9. Create the RecordSource and Locate queries, as described earlier in the section "Locating and Loading a Record with the Unbound Form."
10. Place the RecordSource name, along with the key field name, in the new form's Tag property.
11. Find the EmployeeKey field and change it to the key value for this form. It will be the unique value, such as an AutoIncrement field.

That's all there is to it. After you do it a couple of times, it should take about only five minutes to create new forms.

Coding for Multiuser Error Handling

The main issue to consider with multiuser error handling is that when an error can be retried, such as a locking error, give users a chance to retry it. This can be seen in the centralized error handler created in the VideoApp(ADO).mdb database, located on the book's Web page at www.sampublishing.com.

NOTE

The error handler itself, found in the `ErrorHandling` module, is discussed in Chapter 7, “Handling Your Errors in Access with VBA.”

Listing 21.9 shows the portion of the `ap_ErrorHandler` function that you’re interested in.

LISTING 21.9 VideoApp(ADO).mdb: Trapping Multiuser Errors

```
'-- Check to see how this error is to be handled
Select Case inhHowTohandle
Case apExitApplication
    '-- if quit the application, give a message saying so, then quit.
    strErrorMsg = strErrorMsg & ap_CRLF() & ap_CRLF() & _
        "The application will be closed"
    Beep
    MsgBox strErrorMsg, vbCritical, "Critical Error"
    Application.Quit
Case apExitRoutine
    '-- If exiting the routine, give a message saying the task has been stopped.
    strErrorMsg = strErrorMsg & vbCrLf & vbCrLf & _
        "Some tasks may have not been performed"
    Beep
    MsgBox strErrorMsg, vbCritical, "Task Error"
Case apMultiUser
    '-- For multi-user errors try a couple of times, then check with user.
    If ap_ErrorTryMUAgain(strOrigErrorMsg) Then
        inhHowTohandle = apTryAgain    '-- If user wants, try again
    Else
        inhHowTohandle = apExitRoutine '-- If struck out, exit the routine
    End If
End Select
```

The last `Case` statement calls multiuser handling. The `inhHowToHandle` flag is taken from a table where the error was looked up with the error categorized. The `ErrorInfo` table contains the errors list.

When the error is flagged as a type 5 (multiuser), the `ap_ErrorTryMUAgain` function is called with the error string passed to it. Listing 21.10 shows the code for this function.

LISTING 21.10 VideoApp(ADO).mdb: Attempting to Lock a Record Again

```

Function ap_ErrorTryMUAgain(strErrorMessage) As Integer
    Static intNumRetries As Integer
    Dim strStatusMsg As String

    On Error GoTo Error_ap_ErrorTryMUAgain

    intNumRetries = intNumRetries + 1
    ap_ErrorTryMUAgain = True

    If intNumRetries >= apMaxMURetries Then
        DoCmd.Echo True
        intNumRetries = 0
        If MsgBox(strErrorMessage & vbCrLf & vbCrLf & _
            "Do you want to try again?", 20, "Multi-User Error") <> 6 Then
            ap_ErrorTryMUAgain = False
            Exit Function
        End If
    Else
        dbEngine.Idle
    End If

    If InStr(strErrorMessage, "user") > 0 Then
        strStatusMsg = ap_ErrorCreateStatusMessage(strErrorMessage, _
            Str(intNumRetries + 1) & " of " & apMaxMURetries)
    Else
        strStatusMsg = "Locked, Retrying..." & Str(intNumRetries + 1) _
            & " of " & apMaxMURetries
    End If

    DoCmd.Echo True, strStatusMsg

    Exit Function

Error_ap_ErrorTryMUAgain:
    MsgBox Err.Description
    Exit Function

End Function

```

This function tracks how many times the system has retried the error, and then displays a message when the number of retries exceeds the number of `apMaxMURetries`. (This constant is in the `ErrorHandling` module's declaration section.) Also, the `dbEngine.Idle` command frees up the Jet engine to clear any pending locks.

PART IV

The following code displays a message on the status bar to let users know what's happening. The idea is to give users as much feedback as possible while trying to get the job done.

```
If InStr(strErrorMessage, "user") > 0 Then
    strStatusMsg = ap_ErrorCreateStatusMessage(strErrorMessage, _
        Str(intNumRetries + 1) & " of " & apMaxMURetries)
Else
    strStatusMsg = "Locked, Retrying..." & Str(intNumRetries + 1) & " of _
        " & apMaxMURetries
End If
```

NOTE

This code covers only those multiuser errors with the word user in them. You can display other multiuser errors as well—this is just a starting point.

Listing 21.11 shows the code for `ap_ErrorCreateStatusMessage`, which simply parses this particular error message so that you can display it with a counter on the status bar in a more meaningful way.

LISTING 21.11 VideoApp(ADO).mdb: Displaying the Error Message

```
Function ap_ErrorCreateStatusMessage(strErrorMessage, strRetriesMsg As _
    String) As String

    On Error GoTo CreateStatusMessage_Error

    Dim strUserName As String, strMachineName As String, strStatusMsg As String
    Dim intMachNameStart As Integer, intMachNameEnd As Integer
    Dim intAfterUser As Integer

    intAfterUser = InStr(strErrorMessage, "User") + 5

    strUserName = Mid$(strErrorMessage, intAfterUser, _
        InStr(intAfterUser, strErrorMessage, " ") - intAfterUser)

    intMachNameStart = InStr(intAfterUser, strErrorMessage, "machine")
    intMachNameEnd = InStr(intMachNameStart, strErrorMessage, ".") _
        - intMachNameStart
    strMachineName = Mid$(strErrorMessage, intMachNameStart + 8, intMachNameEnd)
    strStatusMsg = "Locked by: " & strUserName
    ap_ErrorCreateStatusMessage = strStatusMsg & " On machine: " & _
        strMachineName & " Retrying..." & strRetriesMsg
```

LISTING 21.11 Continued

```
Exit Function

CreateStatusMessage_Error:
    Beep
    MsgBox Error$, vbCritical, "Error"
    Exit Function

End Function
```

Summary

When you're dealing with a multiuser environment, it's worth the effort to come up with a robust way of handling issues. Access is great out of the box on a network, but to get performance enhancements and increased control—whether it's tweaking the option settings, splitting databases, or creating unbound forms—some work has to be done.

- Chapter 20, “Securing Your Application,” discusses options for securing your databases, including multiuser issues.
- Chapter 24, “Developing SQL Server Projects Using ADPs,” gives information on creating Access applications that work optimally with SQL Server.
- Chapter 25, “Startup Checking System Routines Using ADO,” has a whole section devoted to verifying the back-end database using ADO.
- Chapter 26, “Creating Maintenance Routines,” contains information and examples for replicating tables from back end to front end for performance.

Welcome to the World of Database Replication

CHAPTER

22

IN THIS CHAPTER

- Understanding Database Replication Concepts 692
- Working with Jet Replication Tools 694
- Converting Databases to Replicas 699
- Synchronizing Replicas 704
- Understanding Replica Set Topologies 710
- Distributing Replicable Applications 713
- Replicating Back-End and Front-End Applications 719
- Handling Replication Conflicts 719
- Understanding Replication Synchronizers 728
- Upgrading Replica Sets to Access 200x 739
- Securing Replicated Applications 739
- Using MDE Files with Replicated Databases 739
- Creating Successful Replication Applications 740

Database replication is the most exciting feature in desktop database technology since relational databases. Replication allows multiple copies of a database at remote, disconnected locations to simultaneously update and insert data. Later, these changes are propagated when the database copies are synchronized. Access allows data and objects such as forms, reports, Data Access Page links, modules, and macros to be replicated.

Access includes functionality such as the unified treatment of conflicts and errors, column-level tracking of changes (to decrease the occurrence of conflicts), an improved priority-conflict-resolution algorithm, added replica types and visibilities to control topology, and extended replication functionality exposed through the Microsoft *Jet and Replication Objects (JRO)* library, an ADO component. This chapter doesn't include information beyond replicating with SQL Server/ADPs.

NOTE

You can still use DAO with Access 2002, but DAO 3.6 doesn't expose any new replication features from the previous release of DAO because Access is moving toward a Universal Data Access solution. Such a solution isn't isolated to a single database type. For example, you can use ADO to access Jet or SQL Server data, depending on the provider defined in the connection.

In this chapter, you learn how you can design replicated databases in Access 2002, and when to use different design techniques. The basic concepts of database replication are introduced, and then more detail is provided on the areas you'll need to know for the successful development of replicated database applications. This chapter also covers Replication Manager, the user interface used to manage replication.

Understanding Database Replication Concepts

Access developers typically start with simple, single-user applications, and then progress to multiuser development. You're now ready to move up to the next level—distributed, multiuser, replicated database applications.

Access delivers replicated relational database functionality to the desktop, making feasible the development of economical and robust solutions to meet a whole new range of business applications. Replication allows users to have copies of the database at different locations; updates and insertions made in one replica propagate to all replicas.

The Replication Design Goal

Understanding the Access development team's intentions will help you develop successful replication applications. Some design aspects might initially appear obscure. Careful study of these features and tools, however, provides many opportunities to build powerful and efficient replicated database applications.

The primary design goals focused on providing a powerful set of replication features and tools for the typical Access user running a typical desktop PC configuration for Access 2002. Also, a PC environment is usually less controlled than, say, a centralized MIS operation, and many features were designed to protect less-experienced users from inadvertently getting into difficulties. The goals were as follows:

- **Ease of use.** Any Access user can create a replicated database application.
- **Minimum performance impact.** Try to keep performance as good for replicated databases as it is for non-replicated databases.
- **Transparency to end users.** A replicated database application should behave exactly the same as a non-replicated version of the same application. Data can be inserted, deleted, and updated in any replica.
- **Support of object and data replication.** Objects include forms, reports, Data Access Page links, macros, and modules.
- **Incremental changes.** Send only incremental changes at replica synchronization.
- **Ability to merge changes made in the same row.** If changes are made to different columns in the same row, you can merge changes without conflict through Access support of column-level tracking.
- **Full support for data convergence between replicas.** Errors are eliminated.
- **Control of design changes.** Allow design changes to only one replica in the replica set.
- **Low consistency between replicas.** For example, at any time two replicas can contain different data; however, the replicas become consistent with successful synchronization. With a design for low consistency, users don't need to maintain expensive and error-prone, full-time communication links.
- **Distributed administration.** This allows any full replica (but not partial replicas) to initiate a synchronization, create an additional replica, and resolve any resulting conflicts.
- **High latency between synchronizations.** For example, replicas won't try to synchronize immediately after a data update, but at some later time.
- **Support for "occasionally connected" users.** This includes connecting and updating laptop users.

Some Typical Replication Applications

Replicated databases offer radical solutions to a range of common business needs:

- **Distributed offices.** Businesses with more than one office need to share data. Due to cost, reliability, and performance, maintaining a constant communications link between offices is often impractical. Database replication allows each office to maintain a local copy of the database and synchronize regularly.
- **Mobile users.** The use of laptop computers has driven the need to take copies of data away from the office. By using database replication, a user can take a copy of the Customers Orders database to the client, construct proposals from current price lists, and enter new orders, which can then be sent to the head office at the end of the day.
- **Application distribution.** Replication applies not only to data but also to the forms, queries, Data Access Page links, reports, and so on that make up an application. Database application developers often need to distribute updates to existing applications. Replication can provide an extremely convenient method by synchronizing the customer's copy of the database to the developer's, thereby "downloading" the latest application updates onto the customer's PC.
- **Load balancing.** Replication is well suited to address two broad categories of load balancing. In a multiuser application, there might be too many users for a single server, in which case you could create a replica and distribute your users between the two replica copies.
- The other situation is one in which, for example, one user wants to conduct intensive detailed queries on your database. Queries can be very demanding on the database engine and can severely affect performance for all other users. In this situation, it would be simple to create an additional replica for the user who's running the queries, and then he can analyze to his heart's content without affecting his co-workers.
- **Backups.** Replication provides a simple backup solution. By scheduling exchanges between two replicas, you can ensure that a reserve copy of the database is always available if disaster befalls the other replica. Another significant benefit is that replication exchanges can occur without you having to take the database offline.

Working with Jet Replication Tools

In the following sections, you learn about the four major replication tools available. You also see a simple comparison demonstrating each tool's advantages. Some of the more esoteric features are discussed later in the chapter. For now, just contrast and compare the basic features provided by each replication tool.

The Briefcase

Windows includes the Briefcase as an easy-to-use tool that lets you work on files at home or on the road, and keep the various file copies synchronized (see Figure 22.1). Because an Access database is inherently a potential multiuser database, you must ensure that any changes you make while at home or on the road can be merged with changes made by your office colleagues. After you install Access, a database file placed into the Briefcase is automatically converted into a replicated database. Now simultaneous updates can occur at each copy and be merged later.

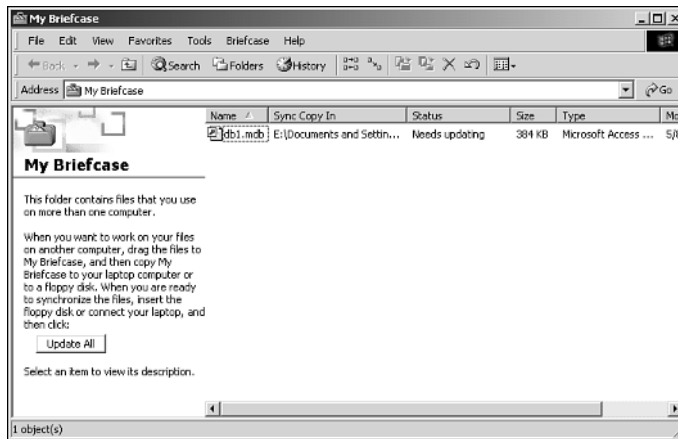


FIGURE 22.1

You can use the Windows Briefcase with a good number of applications.

The Briefcase is best suited to ad hoc replica usage. Imagine that you share a multiuser customer contacts database with your co-workers on a network. You want to take a copy of the database with you on a business trip, so you drag a copy of the database from Explorer into your laptop computer's Briefcase. You modify the data while away from the office, and on return merge your updates with the database replica on the network.

Let's go through an example of creating a database and using the Briefcase to replicate it, and then merge the replicas:

1. Ensure that the Briefcase is visible on your desktop. (If it isn't, right-click the desktop and choose New and then Briefcase.)
2. In Access, create a simple database called Db1.mdb, with one table and one field, and add a couple of records. Close Access and open the Windows Explorer. Drag Db1.mdb

into the Briefcase, answering Yes to the prompts. Click OK (the default) when asked which copy should be allowed to make design changes. This results in the original database being allowed to make the design changes.

3. Open the Briefcase by double-clicking the Briefcase icon. Open the database by double-clicking Db1.mdb in the Briefcase. Add one or two records, and then close the database.
4. Open Explorer, double-click Db1.mdb to open the database in Access, and add another couple of different records. Close Access.

Now you should have two replicas of Db1.mdb, with different records added to each.

5. Open the Briefcase, highlight Db1.mdb, and choose Update Selection from the Briefcase menu. You should be notified that the Briefcase and original copies of the database have been modified, and the Merge icon (two green arrows pointing to each other) should be displayed. Click the Update button; you see an animated icon as the merge takes place.

If you now open either replica in the Briefcase or in Explorer, you'll see that the updates from each have been merged, unlike the result you would get from the Briefcase's default action.

NOTE

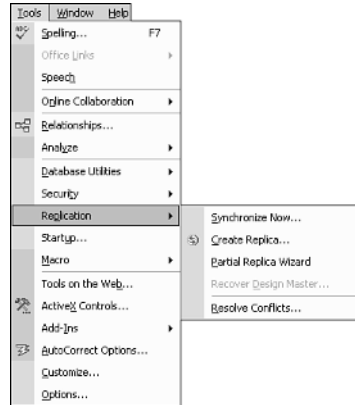
Access provides a significant increase in functionality over non-Briefcase-aware applications. The Access Briefcase Reconciler is an Access installation option that loads and registers an Access-specific DLL used for Windows, ensuring that changes are merged and not lost.

Products that don't provide specific support for the Briefcase use the default Briefcase Update action, which overwrites the original file with the changed file. If the Briefcase and networked copies have been modified, the user is prompted to choose the network copy or the Briefcase copy. When the user chooses a copy, the updates made to that copy become permanent; the updates in the other file are lost.

Creating and synchronizing replicated databases by using the Briefcase is extremely powerful and easy to do. As you'll see, however, greater functionality is possible when you use the Access menus, Replication Manager, and the Jet and Replication Objects (JRO) ADO component.

Access Menus

The Access Tools menu has a choice for replication (see Figure 22.2). This submenu provides all the functionality required by most users: you can convert databases into replicable databases, create additional replicas (including partial replicas) with the Partial Replica Wizard, synchronize replicas, and resolve conflicts.

**FIGURE 22.2**

The Access Replication submenu provides a way to handle replication through the UI.

Let's repeat the example from the preceding section, which described how to use the Briefcase—this time, however, use the Access menu commands. Follow these steps:

1. In Access, create a simple database named Db2.mdb, with one table and one field, and add a couple of records.
2. From the Tools menu, choose Replication and then Create Replica. Answer the prompts by clicking Yes, and accept the suggested default name: Replica of db2.mdb.
3. Add one or two records to the table in Db2.mdb, and then close the database.
4. Open Replica of db2.mdb and add another couple of records.

Now you should have two replicas of Db2.mdb, with different records added to each.

5. From the Tools menu, choose Replication and then Synchronize Now. Click Yes to close the open table. Click OK in the Synchronize Database dialog. A progress box appears; then you're notified that the synchronization completed successfully.

If you now open Db2.mdb or Replica of db2.mdb, you'll see that the updates from each have been merged. Also notice that when you open Db2.mdb, its title bar has been modified to read db2.mdb : Design Master. Similarly, when you open Replica of db2.mdb, its title bar reads db2.mdb : Replica.

NOTE

The other three Replication submenu options—Partial Replica Wizard, Recover Design Master, and Resolve Conflicts—are discussed later in this chapter.

Replication Manager

The Replication Manager utility, which ships with the Microsoft Office Developer (MOD) tools, is a sophisticated tool for managing distributed replicated databases (see Figure 22.3). Replication Manager is generally used only by database systems administrators.

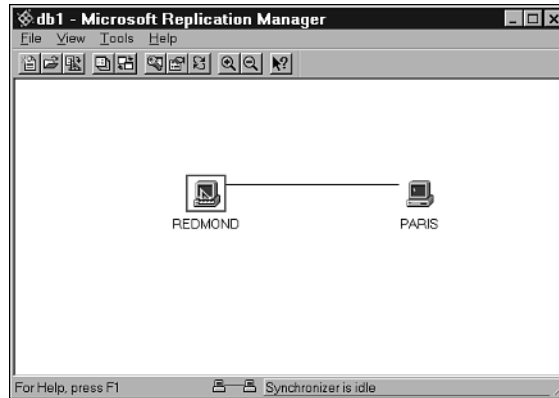


FIGURE 22.3

Being able to view the replicas included in a set is really useful when setting up users at remote locations.

The Replication Manager allows mobile laptop users to specify a shared network location where scheduled exchanges from a remote replica can deposit updates for later collection.

Beyond those provided by Access, the four most significant additional features in the Replication Manager are

- Configuration for synchronization of replicas over the Internet
- Configuration support for mobile laptop users
- Scheduling the synchronization of replicas
- Graphical topology of distributed replicas

Also, synchronization histories, reports, and various property sheets provide the information required to manage sophisticated replicated applications. A detailed synchronization log provides valuable information when troubleshooting distributed applications.

A solid line joining two sites—for example, Redmond and Paris—indicates that a synchronization is scheduled between them. To view and edit the schedule, simply double-click the line.

Jet and Replication Objects Programming

In any application that supports ADO, Jet and Replication Objects—referred to as JRO through the remainder of the chapter—provides a programmatic interface to replication functionality in Microsoft Jet databases. You can use methods and properties within JRO to convert a database into a replicable database, create additional replicas, synchronize replicas, and manage certain properties within a replicated database.

The features in Microsoft Jet 4.0 replication are exposed only in JRO programming, not in DAO programming. They include setting a replica's priority, executing an indirect synchronization in code, setting a replica's visibility, and much more. To use JRO, choose References and then Microsoft Jet and Replications Objects Library from the Access VBE's Tools menu.

Converting Databases to Replicas

Calling the `MakeReplicable` method on a database automatically converts the database to a replicable format. One feature is the ability to track data changes at the column (cell) level as well as the row (record) level.

A column is the most basic unit of information recognized in replication when the `ColumnLevelTracking`, sometimes referred to as CLT, is set on a table. The `ColumnLevelTracking` property is a custom property, which exists both on the database and on each individual table. `ColumnLevelTracking` is the default on each table when the `ColumnLevelTracking` property is set to `True` on the database. The database CLT property can be set only when making the database replicable. When a table's custom `ColumnLevelTracking` property is set to `True`, nonconflicting fields are merged when changes are transmitted during synchronization. When `ColumnLevelTracking` is set to `False` or equally when `RowLevelTracking` is set for the table in the table's property sheet in Access, if any field in a row (record) is modified, the whole record is marked as changed. Therefore, the whole record is updated when changes are transmitted during synchronization and individual fields aren't merged.

A `ColumnLevelTracking` table significantly reduces the potential for conflicts and simplifies the maintenance of replicated databases, if different users frequently edit the same data. `ColumnLevelTracking` creates a small performance hit because of the added system columns and compare logic, and therefore shouldn't be used in cases where users are doing large updates to isolated tables—that is, tables other users wouldn't be updating.

You can find an example of making a database replicable and setting the default tracking in the global module `Replication Routines` in the `Chap22.mdb` database, which you can find on this book's Web page at www.sampublishing.com. Also, the code in Listing 22.1 shows how to make a database replicable.

LISTING 22.1 Chap22.mdb: Converting a Database to a Replica

```
Sub MakeDatabaseReplicable(strTargetDB As String, boolCLT As Boolean)
    Dim TargetDB As New JRO.Replica

    'Error checking code here, making sure target exists...
    'Database is opened in exclusive mode by default.
    TargetDB.MakeReplicable strTargetDB, boolCLT

    Set TargetDB = Nothing

    'Confirm when finished
    MsgBox (strTargetDB & " is now a replicable database")

End Sub
```

NOTE

The database must be opened in Exclusive mode to make it replicable. This means that you can't have this code in the same database as the one you're making replicable.

CAUTION

Create a backup of your original database before converting it to a replicable database. You can't make a database non-replicable after it's converted.

The MakeDatabaseReplicable subroutine is used on the frmMakeDatabaseReplicable form. This form has the user input a database to make replicable in the txtTargetDB text box and the tracking default by checking the chkTrackingLevel check box, where True is ColumnLevelTracking. The user then clicks the cmdMakeDatabaseReplicable command button, whose OnClick event uses the code in Listing 22.2.

LISTING 22.2 Chap22.mdb: Input Form for Making a Database Replicable

```
Private Sub cmdMakeDatabaseReplicable_Click()
    Dim strTargetDB As String
    Dim boolTrackingLevel As Boolean
```

LISTING 22.2 Continued

```
'Place error checking code here to ensure that database exists.
strTargetDB = Me!txtTargetDB
boolTrackingLevel = Me!chkTrackingLevel

'Make target database replicable
MakeDatabaseReplicable strTargetDB, boolTrackingLevel
```

```
End Sub
```

The database is now “replicable,” allowing you to create additional replicas and synchronize between replicas.

In Listing 22.1, the database was converted into a replicable format. The default tracking of all existing tables and newly created tables is either `ColumnLevelTracking` or `RowLevelTracking`, depending on the value of the `boolTrackingLevel` variable. Of course, replication is useful only when there’s more than one copy of the database, so the next logical step is to create an additional replica. You do this by creating a new replica from a database that’s already replicable. (The source database must be replicable before you can create an additional replica.)

To create an additional replica, use the `CreateReplica` method, which is applied to an existing replicated database object and creates a new replica with the filename provided. Optionally, you can provide a user-friendly description (for example, *Jim’s replica of Customer Accounts*), define `ReplicaTypeEnum` and `VisibilityEnum`, set the replica `PriorityEnum`, and define the `UpdatabilityEnum` replication attributes. Each parameter, including the Enum values, are defined next. (Access 2000 introduced the `Visibility` and `Priority` arguments to make replication more powerful.) This is the syntax:

```
rep.CreateReplica strTargetDB, strDescription, ReplicaTypeEnum, _
    VisibilityEnum, PriorityEnum, UpdatabilityEnum
```

The `CreateReplica` method has the following parameters:

- `rep` is the source database.
- `strTargetDB` is a String value specifying the name and path of the replica to be created.
- `strDescription` is a String value describing the replica to be created.
- `ReplicaTypeEnum` is an optional Enum value indicating the type of replica to be created.

The following constants are valid values:

<code>jrRepTypeFull</code>	The replica is a full replica (the default).
<code>jrRepTypePartial</code>	The replica is a partial replica.

- *VisibilityEnum* is an optional Enum value indicating the replica's visibility. (Replica visibility is fully described later in the "Replica Visibility" section.) The following constants are valid values:

<code>jrRepVisibilityGlobal</code>	The replica is global (the default).
<code>jrRepVisibilityLocal</code>	The replica is local.
<code>jrRepVisibilityAnon</code>	The replica is anonymous.

- *PriorityEnum* is an optional Long value indicating the replica's priority for use during conflict resolution. The default value is `-1`, which indicates that the database should determine the default value. For global replicas, the default priority is 90% of the parent replica's priority.

The valid values for a global replica can be further restricted. If the user is the database administrator, the entire range is valid. Otherwise, the maximum value for priority is 90% of the parent replica's priority. For local and anonymous replicas, the value will always be 0 and can't be changed. This value is forced with the creation of the replica, and any other value is ignored.

- *UpdatabilityEnum* is an optional Enum value indicating the type of updates allowed. The following constants are valid values:

<code>jrRepUpdFull</code>	The replica can be updated (the default).
<code>jrRepUpdReadOnly</code>	All replicable objects are read-only.

CAUTION

Making a replica read-only isn't the same as removing all write privileges from the replica. Users might still be able to create data in the replica; however, this data will never be propagated to other replicas.

Also, read-only replicas must be in a read-write file directory. If a read-only replica is on a read-only network share, all synchronizations will fail.

By using the `frmCreateAdditionalReplica` form in the `Chap22.mdb` database, the code in Listing 22.3, attached to the `OnClick` event of the `cmdCreateAdditionalReplica` command button, takes the filename of an existing replicable database, `txtSourceReplica`, and creates the additional replica with the filename passed in the string `txtAdditionalReplica`. This form also takes the user's values for *Visibility*, *Priority*, and *Updatability*. In Listing 22.3, the new replica is given the description `Replica of the source`.

LISTING 22.3 Chap22.mdb: Creating Additional Replica

```

Private Sub cmdCreateAdditionalReplica_Click()
    Dim rep As New JRO.Replica
    Dim strSourceReplica As String
    Dim strNewReplica As String
    Dim eVisibility As VisibilityEnum
    Dim lVisibility As Long
    Dim lPriority As Long
    Dim eUpdateability As UpdateabilityEnum
    Dim lUpdateability As Long

    'Place your error checking here to ensure that your source DB exists.
    strSourceReplica = Me!txtSourceReplica
    strNewReplica = Me!txtAdditionalReplica
    lPriority = Me!txtPriority
    lVisibility = Me!frVisibility
    lUpdateability = Me!optReadOnly

    Select Case lVisibility
        Case 1
            eVisibility = jrRepVisibilityGlobal
        Case 2
            eVisibility = jrRepVisibilityLocal
        Case 3
            eVisibility = jrRepVisibilityAnon
    End Select

    Select Case lUpdateability
        Case -1
            eUpdateability = jrRepUpdReadOnly
        Case 0
            eUpdateability = jrRepUpdFull
    End Select

    ' Create an Active Connection to the source replica
    rep.ActiveConnection = strSourceReplica

    rep.CreateReplica strNewReplica, "Replica of " & strSourceReplica, _
        jrRepTypeFull, eVisibility, lPriority, eUpdateability

    Set rep = Nothing
    MsgBox "Your Replica has been created."
End Sub

```

Synchronizing Replicas

After you make a database replicable and create an additional replica, you want to synchronize the replicas so that updates are propagated.

The `Synchronize` method applies to a replicable database object. It exchanges schema and data with a target replica. This is always a “pair-wise” exchange between a source replica and a target replica, with an option to select the direction of changes from bidirectional, import only, or export only, and the exchange mode from indirect, Internet, or direct. This is the syntax:

```
rep.Synchronize Target, SyncType, SyncMode
```

The `Synchronize` method uses the following parameters:

- *Target* is a String value that specifies the path and filename of the replica with which to synchronize, the name of the synchronizer that manages the target replica, or the name of the Internet server that contains the target replica.
- *SyncType* is an optional Enum value that specifies the synchronization type to perform. The following values are valid for *SyncType*:

<code>jrSyncTypeExport</code>	Sends changes from the current replica to the target replica
<code>jrSyncTypeImport</code>	Sends changes from the target replica to the current replica
<code>jrSyncTypeImpExp</code>	Sends changes from the current replica to the target replica and vice versa (default)

- *SyncMode* is an optional Enum value specifying the method of synchronization. The following values are valid for *SyncMode*:

<code>jrSyncModeIndirect</code>	Indirect synchronization (default)
<code>jrSyncModeDirect</code>	Direct synchronization
<code>jrSyncModeInternet</code>	Indirect synchronization over the Internet

NOTE

Indirect and Internet synchronization modes aren't valid unless they're first configured with Replication Manager.

The code in Listing 22.4 takes the filenames and exchange type of the source and target replicas and does a direct exchange. You can find this code on the `frmSyncTwoReplicas` form in the `Chap22.mdb` database (on this book's Web page at www.sampublishing.com). The code is attached to the `cmdSyncReplicas` command button.

LISTING 22.4 Chap22.mdb: Performing a Direct Exchange

```

Private Sub cmdSynchronize_Click()
    Dim rep As New JRO.Replica
    Dim strSourceReplica As String
    Dim strTargetReplica As String
    Dim eExchangeType As SyncTypeEnum
    Dim lExchangeType As Long

    'Place error checking here to confirm existence of files and folders
    strSourceReplica = Me!txtSourceReplica
    strTargetReplica = Me!txtTargetReplica
    lExchangeType = Me!frExchangeType

    Select Case lExchangeType
        Case 1
            eExchangeType = jrSyncTypeImpExp
        Case 2
            eExchangeType = jrSyncTypeImport
        Case 3
            eExchangeType = jrSyncTypeExport
    End Select

    'Set ActiveConnection to the Source Replica
    rep.ActiveConnection = strSourceReplica

    'Synchronize the replicas
    rep.Synchronize strTargetReplica, eExchangeType, jrSyncModeDirect

    Set rep = Nothing

    MsgBox ("Synchronization of " & strSourceReplica & " and " & _
        strTargetReplica & " is complete.")

End Sub

```

CAUTION

Making an exchange *unidirectional* (that is, either import or export) might still result in a bidirectional exchange. The Exchange parameter applies only to data exchanges, and replicas will always exchange new schema if required. This is a great way to propagate schema changes while avoiding affecting user data.

When creating a replica and after synchronizing, you also should compact the replica because it will have a lot of excess.

Understanding the Design Master and Replicas

A replica set can contain only one Design Master, but it can contain as many replicas as needed. The Design Master is the only replica in which changes to the database design are allowed. You can designate any replica with global visibility as the Design Master. Make sure, however, that only one replica is marked as the Design Master at any time to prevent two replicas from making design changes that would be incompatible in a synchronization. When a database is made *replicable*, properties are set that identify the database as the Design Master.

Let's investigate each attribute in a bit more detail to see why the Design Master is unique. As the following table shows, the Design Master or any replica can create a new replica. There are no restrictions within Access, nor can this be prevented by permissions. You can leave Access and create a copy of the database, and thereby create a new replica, so any Access security would be easily avoided.

<i>Functionality</i>	<i>Design Master</i>	<i>Replica</i>
Creates new replica	Yes	Yes
Updates, inserts, and deletes data	Yes	Yes
Creates local tables or queries	Yes	Yes
Creates replicable tables or queries	Yes	No
Creates replicable Access objects (forms, reports, pages, macros, or modules)	Yes	No

The Design Master or any non-ReadOnly replica can insert new records, update existing records, or delete records if the user has permission. Normal Access security is enforced, so if you want to restrict users, you can use the standard Access object/permission/user security. (See Chapter 20, "Securing Your Application," for more information.)

The Design Master is the only database in the replica set permitted to create replicable objects. When a table or query is created in the Design Master, it's initially created as a local table or query (non-replicable). To make it global in the Access database container, right-click the object and check its *Replicable* property. Now when you synchronize with the other replicas in the set, the table or query will be propagated. Any single replica can also contain local tables or queries that exist only in that replica.

Access allows the replicability of Access objects (forms, reports, Data Access Page links, macros, and modules). Since Access 2000, all Access objects are stored in a single binary large object (BLOB) within the database file. Therefore, all Access objects are either replicable or

local, which must be specified before making the database replicable. See the later section “Using Replicable and Non-Replicable Objects” for more information about the Access project.

You can move the Design Master attribute to one replica during a synchronization. If one database in an exchange is the Design Master, the user can transfer the Design Master to the replica. You can also use the JRO `DesignMasterID` property to move the Design Master property to another replica.

Recovering the Design Master

In case of a disaster (the Design Master is destroyed), you can promote one of the other replicas to become the Design Master for the replica set. To recover a Design Master for the replica set, from the Tools menu choose Replication and then Recover Design Master.

You can also choose these same menu options in Replication Manager. Also, through code you can set the JRO `DesignMasterID` property to allow a replica to become the Design Master.

CAUTION

Two Design Masters in the replica set will be prevented from exchanging with each other. Also, always synchronize the proposed Design Master with all other replicas in the replica set to ensure that all schema changes are applied. If you don't do this, and other replicas have applied more recent schema changes from the old Design Master, future synchronizations might fail.

Replication Visibilities

Microsoft Jet 4.0 replication defines three degrees of visibility for replicas. *Visibility* is a level of restriction and functionality a replica has in a replica set. A replica's visibility can be defined as global, local, or anonymous:

- Global replicas have the same visibility as replicas created in Access 97/95 (Microsoft Jet 3.x). A global replica can synchronize with all other global replicas in the replica set. A global replica can also synchronize with any replica created from it, with some exceptions. The Design Master's visibility is always global. By default, any new replica created from a global replica is also a global replica. From a global replica, you can create a global (default), local, or anonymous replica.
- Local replicas can synchronize only with their parent, a global replica, and aren't permitted to synchronize with other replicas in the replica set. This way, you can easily

control a replica set's topology. For example, you can use local replicas to enforce a star topology at individual sites where you want to ensure that synchronization between sites goes through a global hub at each site. Other replicas in the replica set won't be aware of the local replica. The parent replica can schedule synchronizations with a local replica by using Replication Manager. The parent replica proxies any replication updates and conflicts for the local replica, meaning that all changes appear as though they came from the parent replica. Local replicas always have a priority of 0.

- Anonymous replicas are important for Internet scenarios in which you don't want to keep track of every replica's addition or synchronization in the replica set. An anonymous replica can synchronize only with its parent, a global replica. Anonymous replicas are similar to local replicas, except that anonymous replica information isn't permanently stored in the system tables. These replicas subscribe through the Internet and don't have any particular identity in the parent replica; instead, they proxy their identity for updates through the parent replica. By removing the information about the anonymous replica after a period of inactivity, the overall size of replicas in the replica set is reduced if many replicas in your set are anonymous. It also helps to keep out unnecessary topology information about replicas that participate only occasionally. The parent replica can't schedule synchronizations with an anonymous replica. Anonymous replicas always have a priority of 0.

CAUTION

If you move the parent of a local or anonymous replica by using any method other than Replication Manager's Move Replica command, you lose the ability to synchronize.

Replication System Columns, Tables, and Other Mysteries

When a database is converted to a replicated database, it undergoes many changes. Many of these changes use the 128-bit GUID (Globally Unique Identifier) data type. Its primary advantage in a distributed, replicated database is that if two GUIDs are created at two separate, disconnected PCs, you can virtually guarantee that they'll be never be the same.

GUIDs are used anywhere a unique identifier is required and must be generated even when it's impossible to check the values already used at other replicas. A GUID individually identifies the following:

- Records in any replicated table (ReplicationID)
- Replicas (ReplicaID in the MSysReplicas system table)

- Replica sets (GlobalDbid in the MysRepInfo system table)
- Tables (s_GUID in the MSysTableGuids system table)

These GUID fields are usually hidden from view; to see them, choose Options from the Tools menu, click the View tab, and select System Objects. These system objects are protected from modification by users and developers.

Each record in every user table gains at least three new fields—four if ColumnLevelTracking is the default when it's made replicable:

- s_ColLineage is an OLE Object field of variable size (minimum of 4 bytes) used to resolve the winner in a ColumnLevelTracked data conflict and to optimize record exchanges between replicas. Thus, this field is added only if the table is ColumnLevelTracked.
- The s_Generation field is a 4-byte-long integer used to optimize exchanges between replicas. Each time an exchange occurs at a replica, its generation counter is incremented. This field is indexed.
- Whenever a new record is inserted, it's given a GUID in the s_GUID field that's used as the unique key to distinguish this record from any other new records inserted at other replicas. This field also serves as the unique key to identify conflicts when two replicas update the same record. Because a user can never modify the s_GUID field, it's always possible to track conflicts; this wouldn't be possible if the user-defined primary key were the conflict identifier because it would be subject to potential change. The s_GUID field is indexed.
- s_Lineage is an OLE Object field of variable size (minimum of 4 bytes) used to resolve the winner in a data conflict and to optimize record exchanges between replicas.

The addition of these extra fields increases the database's size. System tables are also added to the database to keep track of replicas in the set, exchange histories, schema changes, deleted records, and other information internal to replication.

Figure 22.4 shows a replicated database with a single text field.

Text	s_ColLineage	s_Generation	s_GUID	s_Lineage
Test	Long binary data	0	(7EBD36BF-A261-429A-B261-A0C6543E242E)	Long binary data

FIGURE 22.4

Remember that your database will grow in size when you make it replicable.

Using Replica Sets

All replicas derived from the conversion of the original database belong to the same replica set. This is very important because only replicas from the same set can exchange and synchronize with each other. The ramification is that replicas from different sets can never synchronize with each other. Each replica set is identified by a unique replica set ID.

NOTE

If you try to synchronize between replicas in different replica sets, the exchange will fail gracefully with the message The replicas are from different replica sets and can't be synchronized. You can never modify this behavior. If you ever need to get data from a replica in one replica set into a replica in another replica set, you must either import, use append queries, or cut and paste the data.

Understanding Replica Set Topologies

Another concept introduced with replicated databases is the replica set topology. This refers to the distribution of replicas and the connections between them, and might be thought of as analogous to the topology of PCs on a LAN. The selection of an appropriate replica set topology is vital for the robust and efficient management of the synchronizations between replicas.

The following sections analyze the singly connected list and star and hub topologies. Using replicas with local or anonymous visibility will enforce a star/hub topology because those replica visibilities can synchronize only with the parent (global) replica.

Singly Connected List

This very simple topology consists of four replicas (Redmond, Paris, London, and Berlin) connected in a list:

- Redmond can synchronize only with Paris.
- Paris can synchronize with Redmond and London.
- London can synchronize with Paris and Berlin.
- Berlin can synchronize only with London.

Assume that replicas exchange daily and, more importantly, the clocks in each computer at each location aren't synchronized. This means that Paris might exchange with London before exchanging with Redmond. Also assume that this is the situation with exchanges between London and Berlin. Therefore, if Redmond made some updates on Monday, these would reach Paris on Tuesday, London on Wednesday, and Berlin on Thursday.

A singly connected list has two potential disadvantages:

- Latency (or delay) of update propagation can be significant.
- A failure between any two replicas will prevent update propagation to replicas up- or downstream from the point of failure.

Star and Hub Topologies

Consider the same four replicas (Redmond, Paris, London, and Berlin) connected in a star and hub topology (see Figure 22.5).

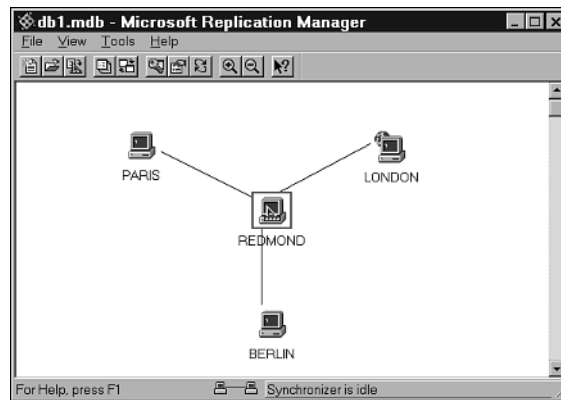


FIGURE 22.5

With the star and hub replicas topology here, everything goes through Redmond.

Now, assume that the schedule is initiated at Redmond. Redmond exchanges with Paris, and then London, and then Berlin. Redmond immediately repeats the process, again exchanging with Paris, and then London, and then Berlin. All replicas are now up-to-date and consistent.

In general, a star and hub topology is an excellent choice, balancing robustness and efficiency without the burden of a complex scheduling algorithm and associated management, which is often the source of problems that are difficult to identify. Latency in update propagation can be easily controlled.

Automating Star and Hub Synchronization

A simple example demonstrates how you can code synchronizations between replicas in a star and hub topology. It's assumed that you already have code that writes the path to any new replica created in a table called Replicas, and you define which replica is the hub.

The Replicas table has two fields:

- ReplicaPath, a string pathname to each replica in the replica set
- Hub, a Boolean field that indicates “Yes” if it’s the hub, or “No”

The code simply creates a recordset from the Replicas table, and then loops through the records. If the current record doesn’t point to the hub, synchronize by using ReplicaPath.

The code in Listing 22.5 is behind a button named cmdSyncAllReplicas on the Synchronize Star & Hub Topology form.

LISTING 22.5 Chap22.mdb: Synchronizing Star and Hub Replicas

```
Private Sub cmdSynchronizeAllReplicas_Click()
    ' Synchronize replicas in a star & hub topology
    ' The 'Replicas' table stores the path to the replica
    ' and a Y/N if it is the hub replica

    ' This example has four entries in the "Replicas" table :
    ' c:\temp\db1.mdb - the hub
    ' c:\temp\db2.mdb
    ' c:\temp\db3.mdb
    ' c:\temp\db4.mdb

    ' This example expects this database is saved in c:\temp\db1.mdb

    Dim rep As New JRO.Replica
    Dim conn As New ADODB.Connection
    Dim rsReplicas As New ADODB.Recordset

    conn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data source=c:\temp\db1.mdb;"
    Set rep.ActiveConnection = conn

    rsReplicas.Open "Replicas", conn, adOpenKeyset, adLockOptimistic
    Do Until rsReplicas.EOF
        ' Loop through replicas, synching with each
        If rsReplicas!Hub = False Then ' Hub can't sync with itself
            rep.Synchronize rsReplicas!ReplicaPath, ,jrSyncModeDirect
            MsgBox (rsReplicas!ReplicaPath & " synchronized")
        End If
        rsReplicas.MoveNext
    Loop
```

LISTING 22.5 Continued

```
rsReplicas.Close  
MsgBox ("Synchronization complete")
```

```
End Sub
```

CAUTION

You might have already noticed that the `MSysReplicas` table already contains the path to each replica in the replica set. It's tempting to use these values rather than create your own, but that would be unwise. Microsoft provides no guarantee that system tables will maintain their format between releases. If your code relies on values in `MSysReplicas`, your code might not work in future releases of Access. Also, with the new replica visibilities, you might not be synchronizing your entire replica set.

22

WELCOME TO THE
WORLD OF
DATABASE

Distributing Replicable Applications

Many developers find replication an ideal way to distribute application updates to their users. It's tempting to

- Create a database application.
- Make it replicable (this becomes your Design Master).
- Create additional replicas from the Design Master.
- Whenever you want to update the application design, make the modifications in the Design Master and then synchronize the Design Master with the replicas to distribute your changes.

TIP

Move the Design Master to one of the stars. Now you, as the application designer, can control when you synchronize with the hub, which will then distribute your application to the other replicas. You can fully test your design modifications in a controlled environment before propagating them to all your users.

Also, if you make your Design Master the hub in a star and hub topology, users will be offered the choice of moving the Design Master to their replica when they synchronize. If the Design Master isn't at the hub or a public network share location, users can't initiate a synchronization with the Design Master and can never make unauthorized global design changes.

For additional protection and control, ensure that the Design Master isn't on a public network share. If other replica users don't have network file-read permission for the Design Master's location, they can't initiate an unauthorized synchronization.

Using Replicable and Non-Replicable Objects

Objects that are propagated around all replicas are termed *replicable objects*. Objects that exist at a single replica are termed *non-replicable*, or *local*, objects.

If you create a new table or query in the Design Master, it will be made local by default. If you want to propagate the table or query, you must set its property to `Replicable` (right-click the object in the database container window, choose Properties, and then select the Replicated check box).

TIP

A common mistake is to create a new table in the Design Master and then synchronize with a replica and expect the new object to appear at the replica. When the table isn't seen at the replica, the assumption is that replication isn't working. You must set the object's property to `Replicable` before it will propagate from the Design Master to other replicas.

As mentioned earlier, Access allows the replicability of Access objects (forms, reports, Data Access Page links, macros, and modules). Because all Access objects are stored in a single BLOB within the database file, they are either replicable or local, which must be specified before making the database replicable. In this format, an individual Access object can't be identified or tracked by replication. So if the Access project in the Design Master is made replicable, all Access objects are made replicable and synchronized if any single object is modified. When the Access project is marked as replicable, project objects can't be created or modified in any replica except the Design Master. You can choose not to make the Access project replicable before making the database replicable by setting the `ReplicateProject` property in the custom database properties to `False`. In this case, the Access project in each replica isn't replicable, and all Access objects created in a replica are local.

NOTE

If the Page object is replicable, the Data Access Page link (.htm file) must be stored on a network so that the page can be accessed from the Design Master and all replicas in the replica set.

Enforced relationships between tables are permitted only when both are replicated or when both are local. You can't have an enforced relationship between a local and a replicated table.

Creating Partial Replicas

Partial replicas allow the creation of replicas with data subsets. This contrasts a full replica, which contains all the data. Partial replicas are particularly useful where users at certain sites either don't need all the data (for example, if they're a provincial office and don't need the data for the whole organization), or if they have restricted hardware, such as a laptop.

You create a partial replica by setting a filter on one or more tables and defining the relationships that should be applied to extract the data from the other tables in the database. A partial replica can be created either through JRO code or by using the Access 2000 Partial Replica Wizard.

Partial Replica Wizard

The easiest way to get started with partial replicas is to use the Partial Replica Wizard.

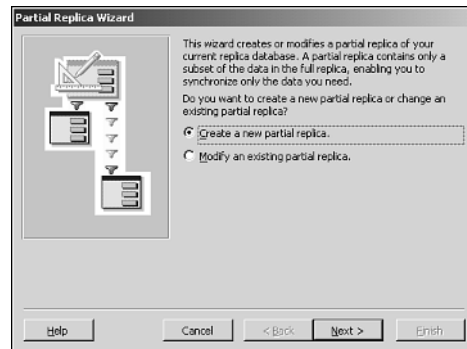
By using the wizard, you first select one table on which you will apply a filter. Let's call this the focus table. Apply one or more filters to the focus table, and then select the relationships between the focus table and the other tables in the database.

NOTE

The Partial Replica Wizard restricts you to filters on a single table. For more complex filter criteria, you must use code.

For example, by using the Northwind database that ships with Access, follow these steps:

1. Make the database replicable by choosing Replication and then Create Replica from the Tools menu.
2. From the Tools menu choose Replication and then Partial Replica Wizard to start the wizard (see Figure 22.6).

**FIGURE 22.6**

The Partial Replica Wizard guides you through creating a new partial replica or modifying an existing partial replica.

NOTE

You can create a partial replica from a full replica with global visibility if you use the wizard. Otherwise, if you create the partial replica in code, visibility can also be local or anonymous. You can't create a partial replica from another partial replica.

3. Select Create a New Partial Replica and click Next.
4. Supply an appropriate filename for the partial replica. You can either accept the suggested filename (Partial Replica of *<filename>*) or enter another name.
5. Select the table that will have the filter expression (for example, Categories).
6. Set the filter expression by selecting one or more field and expression pairs—for example,

[Category ID] > 2 AND [Category ID] < 5

Partial replica filters don't support user-defined expressions.

TIP

Using the filter expression in the replica description—for example, Nwind where Category ID > 1—helps you remember how each replica was created, and assists later when you troubleshoot the replica.

7. Specify the data to include for the other tables in the database. The default is to select all records. Remove the check mark to select only those records related to those specified by the filter, thereby creating a smaller partial replica.

Creating a report is strongly advised because you might need to refer to it at a later date.

NOTE

A partial replica actually contains all the tables of a full replica. The filter in step 6 simply forces the data to fall within the filter parameters—therefore, some tables might be empty.

Creating Partial Replicas in Code

JRO programming allows a more flexible approach to creating partial replicas, allowing filters on more than one table.

NOTE

There are restrictions on using the Partial Replica Wizard for editing a partial replica's filter that has been created by using JRO. If the replica has a complex filter created through JRO, the wizard will warn you, and the filter will be replaced by the wizard's single table filter expression.

Creating a partial replica is a four-step process:

1. Create the replica and identify it as a partial replica. (At this stage, it has all the tables defined, but no data.)
2. Create the filters for the tables.
3. Define the filter relationships.
4. Populate the partial replica with the data that satisfies the table and relationship filters.

Listing 22.6 shows the code for creating the partial replica.

LISTING 22.6 Chap22.mdb: Creating a Partial Replica

```
Sub CreatePartialReplica(strSourceRep As String, strPartialRep As String)
' This code demonstrates how to create a partial replica with a
' relationship filter and a table filter.
' NOTE: PopulatePartial requires an exclusive connection.

Dim repRW As New JRO.Replica
Dim repPartial As New JRO.Replica
```

LISTING 22.6 Continued

```

' Place error checking code here to ensure Source Replica exists.
repRW.ActiveConnection = strSourceRep

' Create a Partial Replica
repRW.CreateReplica strPartialRep, _
    "Partial Replica of " & strSourceRep, jrRepTypePartial

Set repRW = Nothing

' PopulatePartial requires an exclusive connection to the database.
repPartial.ActiveConnection = "Data Source=" & strPartialRep & ";" & _
    "Mode=Share Exclusive"

' Set a filter on the Customers Table for Region='CA'
repPartial.Filters.Append "Customers", jrFilterTypeTable, "Region='CA'"

' Give me only the Orders for the above Table Filter by
' Setting a Relationship Filter
repPartial.Filters.Append "Orders", jrFilterTypeRelationship, _
    "CustomersOrders"

repPartial.PopulatePartial strSourceRep
Set repPartial = Nothing

End Sub

```

You must set a filter for each table that has the data you require. The default is to supply no rows for each table; if you don't set a filter for a particular table, you don't get any rows. Remember, a partial replica is created empty. Only by setting filters and relation properties can you populate the partial replica with data rows.

Again, if you don't specifically set the relationship filter, the default is false, which means that there are no data rows in the related table.

NOTE

You must synchronize from a partial replica to a full replica. You can't synchronize between two partial replicas.

When you combine filters in a partial replica, they create a logical OR result. For example, if you set two filters—for Customers where Region = 'CA' and Orders where Value > 100—you get records for all customers from California and all records for orders where the value is greater than \$100.

Replicating Back-End and Front-End Applications

Some replication applications will benefit from a back-end/front-end design. In this design, the front end contains the forms, reports, and code; the back end contains the tables and data. The front end uses attached tables to the back-end data. The front and back ends are two separate replica sets, with each managing its synchronizations separately. (For more information on front and back ends, see Chapter 21, “Handling Multiuser Situations.”)

You can develop a front-end application by using sample back-end data. When the development phase is complete, the application is synchronized with other replicas in the replica set that attach to the “real” back-end data. This development methodology allows third-party developers to distribute applications to numerous customers who never need to share private data.

NOTE

Linked tables have special attributes in replicated databases. Links to tables created in the Design Master are propagated to all replicas. A linked table is a global property. The actual path to the attached table is also propagated but is treated as a local property. The path created at the Design Master is propagated to replicas as a default. A replica can modify the path to the attached table, even if that replica isn't the Design Master.

Handling Replication Conflicts

Replicated databases introduce a new problem: What happens when users at different replicas simultaneously make incompatible modifications to the data or design? As of Access 2000, events causing synchronization conflicts and errors are viewed simply as synchronization conflicts. A single mechanism is used to record and resolve them, making resolution of such problems easier.

When a conflict occurs, a resolution algorithm determines a winner and a loser. This release of replication introduces an algorithm whereby replicas in a replica set are assigned priorities, and the highest priority replica wins in a synchronization conflict. Where priorities are equal, the replica with the lowest replica ID wins. The winning record is placed in the table in both replicas. The losing record is placed in a “conflict table” and replicated to both replicas. The conflict table is a replicated table named *TableName_Conflict* (where *TableName* is the name of the table in which the conflict occurred). Access automatically invokes the Microsoft Replication Conflict Viewer to help you resolve entries in conflict tables.

Consider the simple case of two users simultaneously updating the same record (row) in a `RowLevelTracked` table, or the same cell (column) in the record in a `ColumnLevelTracked` table, in different replicas. (Of course, in a traditional, non-replicated database, the locking mechanism will prevent such a problem.) When the replicas try to synchronize, Access recognizes that there's a conflict. The “winner” is the record in the replica with the higher priority.

The one exception to this data-conflict rule is that a deletion always wins. For example, if one replica modifies a record and another replica deletes the record, after an exchange the record is deleted at both replicas—although the losing record is still placed in the conflict table and replicated to all replicas.

If a conflict occurs, after synchronization is successfully completed, users are told that a conflict has occurred when they open the replica or choose Replication and then Resolve Conflicts from the Tools menu. They are asked whether they want to resolve the conflict immediately; if so, the Microsoft Replication Conflict Viewer appears, displaying the table with the winning and losing records side by side. Users then need to implement the following steps:

1. In the Conflict Viewer, select a table with conflicts and click View.
2. For each conflict shown in the Conflict Viewer, choose one of the following options:
 - To keep the data in the replica that won, click Keep Existing Data.
 - To modify the data in the replica that won and copy it to the replica that lost on the next synchronization, click Keep Revised Data.
 - To keep the data in the replica that lost and overwrite the data in the replica that won on the next synchronization, click Overwrite with Conflicting Data.
 - To modify the data in the replica that lost and overwrite the data in the replica that won on the next synchronization, click Overwrite with Revised Data.
3. Click Resolve.
4. Repeat steps 2 and 3 as many times as necessary to resolve each conflict in the table.

NOTE

In one rare case, users can completely lose their data. On a delete-style conflict, if users don't first resolve the deletion of a supporting record, but instead first restore the records that depend on that record, all data will be lost.

Using the Conflict Viewer

Let's set up a sample database to use with the Microsoft Replication Conflict Viewer. In Access, create a very simple database called `Db3.mdb`, with one table and one field, such as

Table = 'tblCustomers'

Field = 'Name', type Text

No primary key

Follow these steps:

1. Enter the name **Tom** as the first record (see Figure 22.7).

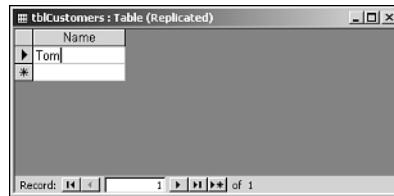


FIGURE 22.7

This table is used in a simple database for the conflicts example.

2. From the Tools menu, choose Replication and then Create Replica to make the database replicable and to create a second replica. Call the second replica *Replica of db3.mdb*. The original database is now the Design Master for the replica set.
3. In the Design Master (db3.mdb), edit the first record in the tblCustomers table so that it now reads Thomas. Close the Design Master.
4. Open Replica of db3.mdb and edit the first record in the tblCustomers table so that it now reads Tommy.
5. Synchronize the replicas (from the Tools menu, choose Replication and then Synchronize Now; then choose OK at any prompts). A conflict will have occurred in one of the two replicas. Because the replicas were created by using a default priority, the Design Master will “win” because its priority is higher than the other replica. In either case, the conflict table tblCustomers_Conflict will be replicated to both and Access will prompt you that a conflict has occurred (see Figure 22.8).

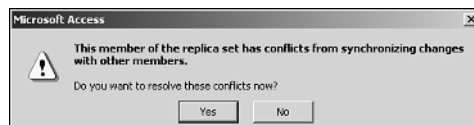


FIGURE 22.8

This dialog appears when a conflict occurs.

NOTE

If you didn't get a dialog warning you that a conflict occurred, do the following:

- Confirm that you edited the same field in the same record, which originally contained Tom, and that you entered different data at each replica. If you entered the same data at each replica, Access is smart enough to know there's no conflict.
- Confirm that you edited this record in *both* replicas.
- Confirm that you synchronized the replicas.

6. Click Yes to resolve the conflicts now.
7. You're informed that there's one conflict in the `tblCustomers` table. Click the View button. The existing (winning) record is displayed next to the conflict (losing) record (see Figure 22.9).

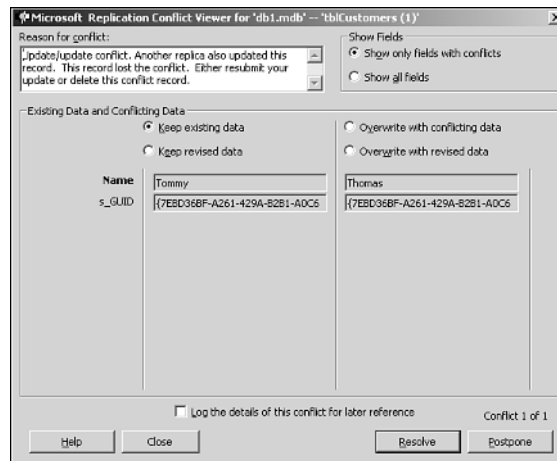


FIGURE 22.9

The Conflict Viewer helps you decide which record to keep.

8. In the Conflict Viewer, do one of the following:
 - Select Keep Existing Data and click the Resolve button to leave the data in its current state.

- Select Keep Revised Data to edit the winning record, and then click Resolve to send the new changes to the next synchronization.
- Select Overwrite with Conflicting Data to keep the data from the losing replica, and then click Resolve to send the record to the next synchronization.
- Select Overwrite with Revised Data to edit the losing record, and then click Resolve to send the new changes to the next synchronization.

After you resolve the conflict, the conflict record and the `tblCustomers_Conflict` table are deleted.

Using an Alternative Conflict-Resolution Algorithm

You might be tempted to replace the Microsoft Replication Conflict Viewer with a conflict resolver that better meets your business needs. For example, you might want to resolve conflicts so that the most recent update automatically wins. Consider a number of issues when doing this:

- The algorithm used to determine the “winner” and “loser” is hard-coded into Access and can’t be modified. Therefore, you can’t modify how conflicts are initially recorded. However, you can watch for conflicts at each replica and provide a conflict resolver to replace the one that shipped with Access. This might be termed a *deferred* conflict resolver because it’s not called at the time the conflict is first noticed by Access, but at a later time. The task of this resolver is to process the conflict record Access has created.
- A replacement conflict resolver must be a function written in VBA and identified as a database property. To register your new function, choose Database Properties from the File menu and select the Custom tab. Set the Name to `ReplicationConflictFunction`; set the Type to Text, and enter the name of your VBA function in the Value text box. Or use JRO to set the `ConflictFunction` property to the name of your custom function.

Take care in selecting a conflict-resolution algorithm. A date/time base algorithm, for example, in which the most recent update wins, must take into account computers that might be in different time zones (such as Greenwich mean time in England and Pacific standard time in the U.S.), when the clocks in each computer aren’t synchronized, or when a user resets his date/time for daylight saving time. If great care isn’t taken, each scenario can lead to diverging data in replicas.

Your algorithm also must deal with all conflict types that might occur during synchronizations.

CAUTION

Only experienced users should try to resolve conflicts themselves, rather than let Access handle it. You should estimate at least 500 lines of VBA code to replace the Microsoft Replication Conflict Viewer. If your replacement conflict resolver doesn't correctly handle all the conflicts described in the following sections, your conflict tables will grow.

Identifying Replicas with Conflicts

The algorithm in Listing 22.10 identifies replica tables that have outstanding conflicts. It takes the replica's name and prints out any tables with conflicts.

LISTING 22.10 Chap22.mdb: Finding the Tables with Replication Conflicts

```
Sub Conflicts(strReplica As String)
    Dim rep As New JRO.Replica
    Dim conn As New ADODB.Connection
    Dim rs As New ADODB.Recordset

    ' Set required connections
    conn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" + _
        "Data source=" & strReplica & ";"
    rep.ActiveConnection = conn

    ' Find tables with conflicts
    Set rs = rep.ConflictTables
    rs.MoveFirst
    Do While rs.EOF = False
        Debug.Print rs(0) & " Table had a conflict!"
        rs.MoveNext
    Loop

    rep = Nothing
    conn = Nothing

End Sub
```

Using the Last-Update-Wins Algorithm

The following algorithm is designed to work with replicas that have a special Date/Time field, Date, in each row that records the time an update was made. It makes no allowances for the time zone of each replica (such as GMT or PST).

The pseudocode for this algorithm is

```
Open database
For each table
  If there is a conflict table
    For each conflict row GUID == base data row GUID
      If conflict has a later date/time stamp field
        replace base row with conflict row
        delete conflict table row
    Next conflict table row
Next table
```

Listing 22.11 shows the actual code.

LISTING 22.11 Chap22.mdb: Last Update Wins Conflict Algorithm

```
Private Sub Resolve(strReplica As String)
  Dim rep As New JRO.Replica
  Dim conn As New ADODB.Connection
  Dim cmd As New ADODB.Command
  Dim rs As New ADODB.Recordset ' Create record set for conflict tables
  Dim rstWin As New ADODB.Recordset ' Create record set for conflict winner
  Dim rstConflict As New ADODB.Recordset ' Create record set for loser
  Dim fld As ADODB.Field

  ' Open the database & tables
  conn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "Data source=" & strReplica & ";"
  Set rep.ActiveConnection = conn
  Set cmd.ActiveConnection = conn

  ' Retrieve the Conflict Tables recordset
  Set rs = rep.ConflictTables
  rs.MoveFirst
  Do While rs.EOF = False
    ' Get the records from the conflict table
    cmd.CommandText = "Select * From " & rs(1) & " Order By s_GUID"
    rstConflict.Open cmd, , adOpenKeyset, adLockOptimistic
    ' Get the records from the base table
    cmd.CommandText = "Select * from " & rs(0) & " Order By s_GUID"
    rstWin.Open cmd, , adOpenKeyset, adLockOptimistic
    ' Process each record, starting at the first
    rstWin.MoveFirst
    rstConflict.MoveFirst
    While Not rstConflict.EOF
      ' If date on the conflict row > date on winning row
      ' then resubmit the conflict row
```

LISTING 22.11 Continued

```

    If rstConflict("s_GUID") = rstWin("s_GUID") Then
        ' This code assume you have a Date column in your table
        ' which records the date/time when the record was edited.
    If rstConflict("Date") > rstWin("Date") Then
        For Each fld In rstConflict.Fields
            If fld.Name <> "ConflictRowGUID" And _
                Left$(fld.Name, 2) <> "s_" Then
                rstWin(fld.Name) = rstConflict(fld.Name)
            End If
        Next
        rstWin.Update
    End If
    ' Remove conflicting record & clean up
    rstConflict.Delete
    rstConflict.MoveNext
End If
rstWin.MoveNext
Wend
rstWin.Close
rstConflict.Close
rs.MoveNext
Loop      ' Next Table

rep = Nothing
conn = Nothing

End Sub

```

Understanding Various Replication Conflicts

The following summarizes the types of synchronization conflicts that can be encountered:

- **Simultaneous update.** The most frequent conflict type is when two users simultaneously update data in the same record or field, depending on the tracking level set for the table. The losing record is logged to the appropriate conflict table.
- **Update-delete.** Since Access 97, delete actions have always been processed. This means that, no matter what the replica's priority is, the deleted record will always win in case of a conflict and be deleted. However, in Access 200x, the record that's about to be deleted is checked to see whether any updates were unknown before the delete occurs. If this is the case, the updated record is logged to the appropriate conflict table.

- **Unique key.** These errors can occur when there's a unique key constraint on a field in a table. Two replicas enter a record that contain the same key value, even though only unique values are permitted.
- **Table-level validation.** A record contains a field value that doesn't meet a table-level validation rule. To prevent a TLV conflict, synchronize with all replicas immediately before applying a TLV rule, and then again immediately after.
- **Referential integrity.** These conflicts can occur when a relationship between tables is enforced and cascading updates and deletes aren't enabled. You can avoid such conflicts through careful application design, such as if cascades for updates and deletes are always selected whenever referential integrity is enforced. There are three kinds of referential-integrity conflicts:

On delete	The primary key record is deleted in another replica, so the foreign record is rejected.
On update	The primary key record is updated in another replica, so the foreign record is rejected.
Foreign key	A foreign-key violation resulted from an invalid primary key record that was involved in another replication-conflict type.
- **Locking.** Locking errors are the easiest to understand and usually fix themselves without any user intervention. The record change couldn't be applied during synchronization because another user locked the table involved. The synchronization fails, but no conflict is logged. The solution is to try the synchronization again when the table isn't locked.

NOTE

Design changes are always processed before data changes. Also, all conflicts, except update-delete and locking, are resolved by using the priority of the replicas involved.

NOTE

Partial replicas receive conflicts associated with all records in their filter, including newly added records that are added to the partial replica during synchronization that might have conflicts associated with them.

Understanding Replication Synchronizers

An important aspect of Access replication that has been deferred in this chapter until now is the Synchronizer. If you want to use replication over the Internet—to set scheduled synchronizations or support rarely connected laptop users—you must understand and use replication synchronizers.

The Synchronizer ships with Replication Manager in the Microsoft Office XP Developer (MOD). The Synchronizer is a background software agent that manages the scheduled synchronizations between replicas and provides the necessary support for rarely connected users. It also provides support for Internet and indirect synchronization (in addition to the aforementioned ability to schedule synchronizations).

Replication Manager 200x provides support for centralized replica administration. You can open any replica, local or remote, whether it's managed by your Replication Manager, another Replication Manager, or not even managed at all, and view the properties and synchronization history.

You can set a scheduled exchange to a replica that isn't being managed. You can create a schedule from one Replication Manager to a remote replica that doesn't have an associated Replication Manager. This particularly benefits locations that don't have staff trained in Replication Manager.

Suppose that you're in London and want to schedule an exchange with a Parisian user who's using a laptop and therefore isn't always connected to the LAN in Paris. You set a schedule from a Synchronizer running in London to a Synchronizer running in Paris. The London Synchronizer, at the appropriate time, extracts the London updates and sends them to Paris. If the laptop's replica is unavailable (maybe the laptop user is on the road), the synchronization updates are stored on the Paris LAN in a special network location called the Paris Synchronizer Dropbox. The laptop user in Paris can subsequently reattach to his LAN and, by using his own Synchronizer, retrieve from his dropbox the synchronization updates sent from London and apply them to his replica.

When the MOD tools are installed with the replication option, the Replication Manager and Synchronizer are installed on your computer. The Synchronizer is optionally placed in the C:\Windows\Start Menu\Programs\Startup folder and starts each time your computer is switched on.

Replication Manager 4.0 can support synchronizations either over a LAN or over the Internet. This first example describes how to set up for any network-type connection (LAN, WAN, RAS, and dial-up). Later in the chapter, setting up an Internet synchronizer is described.

The first time you run Replication Manager, the configuration wizard prompts you to enter a number of parameters for the Synchronizer. First you get a page of introduction (see Figure 22.10):



FIGURE 22.10

Use the *Configure Replication Manager Wizard* to manage your Access replication.

- If Support Indirect Synchronization is selected, you can specify a temporary folder (referred to as a *dropbox*) so that other Synchronizers can send synchronization data, even when this Synchronizer is unavailable (see Figure 22.11). In particular, this is for rarely connected users, such as laptop users, who might be traveling and not connected to the network at all times. When rarely connected users reconnect to the network, their Synchronizers retrieve any synchronization data from their dropboxes.

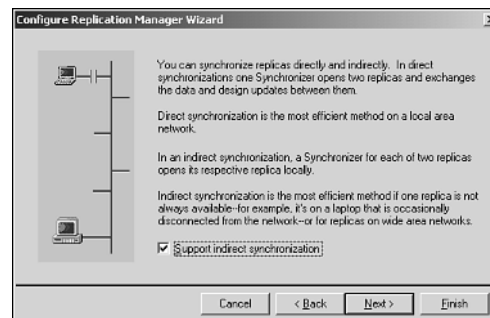


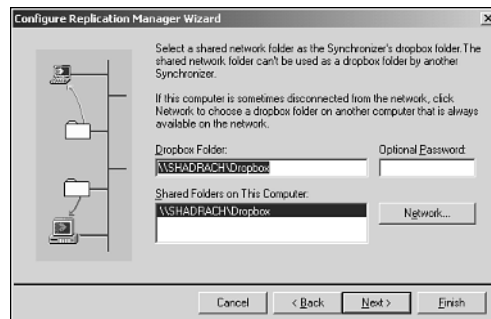
FIGURE 22.11

Select indirect synchronization in the *Configure Replication Manager Wizard*.

TIP

Indirect synchronization support is optional, but when you synchronize over a wide area network (WAN), you can achieve significantly faster, more reliable network response and reduce the number of data packets sent. In contrast to direct synchronization, which opens both members of the replica set involved in the exchange, indirect synchronization relies on a series of message exchanges between replicas and Synchronizers. The Synchronizer managing each replica collects changes into one or many message files (*.msg), which are then sent to a dropbox, which is used by the partner Synchronizer. The partner Synchronizer then processes those message files. Additional message files continue to be sent between Synchronizers until the synchronization is completed.

- For selecting a dropbox folder, you can select a folder shared on the network or your computer as a dropbox for the Synchronizer (see Figure 22.12).

**FIGURE 22.12**

Specify the shared network folder by using the Dropbox Folder selection.

- Is this computer an Internet server? Choose No. It's much easier to set up replication over the LAN first. (Internet replication is covered later in this chapter.)
- When you execute an indirect synchronization by using the JRO Synchronize method or by clicking Synchronize Now in Replication Manager, a “you-pick” synchronization is executed by using the Synchronizer. This means that the Synchronizer will first read the Registry for the order in which it should attempt to execute each type of synchronization. The default order is file system (dropbox), Internet, and then direct synchronization, if the first two methods fail (see Figure 22.13). These attempts don't cause any significant performance hit on synchronizations.

**FIGURE 22.13**

Default order in which to attempt a synchronization.

- Path to Log File gives a file location where the Synchronizer writes a log, which is used for troubleshooting (see Figure 22.14). The system automatically maintains the log file, so no user intervention is required to curtail its size.

**FIGURE 22.14**

Enter the path to where you want the log file written.

- For Synchronizer Name, enter a user-friendly name displayed by Replication Manager (see Figure 22.15).

You can subsequently modify each parameter with the Replication Manager by choosing Configure Microsoft Replication Manager from the Tools menu.

**FIGURE 22.15**

Enter a descriptive name that will help you identify the Synchronizer.

NOTE

Neither the Access menus nor JRO methods let you read or modify Replication Manager or Synchronizer settings. The only exception is that the Retention Period can be modified in JRO.

The Retention Period can be an important tool in keeping database size down, but it's also dangerous because you can make replicas expire.

When the Synchronizer starts running, and every 30 minutes thereafter, it scans its list of managed replicas. Within each managed replica it looks for a synchronization schedule and executes it accordingly. Also, keep in mind that each time the synchronizer starts, it synchronizes all its managed replicas.

NOTE

Scheduled synchronization isn't between replicas, but between Synchronizers. Think of a Synchronizer as a background service, always sending messages to other Synchronizers or checking for new synchronization messages, which are applied to replicas. This is similar to an e-mail system, in which an e-mail background agent is always looking for new mail and placing it in a user's inbox, or sending mail from the user's outbox. This isn't the case when you synchronize from a managed replica to an unmanaged replica or when synchronizing between replicas that the same Synchronizer manages.

The connection between a replica and a Synchronizer is that

- Each replica has one Synchronizer managing it.
- When a replica isn't managed by a Synchronizer, it fakes a "virtual Synchronizer" for itself. This simply is a network address that a real Synchronizer can use as a target for direct synchronizations. Because a virtual Synchronizer is always the passive partner in a synchronization, it can't initiate an exchange or create a schedule.

To initiate a scheduled synchronization, a real Synchronizer must manage either the target or source replicas. You can schedule a synchronization between two replicas managed by the same Synchronizer by setting the Locally Managed Replica Schedule for that Synchronizer. The later section "Scheduled and On-Demand Synchronizations" explains how to set a schedule.

22**WELCOME TO THE
WORLD OF
DATABASE**

For locally managed replicas, one Synchronizer manages more than one replica and initiates a synchronization between the replicas at the appropriate time. This is a direct synchronization.

For replicas at remote locations, the Synchronizer initiating the synchronization opens its local replica and tries to open the remote replica. The Synchronizer then executes a direct exchange between the two replicas. If the remote replica can't be opened (it might be on a laptop and not currently connected to the network), the Synchronizer places the synchronization data in the dropbox specified for the Synchronizer that manages the remote replica.

Synchronization Phases

A synchronization between replicas has a number of phases. These phases occur in sequence and control what and when items are sent between the source and target replicas.

When two replicas exchange, they synchronize schema and data. You can't exchange data but not schema. You can, however, configure your system to exchange schema without data.

TIP

Sometimes you might want separate synchronization schedules for your data and your application. Split your application into a front-end/back-end application, where the front-end database contains the application's forms, reports, and so on, and the back-end database contains the tables and data. Then set a separate schedule for each.

The first phase of the exchange ensures that both replicas have the same schema version. If two replicas aren't in the same schema version, the replica with the highest schema version initially sends the schema updates; only when the schema versions are the same are data updates applied.

If the exchange between two replicas is indirect, a protocol ensures that the two replicas always synchronize their schema before data. Global schema modifications can be made only in the Design Master, which records each modification, incrementing the schema version for each change.

Direct and Indirect Synchronizations

Synchronizations between two replicas can be direct or indirect. In a direct synchronization, both replicas are opened simultaneously, with updates directly read from one replica and immediately written into the other. An indirect synchronization occurs when one of the replicas isn't accessible, as might be the case for a replica on a laptop that's not currently connected to the network. In that case, the replica initiating the synchronization reads its updates and writes them to a previously designated network share. Later, the laptop user connects to the network and reads the network share for any updates to be applied.

Indirect synchronization is available only in the Microsoft Office Developer tools. Once configured, Access or JRO can be used to execute indirect synchronization.

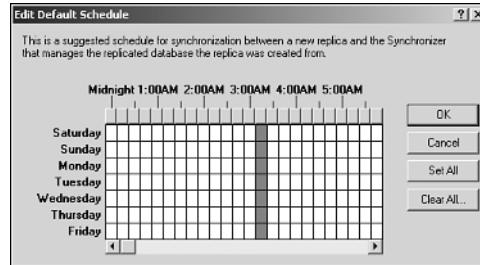
Scheduled and On-Demand Synchronizations

Briefcase and Access's Tools, Replication, and Synchronize Now menus provide users with on-demand synchronizations. You simply indicate that you want to synchronize two replicas, and it happens immediately.

The Replication Manager offers users the choice of setting a schedule of synchronizations between replicas. Replicas can be remote or local. In Replication Manager, you can set a schedule in increments of 15 minutes over a seven-day period simply by right-clicking the line joining two icons representing the location of the two replicas. You can set separate schedules for each remote exchange, as well as set a schedule for exchanges with local replicas on the LAN. Figure 22.16 shows the Edit Default Schedule dialog.

Synchronizing Replicas over the Internet

You can perform replica exchanges via the Internet by using a synchronizer configured for the task. You need Access 200x Replication Manager 4.0 for Internet support, which ships in the Microsoft Office Developer (MOD). Only the Internet server's replica is required to have Replication Manager installed and configured.

**FIGURE 22.16**

To set a schedule, simply click the time and day you want. To select the same time every day of the week, click the appropriate button on the grid's top row.

Internet synchronizations also use a dropbox, like indirect synchronization, but the dropbox on the Internet server is used for all replicas in the replica set. Internet synchronization includes several features, such as the following:

- Support of the HTTP 1.1 protocol eliminates the reliance on FTP for synchronization and enables support through a proxy server.
- The addition of several Registry keys help control Synchronizer timeouts.

Handling Counter Fields

Counter fields are treated specially in replicated Access databases. In non-replicated databases, counter fields are always incremented by one, and are usually the primary key field with a unique index constraint. This is fine with just a single database. In a replicated database, however, two replicas might increment the counter; when the two replicas synchronize, you have a conflict on the unique key constraint.

To circumvent this potential problem, Microsoft modified the behavior of counter fields in replicated databases. So rather than increment by one, counter fields create a random number between 1GB to 4GB (4×10^{12} —actually, -2GB and $+2\text{GB}$). This doesn't absolutely guarantee that a conflict will never occur, but it greatly reduces the risk. All previous counter values are preserved and, of course, any forms or code that relied on the counter field still work.

The problem with randomized counters is that counter fields are often used for, say, OrderID, with the implied meaning that the higher OrderID means a more recent order. This is no longer valid, and you might want to consider using a combination of a date/time value and the counter field instead. For example, the new OrderID for an order entered on 19 October 2001 might be 102001-123456789.

In a large database, you might still encounter conflicts with the randomized counter value. Consider replacing the use of the counter field with the ReplicationID (s_GUID) field. The GUID is a 128-bit hex value, and not pretty for use in applications, but it's always unique.

Using Read-Only Attributes with Replication

In Microsoft Replication Manager, you can choose to make the replica set a single master for data updates. This ensures that inserts and updates can occur only in the Design Master and guarantees consistency between all replicas. If you decide instead to permit updates at more than one replica, you might get data conflicts when two or more sites update the same record simultaneously. Figure 22.17 shows the dialog to enforce the replication read-only constraint.



FIGURE 22.17

Use this dialog to create a read-only replica set in replication.

When creating each replica, you can specifically make it read-only.

Read-only has a special meaning for replicas, and needs some explanation to distinguish it from other uses of the term. A read-only replica can't update or insert any data that's replicated around the replica set. Such a replica can still create local tables and queries and enter local data. Synchronization can update read-only replicas with data from other replicas, as well as modify internal system information.

NOTE

Never place a replica that will participate in synchronization exchanges in a read-only network-shared folder. A replica needs write privileges so that it can record the synchronization history and other internal information.

The only time you might consider placing a replica in a network read-only share is for product distribution. Of course, you still can make a database with the normal read-only permissions by using standard Access/Jet security, and therefore only users with adequate permissions can modify the database.

Performing Replication Identification Fixup

A particularly useful and little-known feature in Access replication is replication identification fixup. After a little background, you will see how this method can be a very powerful for product distribution.

One architectural foundation of Access replication is that each replica is unique and is identified by a unique ReplicaID. This value is stored within each database file.

A potential problem will occur if a user copies a replicable database because this implies that two replicas now have the same ReplicaID (and possibly two Design Masters). Access's solution is that whenever a database is opened, it stores the Unified Naming Convention (UNC), or pathname, to the file. Then, whenever a database is opened, the path used to open the file is compared to the UNC path stored internally; if it's different, the ReplicaID is changed to a new, unique value. If the original database was also the Design Master, it's modified so that it is no longer the Design Master (to prevent two simultaneous Design Masters). This algorithm works whether the file is opened via any combination of UNC network pathname or local pathname, on Windows, or on a Microsoft or Novell network.

CAUTION

Replica identification is particularly important when using the Access Compact utility or making backups. Be careful not to "lose" the Design Master for your replica set. See the following section.

The benefit for product distribution is that a database might simply be copied from a network share, or distributed on floppy disks onto a user's computer. As soon as Access opens the database, it notices that the pathname used to open the file is different from the pathname stored internally, immediately changes the ReplicaID, and stores the new pathname. This is all transparent to users.

Using the Last Synchronization Partner

Name fixup goes even further in making product distribution easy. Each replica has a last synchronization partner used in exchanges. If a user opens a database in Access and chooses Replication and Synchronize Now from the Tools menu, the default replica for the synchronization is the last replica used for a synchronization. However, if a replica has been the subject of a name fixup and given a new ReplicaID, the default partner remains the same.

What this means for third-party application developers is that users need only copy (or use the Microsoft Office Developer's Setup Wizard) to place a replica onto their hard disk; they can then easily synchronize back to the original replica set without any previous knowledge of the origin of the replica. The ReplicaID is automatically and transparently fixed up.

Using the Compact Utility with Replicated Databases

Replication makes extensive use of temporary disk space. However, this space can be recycled and reclaimed by the Compact utility supplied with Access. Simply executing a synchronization can also reclaim disk space.

You're strongly advised to use Compact and Repair regularly with replicated databases.

NOTE

There is an added advantage of using Compact twice in succession on a replicated database. The first pass reclaims some space and then marks other space as available for reclaiming later. A second Compact reclaims all available space. (You don't gain an advantage by running Compact more than twice.)

Deciding Whether to Back Up Your Replicas

This might sound contrary to all other books on databases, but backing up your replicas (in particular, the Design Master) isn't a good idea. If one replica is lost, simply create a new replica from one of the other replicas in the set. If the Design Master database was lost, make one of the other replicas the new Design Master.

Consider what would happen if you restored the Design Master. Assume that you've made a backup. Next, you make some design changes and synchronize with another replica. Now the Design Master somehow gets deleted, so you recover from the backup, which doesn't have the new design changes. The Design Master tries to synchronize with the replica, but now the replica has design changes that aren't in the Design Master.

NOTE

If you restore a backed-up Design Master and synchronize it, Design Master updates its schema correctly only if design changes haven't been made that would conflict with those that previously existed.

Upgrading Replica Sets to Access 200x

Replication introduces a new problem when upgrading to Access 200x because generally not all users will upgrade at exactly the same time. Each replica in a replica set must be individually upgraded. When upgrading, all Microsoft Access objects stored in the Design Master are made replicable. Local Microsoft Access objects stored in each replica, except those in the Design Master, will be lost unless they are first imported into the Design Master before upgrading. Local tables and queries remain local in all replicas of the replica set.

Securing Replicated Applications

Security in replicated databases, with the exception of file passwords, works just the same as non-replicated databases.

You can't replicate the System.mdw security file in Access. Therefore, you must either re-create the exact same user/PID in each System.mdw; or re-create the exact same user group/PID in each System.mdw, assign permissions to the group, and make local users join this group. You also can use Briefcase replication to distribute the latest copy of the .mdw to each user.

You can specify a security file for the Replication Manager by using command-line parameters, in a manner similar to Access:

```
REPLMAN.EXE /WRKGRP path
```

path is the path to System.mdw to be used to open the secured database.

NOTE

You can't use simple file password security with replication. If you have a non-replicated database that already has a file password, you must delete the password before you can make it replicable.

Using MDE Files with Replicated Databases

You can compile your Access application code modules (choose Database Utilities and Make MDE File from the Tools menu), which renames the file extension from .mdb to .mde. This prevents editing or addition to the objects in the database, which is useful when you distribute the application to third parties. A side effect is that it also prevents the merging of replicated objects from the Design Master with local objects at replicas. The net result is that you can't convert a replicated database into an .mde file. To use .mde files with replication, you must convert the file to the MDE format first, and then make it replicable.

Creating Successful Replication Applications

The business application shouldn't encounter many conflicts of data caused by replicas simultaneously updating the same data. Resolving conflicts isn't agreeable to users and will be time-consuming.

Focus on applications in which data is more often inserted, rather than in which existing data is updated. The only exception is where you can guarantee that all updates will occur at a single site, and therefore conflicts between replicas will never occur.

Replication provides data on a "real-time-enough" basis to remote sites. If your business application requires up-to-the-second synchronizations, consider a two-phase commit solution, not Access replication.

Although conflicts should be rare, you can reduce the chances of problems by following a few guidelines:

- If referential integrity is enforced, always ensure that cascading updates and deletes are on. The problem with one replica deleting the primary key while another replica inserts a record with that foreign key will be automatically removed.
- The Microsoft Replication Conflict Viewer can require a certain level of user expertise. When a conflict occurs, users are asked to select which record should win the exchange. If your users can't cope with the Conflict Viewer, write an alternative that automatically makes the correct choice for your application.
- Consider making only a single replica of the site for all updates and all other replicas read-only to prevent any conflicts.

Summary

Replication brings you a totally new range of functionality. These tools allow you to provide solutions to a much wider range of business solutions than was previously possible. Replicated databases provide a new set of issues to be managed when dealing with users over a wide geographical area. Proceed as you would with any new technology. Confirm that your application scales up gracefully to the multisite, multiuser application you're planning.

Access 200x, with database replication, offers a more powerful set of tools than is available in any other desktop database. Use these tools correctly, and you'll be well rewarded! For more information on ADO and security, refer to the following chapters:

- Chapter 5, "Introducing ActiveX Data Objects," explains more about the other objects used in ADO and how to use them in Access 2002.
- Chapter 20, "Securing Your Application," discusses in greater detail how to secure your database.

Moving Workgroup Applications to Client/Server

CHAPTER

23

IN THIS CHAPTER

- Understanding Client/Server 742
- Factoring for Migration to Client/Server 745
- Planning for Client/Server 750
- Knowing What to Watch for in Application Development 753
- Converting Existing Applications 756
- Distributing a Client/Server Solution 772
- Keeping Certain Issues in Mind with Access and SQL Server 776

For many reasons, using Access as a front-end development tool and as back-end storage for data some day might no longer be optimal for your applications. At this point, you should consider a client/server solution. Access gives you the flexibility of storing data in various storage formats, including dBASE, Paradox, FoxPro, Excel, Lotus 1-2-3, text files, and ODBC databases.

This chapter focuses on migrating the data in your Access applications from Jet to SQL Server to create client/server solutions. After reading this chapter, you should have a better understanding of how to plan for client/server migrations and how to carry them out. You also learn a few specifics about using SQL Server's upsizing tools.

NOTE

When SQL Server is mentioned in this chapter without a product version number, it means all versions used with Access 2002 client/server applications: SQL Server 6.5, 7.0, and 2000. Where version-specific behavior is discussed, the text indicates the version number.

Understanding Client/Server

Because of the confusion over what client/server is and its importance in the marketplace, a definition of client/server needs to be provided. You also need to understand what *Open Database Connectivity (ODBC)* is and why Access is a good selection as a front end for client/server applications. Last—and most important—you need to be aware of the factors for client/server migration.

Although many people understand client/server and know how to implement a client/server solution, some people falsely believe that merely having data on the server and the application on the desktop means that they're running a client/server application. In fact, many people confuse their file-server applications for client/server.

By definition, *client/server* means having all application data stored in a back-end server that specializes in query processing and data management, such as SQL Server or Oracle, that's accessed from a front-end client application. In client/server applications, you're actually running two separate programs: one on the client machine and one on the server. The client application manages the interface while the server application manages the data. Client/server requires that both programs communicate with each other for the application to work.

In direct contrast, a file-server application has one program managing the entire application, although the application code and the data might reside in different locations and files.

For example, an Access application that retrieves data stored in an Access .mdb file on your LAN isn't a client/server application—it's a file-server application. In this scenario, the .mdb file on the LAN is merely a container for data; no query processing takes place in it. SQL Server and its data are located on a Windows NT/2000 server. You send the request to the server; the server processes the request and sends back only the results that the application needs.

Working with Open Database Connectivity

Now, how does client/server work? Client and server applications need to have a common language or a translator to communicate with each other. Open Database Connectivity (ODBC) is an open standard that allows communication between applications and various database servers. This has been leading the way for client/server development.

The ODBC drivers available provide a translation layer between front-end applications and their server data storage. For example, the ODBC driver for SQL Server allows Access to communicate seamlessly with Microsoft SQL Server. Drivers are also available that allow an Access application to communicate with non-Microsoft database servers (such as Oracle and Sybase) and with file-server databases (such as FoxPro and dBASE). Figure 23.1 depicts ODBC's architecture.

NOTE

To see which ODBC drivers are now installed on your computer, in Windows 98 use Control Panel's ODBC tool; in Windows 2000, double-click the Control Panel's Administrative Tools icon, and then select Data Sources (ODBC). The ODBC icon has different names depending on the Windows version you are running.

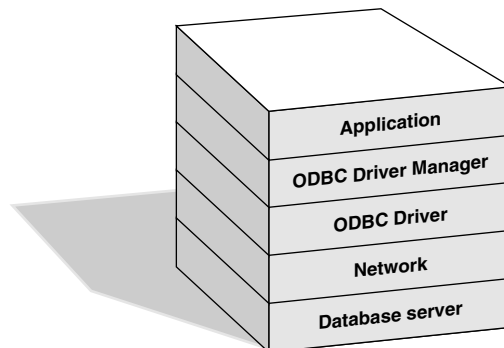


FIGURE 23.1

This diagram depicts the flow of communication between the application and the database server.

The components involved in ODBC are as follows:

- **Application.** This is your front-end application on the client machine. The data request is sent from the application to the ODBC Driver Manager. For example, in client/server development, this would be a front-end application developed in Access.
- **ODBC Driver Manager.** Here, the application's request is translated for the database-specific driver.
- **ODBC Driver.** The ODBC driver sends the request across the network to the back-end database server. For example, if the back-end server is SQL Server, this would be the SQL Server-specific ODBC driver.
- **Network.** The back-end server resides here. The initial request from the application crosses the network to reach the server.
- **Database Server.** This is where your data request is fulfilled. For instance, SQL Server would process the query that originated from Access and send the data back across the network server to the front-end application.

Reasons to Use Access for Client/Server

Access lends itself well to being a suitable front end for client/server applications:

- It provides an easy-to-use tool for scalable development work. With proper foresight, you can easily migrate to client/server applications that use Access for both the client application and data storage.
- Compatibility is another big selling point for Access. It provides ODBC drivers for the most popular database servers and file formats.
- Another issue is cost per user. If you use the Microsoft Office Developer Edition (MOD) tools to create a runtime version of your application, there's no royalty or per-seat charge for end users to use Access. Specifically, the installation disks that the MOD creates include a runtime version of Microsoft Access that's used to run your application. Using the MOD allows an unlimited number of end users to run your application without requiring them to buy a license for Access. Keep in mind, however, that many back-end servers do have a per-seat or connect-time charge.

By using the Project (.adp) file type, Access also provides native-mode connectivity to SQL Server databases. You can design and develop SQL Server databases with the Access user interface and connect Access forms and reports to server data. See Chapter 24, "Developing SQL Server Projects Using ADPs," for more information on Access Projects.

Factoring for Client/Server Migration

You should consider a number of factors when deciding whether to migrate to client/server applications from the file-server world, such as the amount of data that you have, the number of people using the database, and security.

Amount of Data

A common misconception is that database size is the only factor to consider when deciding whether to migrate to client/server. Database size does play a role, certainly, especially when you consider network traffic. Keep in mind that knowing your database will eventually grow to upward of 100MB doesn't necessarily mean that you must move to client/server. However, as your database's complexity increases, the size factor becomes more and more important.

You should ask yourself these questions:

- How much data do I plan for the application to handle?
- Do I anticipate 500 records or 1 million records?

Database structure comes into play as well. There's a huge difference between one table with 1GB of data versus 1,000 tables that comprise that amount. In general, complex queries against multiple tables take much longer to execute than simple queries against one table. Having a 1GB table isn't necessarily cause for migrating to client/server—it might perform adequately in its current file-server environment.

When you initially develop an application, using Access as a back end might make a lot of sense. However, consider initiating a standard set of tests on your application to monitor its performance as the database size grows. For instance, you can search for a random record on an indexed field (of course) and track the response time. Use this to judge performance degradation as your data file expands. Also consider doing searches on various joins and aggregations as additional performance tests.

Listing 23.1 shows the `LogTimes` procedure, which processes a number of queries based on `Northwind.mdb` and records the elapsed runtime in a table. You can modify this procedure to work with your application by replacing the code inside the `For . . Next` loop with code that executes queries against your database. To test `LogTimes` with your copy of `Northwind`, you need to copy `tblLog` and `basLogTime` from `Chap23.mdb` to your copy of `Northwind.mdb`, and then run `LogTimes` from the immediate window. `Chap23.mdb` can be found on this book's Web page at www.sampublishing.com.

LISTING 23.1 Chap23.mdb: Processing Queries and Recording a Table's Runtime

```

Sub LogTimes()
    Dim rstAny As New ADODB.Recordset
    Dim rstLog As New ADODB.Recordset
    Dim lngOrderID As Long
    Dim varStart As Variant
    Dim intCount As Integer

    ' Get start time and seed randomizer
    varStart = Timer
    Randomize

    ' Set iteration count and process queries
    For intCount = 1 To 5
        ' Find random record in order details table by
        ' searching for a OrderID between 10248 and 11077
        lngOrderID = CLng((11077 - 10248 + 1) * Rnd + 10248)
        rstAny.Open "Order Details", CurrentProject.Connection, _
            adOpenDynamic, , adCmdTableDirect
        rstAny.Find "[OrderID] = " & lngOrderID
        rstAny.Close
        ' Run an aggregate query any move to the last record
        rstAny.Open "[Product Sales for 1997]", CurrentProject.Connection, _
            adOpenStatic, , adCmdStoredProc
        rstAny.MoveLast
        rstAny.Close
    Next

    ' Write the elapsed time to the log table
    rstLog.Open "tblLog", CurrentProject.Connection, , adOpenDynamic
    rstLog.AddNew
    rstLog![TestDate] = Now
    rstLog![TestDuration] = Timer - varStart
    rstLog.UPDATE
    rstLog.Close

    Set rstAny = Nothing
    Set rstLog = Nothing
End Sub

```

LogTimes first searches for a random record in a rather large table, Order Details. This way, you get an idea of how row size affects search times on an indexed field. Next, LogTimes executes the Product Sales for 1997 query and uses the MoveLast method to move to the end of the recordset. Product Sales for 1997 is an aggregate query that joins the Orders, Order Details,

Products, and Categories tables. Running this query shows how database size affects table joins and aggregations.

NOTE

LogTimes uses a For . . . Next loop to process each query five times. This filters out any bias for caching.

If you choose to adapt LogTimes for use in your applications, you can see how database size affects performance over time. Your times will be relative, of course, and subject to the hardware the database resides on. The code in LogTimes gives you data for making decisions regarding when a move to a client/server database might make sense.

Use and Purpose of Database

Is your application an *OLTP* (*online transaction processing*) or a *DSS* (*decision support system*)? This can affect your decision on whether to migrate to client/server. For OLTP applications, speed is essential. If phone operators are taking orders, they can't afford to wait five minutes for the database to bring up an inventory checklist.

Database Design

A well-designed database with 1GB of data might run fine with Access as a back end, whereas a poorly designed database with 2MB of data might perform terribly. Refer to sources on database design if you're unsure of what a well-designed database really entails.

Concurrent Use and Number of Users

One of Access's great features is its capability to share an .mdb file on a network drive. Access handles most of the record-locking issues and allows multiple users to access company data. However, having a shared .mdb file for more than 10 concurrent users isn't recommended. For more than 10 simultaneous users, a server database can handle traffic demands much better.

Backup and Recovery

SQL Server and many other server databases can do *live backups*, which means that administrators can back up data without forcing users to first log off. In a file-server application, such as one using Access as the back-end storage format, administrators would have to ensure that everyone first logged out of any application using the server data. You can't back up, repair, or compact an Access database unless everyone is logged out of it. If your application needs to run 24 hours a day, SQL Server is an obvious selection for a back-end database.

SQL Server also logs transactions automatically and can be restored after a system failure. Access has no built-in mechanism for doing this, although it's possible to write code to mimic this behavior. Access simply isn't as sophisticated as a true database server such as SQL Server or Oracle.

Security

Is security of your data an important factor for your organization or customer? Access and SQL Server have built-in security that you can implement, but SQL Server provides greater flexibility. With SQL Server, you can set permissions on tables, views, and even individual columns. Like Access security, SQL Server supports user- and group-level security. SQL Server also provides integrated security with Windows NT/2000, which uses network logon IDs so that users need to log in only once.

Why is SQL Server better?

- In SQL Server, a database is bound tightly to its server, and no direct access to the data is available. An Access .mdb file, on the other hand, always has the potential of being deleted, moved, or copied off the network. You can choose to use Access to test security for your client/server application and then re-create the security on SQL Server.
- SQL Server security, although flexible, is straightforward and data-centric. Access security, in contrast, must handle the complex security needs of the myriad Access objects (tables, queries, forms, reports, and code), and their relationships to each other and to application users.

Data Sharing Among Applications

Another factor that warrants consideration is whether your data needs to be shared across various applications. By placing your data in an Access .mdb file, you're limiting its usefulness to those applications that use Jet or ODBC. SQL Server, on the other hand, is more easily accessible by a number of DOS and Windows applications by using the DBLibrary set of function calls. Obviously, if you now have DBLibrary applications accessing SQL Server and want to share data, your Access applications need to use SQL Server as the back end.

Network Traffic

Switching to a well-implemented client/server solution decreases the amount of network traffic because processing is carried out on the server and only results are sent back across the network lines. If Access is the back end, it will send the much larger set of data across the network to be processed at the client machine.

Consider the query `SELECT * FROM customers WHERE city = 'Richmond'`. Now assume that 10,000 records are in the table and only five records are in the result set. This query executes much more quickly against a SQL Server database than an Access database located on the same network server. Why? Access databases don't process the query; they merely store the data in the .mdb format. In the case of an Access database, the entire recordset—yes, all 10,000 records—would be brought back across the network to your application to execute the query if the city column wasn't indexed. With SQL Server, the query would be processed on the back-end server, and then the result set of five records would cross the network to bring the answer to your application.

If your data request is based on indexed fields, the amount of data transmitted from back end to front will be considerably less in the 10,000-record example because only the index for this table travels across the network. Access then uses the index information to create a request that will retrieve only the result set across the network. However, if your query is based on non-indexed fields, you're putting an incredible burden on your network.

Record Aggregation

If your application uses record aggregation (grouping, summing, and other operations that use the SQL `GROUP BY` clause), the performance differential between the file-server and client/server models becomes quite large.

Consider a query that groups data by one field and sums the values in a second field. A file-server Access application would need to retrieve every value needed for the sum (every record indicated by the SQL `Where` statement). Compare this scenario to a server-based model where the server processes data through its query engine or by using code placed in a stored procedure. The server aggregates or alters the data locally before sending only the results to the front-end application; the detail records used in the computation never leave the server. This approach yields far better performance than the file-server model.

Bet Your Career on Choosing the Right System

If you're in charge of selecting a server database for a mission-critical application, would you prefer using a robust, fault-tolerant system? If you had selected a shared Access .mdb file and one user crashed his computer, it could mean the corruption of the data, and all other users would be locked out of the system.

Using Access as a front end and back end doesn't protect you from potential data loss as SQL Server does. Remember, SQL Server logs transactions to enable recoveries and allows live backups; Access doesn't.

The old adage, "An ounce of prevention is worth a pound of cure," applies to this scenario. Yes, it's more complex and likely more time-consuming to set up a SQL Server back-end

database than to use a standard Access .mdb file, but you and your customer will be in a much better position in case of failures. If reliability, security, and safety are of the utmost importance, choose SQL Server.

Planning for Client/Server

Planning for client/server applications as early in the development process as possible is important. Your initial decision of data structure and your approach to application development affect your entire migration process. Even before considering the migration path to client/server, you need to think about your data storage needs and how they will affect your network. Can your network handle the traffic demands that your application might have?

The following sections cover many server-specific rules that could trip up your client/server applications, as well as various development issues that you need to be aware of.

Microsoft wants you to believe that migrating Access file-server applications to client/server using SQL Server is easy. Well, not quite. True, it might be easy if the process is well planned from the beginning, but too many times this doesn't happen. The following sections stress many items to keep in mind as you develop your Access applications. It's always possible that someday your applications will be migrated to client/server. Even if you think it's not probable, why eliminate the possibility?

If you have an Access application, it will run perfectly when your tables are upsized to a SQL Server database—right? Unfortunately, this isn't the case. You need to be aware of and take into consideration a number of important server-specific rules as you plan your applications. SQL Server has a number of differences from Access, including different limitations on field and table names, an increased number of reserved words, case sensitivity, and how it processes query requests.

As you plan for upsizing, you must consider which upsizing option you will use. Access lets you upsize an application in two ways:

- **Upsize the tables.** If you choose to upsize only the data tables to SQL Server with the Upsizing Wizard, each table in your application is moved to the server and linked back to your application. This is the only upsizing behavior available in Access versions before Office 2000. This option creates an application that's not truly client/server; instead, the application follows a high-end file-server model with SQL Server as the back end. This chapter refers to this structure as a *SQL-linked application*.
- **Create a Project.** The Upsizing Wizard can convert your application into an Access Project, where no Jet components exist and all tables and queries in your application become tables, views, and stored procedures on the SQL server. A Project provides a true two-tier client/server application where the server is used for all data processing of records returned to the client.

Field and Table Names

Isn't it great that Access allows spaces in field and table names and lets you use an unwieldy number of characters to name objects? Well, it's not so wonderful after you venture into the world of client/server development. You need to have the foresight to prepare for the possibility of migration to SQL Server or another back-end database.

SQL Server 6.5 didn't allow field and table names longer than 30 characters, so limit your object names to this length if you plan to upsize to SQL Server 6.5. SQL Server 7.0/2000 supports the same 64-character object name limit allowed in Access.

If you want to use a different back-end database in the future, adhere to that back end's name length limit. For example, some back-end databases have a 10-character maximum field name length. Always consider the lowest common denominator among back-end database restrictions when naming objects.

Next is the issue of embedded spaces in field names or table names. SQL Server didn't allow this until version 7.0. If you're upsizing to a SQL Server 6.5 database, the Upsizing Wizard will change spaces to underscores (_) in your object names, which might affect how your code runs or application objects behave. If you expect to upsize your application to SQL Server 6.5 or any other back end that doesn't support spaces, don't use them in your object names.

Reserved Words

SQL Server has even more reserved words to beware of than Access does. In your first conversion from Access to SQL Server, you might not understand why the Upsizing Tools renames one of your fields from DESC to DESC_. You'll later realize that although Access accepted this field's name, DESC is a reserved word in SQL Server to indicate descending sort order. That's life. In any case, this is something to consider when beginning development of your Access database.

NOTE

Although Access is sometimes forgiving when you use certain reserved words in an application, it's a good policy not to use reserved words as object names or other identifiers in Access or SQL Server databases or in VBA program code. A *reserved word* is the name of an object in the Access, VBA, or SQL Server object models (as shown in the Object Browser window) or a syntax element from the Visual Basic programming language. See the Access help topics "Scoping and Object-Naming Compatibility" and "Guidelines for Naming Visual Basic Procedures, Variables, and Constants" for more information.

Case Sensitivity

By default, SQL Server is installed with a sort order of Dictionary Order, Case Insensitive. This option means that the characters *a* and *A* are equivalent for the purpose of sorting and comparisons, which is the same behavior found in Access using Jet. The sort order determines not only how SQL Server sorts query results, but also how to process queries that involve character comparisons.

CAUTION

If your installation of SQL Server is installed as Case Sensitive, pay particular attention to all your queries and application code. This is guaranteed to cause headaches for developers and end users alike! With a case-sensitive installation, the value *customer* doesn't equal the value *Customer* in a WHERE comparison. Any queries or code that refer to the *customer* field without the proper capitalization will fail. Be sure to check with your SQL Server system administrator about this issue before you upsize.

Query Processing on the Server

To take full advantage of your back-end server, you should do as much processing on the server as possible. Remember, SQL Server specializes in processing large amounts of data. In most cases, processing queries in SQL Server is faster than processing queries on client applications.

Moreover, if the back-end server can't process your request for information, it sends a large amount of data back to Access across the network and lets Access come up with an answer. Ensuring that you're sending requests that the server can understand—through ODBC or SQL Pass-Through queries—is an important step that shouldn't be taken lightly.

Fortunately, if you have an extremely complex query that SQL Server can't fully understand, Access still sends limited portions of the query that SQL Server can deal with. A good example is an Access crosstab query. SQL Server doesn't understand these queries, but can still do the initial select query and return the results to Access. Access takes those query results and completes the crosstab query.

Sometimes you might not know where a query is actually processed. You can use some third-party tools to monitor the communication to and from the server to determine whether the server sent back the full result set, partial result set, or nothing at all.

You also can enable ODBC tracing to log all ODBC calls to a text file. This log file can provide invaluable information; you really know what ODBC calls are being made behind the

scenes, which might help you determine why your application isn't performing as you would expect. To enable ODBC tracing, follow these steps:

1. Double-click the ODBC Data Sources icon in the Windows Control Panel (note that this icon has a different name in each of the most recent Windows versions—95, 98, and NT). For Windows 2000, you have to choose Administrative Tools from the Start menu, and then select Data Sources (ODBC).
2. On the Tracing page (see Figure 23.2), select a different log file, if desired. By default, ODBC calls are logged to `SQL.LOG`.
3. Click the Start Tracing Now button.

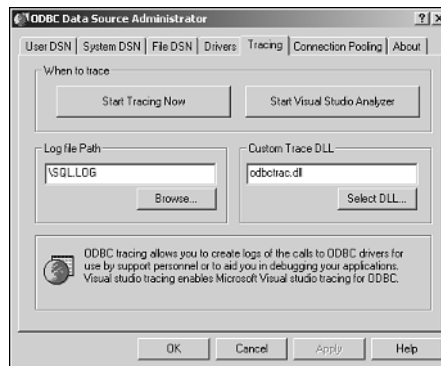


FIGURE 23.2

Turn on ODBC tracing with the ODBC Data Source Administrator.

Knowing What to Watch for in Application Development

When developing client/server applications, you need to watch out for many things if you plan to migrate to client/server. These include—but aren't limited to—limiting your data, using combo boxes, using Access-specific and user-defined functions, creating heterogeneous and cross-database joins, dealing with ActiveX objects, and using local Access tables.

Limiting Your Data

In client/server development, it's very important to limit the amount of data being sent back and forth across the network and to retrieve only the columns and rows you need. If you read nothing else in this section, leave it with the message that limiting your data is crucial.

Moreover, when designing forms, the goal is to limit the amount of data that initially appears. Ideally, you could create a form that doesn't require a server call at all. Take the approach of retrieving data as needed. Why spend 30 seconds retrieving an entire data set when the user needs to work with only one of those records? It's more efficient to retrieve only what you need as you need it, in many cases. For example, you could create a customer form that's bound to a table and let users navigate through the form's records to find the desired customer record. When this form opens, it pulls down large amounts of data from the server. Alternatively, you could create a form that lists all customer names, let users select a single customer from the list, and then open the customer form with a `WHERE` clause that pulls in the single desired customer record from the server.

TIP

This example applies not only to data records, but also to fields. Avoid selecting all fields from any given table when at all possible. If you don't need all the fields, don't drag them across your network.

If you still use DAO, also consider whether to use a dynaset or snapshot recordset object. Keep in mind how these recordset objects vary. A dynaset sends two queries to the back-end server: one to return enough data to fill the first visible page and a second to return the primary key values for the remaining records. A snapshot, on the other hand, returns the entire data set in read-only mode. In larger data sets, both objects can put an unnecessary burden on your network. However, for small- to medium-sized query result sets, a dynaset generally gives a better response time because it continues background fetching of the keys only as you peruse the first page of data.

Using Combo Boxes

Combo boxes are a great feature in Access, but a word of caution is necessary. As your data set grows, the performance degradation you'll experience can soon outweigh the usefulness of the combo boxes. A combo box that presents 25 items makes sense, whereas a listing of 1,000 items is slow, and it's unreasonable to expect the end user to stumble through such an unmanageable list. Further, combo boxes on forms might keep ODBC connections open to the server, placing an additional burden on network and server resources. When migrating to client/server, consider whether it makes sense to continue using combo boxes in every case.

As alternatives, consider using pop-up forms or providing a means of filtering your combo boxes to make selection easier. If the value list in the combo box is static, consider using VBA code to fetch data items for the combo box from the server at form load time and using the data to create a static string for the combo box's `RowSource` property.

Using Access-Specific and User-Defined Functions

Access provides great flexibility in query development, such as the capability to include user-defined functions, immediate IF statements (IIF), and the TRANSFORM statement for crosstab queries. Unfortunately, SQL Server doesn't understand these Access-specific syntaxes. If such queries exist in an application upsized to use SQL table links, Access processes the query locally after the server sends the entire data set back across the network. If such queries exist in an application upsized to an Access Project, the Upsizing Wizard generally doesn't convert them. Be sure to remove any function types or Jet-specific syntax from your queries before upsizing. See the later section "Cleaning Up Queries" for more information.

NOTE

For an application that will be upsized to use SQL table links, avoid DAO code that operates on table-type recordsets, such as the `OpenTable` method (pre-Access 2000), the `OpenRecordset` method of type `dbOpenTable`, and the `Seek` operation because you can't use these with tables linked from SQL Server. For an application that will be upsized into an Access Project, use only ADO code in your modules, not DAO (see the help topic "Converting DAO Code to ADO"). DAO code won't run after it's upsized into a Project.

Creating Heterogeneous and Cross-Database Joins

SQL Server 6.5 can handle only local joins; heterogeneous joins and cross-database joins aren't supported. A *heterogeneous join* joins two different database formats, such as Access and SQL Server; a *cross-database join* joins two tables in the same format but in different locations. While developing applications that will be upsized to SQL Server 6.5, try to keep data that will be joined in queries in the same format and in the same database to avoid the limitations of heterogeneous and cross-database joins.

As of version 7.0, SQL Server is freed from such limitations through a concept called a *linked server*, which is a defined data source known to the SQL Server database that points to another database server. You can refer to linked servers in SQL statements and views in Access Projects, providing for cross-database joins that might or might not also be heterogeneous.

Dealing with OLE Objects

Just as they are in Jet tables, memo fields and OLE objects such as bitmap pictures or video clips can be a tremendous burden on SQL Server query execution. Rather than display these field types automatically, exclude them from SQL statements and queries/views unless they're needed. This will minimize network traffic in a file-server or client/server application.

TIP

To minimize network traffic, base your form on a query that selects only non-OLE and non-memo fields. Then create a second query to retrieve as needed your OLE objects and memo fields that can be launched via a command button or menu selection.

Using Local Tables for Static Information

In cases where you upsize an application to use linked SQL Server tables, you might be able to increase your client/server application's performance dramatically by loading some data into local temporary tables during the application startup routine. Suppose that you have 100+ fairly static categories for the banking transactions in your application. If these codes populate a combo box on a number of your forms, it makes sense to store these values locally. Be sure also to create an index on the local temporary table.

If you decide to store locally data that can change during the course of the day, consider providing users with a way to refresh these local temporary tables.

To use local tables, first create the table structure in Access. Then write a function that queries your SQL Server database when your application starts up. This function should replace existing table values with the current values from SQL Server. (For more details about performing this task, see the example in Chapter 26, "Creating Maintenance Routines.")

This concept must be implemented differently in an application upsized to an Access Project because a Project can't contain local (Jet) tables. One approach is to pull the required temporary values from the SQL server into a local VBA array and use the array to populate form controls as needed. Other approaches include using ADO code to store data locally in an .mdb file that supports the Project but isn't physically linked to it, or persisting the recordset to XML.

Converting Existing Applications

After you decide to migrate an existing file-server application to client/server, review your data structure to ensure that you have a well-designed, properly normalized database design. Then you need to prepare any existing queries, forms, and reports for the migration.

Starting with a Well-Designed Database

When you're ready to convert your application to SQL Server, first step back and analyze your data model. Is it properly normalized? Are any changes needed before the migration?

NOTE

You should make changes to the Access database rather than assume that it's easy to make changes when the data structure exists in SQL Server. Changing SQL Server database structures can be very complicated thanks to the presence of referential integrity, triggers, and stored procedures that depend on the data structure. However, the SQL Server development tools integrated into Access (as of Office 2000) provide far more extensive database design capabilities than were available in previous Access releases.

Using Timestamp Fields

SQL Server and many other back-end servers support a timestamp data field. This field serves as a version column that the Access Jet database engine can use to determine whether a record has changed in a linked table. The absence of a timestamp necessitates a column-by-column comparison, which is less efficient than the alternative and causes unnecessary network traffic. Depending on the options you choose while upsizing, the Upsizing Wizard adds timestamp fields to SQL Server tables as needed.

Cleaning Up Queries

As mentioned earlier in the section “Using Access-Specific and User-Defined Functions,” you need to eliminate any Access or VBA specifics from your queries if you want them to run on SQL Server. But if you really need to run that user-defined function on your data set and don't want to remove it from your query, move the function down to the form level.

For instance, you might have a user-defined function called `YourCommission()` that calculates a complex commission amount for your sales force. You should eliminate this from the query and put the function on the appropriate form control. This is much more efficient; you still take advantage of the server data processing speed and then run the function in your front-end application.

The same holds true for reports. Remove any Access specificity from the query and move it down to the report level.

Unfortunately, even after removing all Access-specific functions, a query might not run on linked SQL Server tables and might lead to an ODBC call failed message. It's a good idea to test each query against the back-end server because it's possible that the query is still too complex for SQL Server. For instance, you might encounter a `GROUP BY` limitation of 255 characters. The easy solution here is to base some queries on queries that limit the `GROUP BY` clause.

NOTE

As the number of tables included in your query grows, it becomes harder for SQL Server to choose how to optimize the query. SQL Server optimizes up to four tables at a time. When designing your queries, keep in mind that the more complex the query, the less SQL Server will fully optimize it.

Here are a few guidelines for query construction:

- **Re-create indexes.** Make sure that all indexes are re-created on SQL Server for use by your queries. Any column used in a query JOIN should be indexed, as should any column used for searches and in query WHERE conditions.
- **Use equi-joins.** These result in best performance if unique indexes are available.
- **Avoid SQL keywords.** If you're upsizing an application to an Access Project, queries are moved to server-side views and can't contain Jet syntax not known to SQL Server, including DISTINCTROW, PARAMETERS, and TRANSFORM.
- **Avoid disjunctions and hard-to-optimize constructs.** Using OR and IN constructs generally result in poor performance. NOT, <>, and != aren't useful to optimize for index selection.

In addition to making all queries understandable by SQL Server, review your application's queries and make sure that all queries are efficient.

NOTE

As of SQL Server 2000, extended properties such as validation rules are now supported and created when upsizing with the Upsizing Wizard.

Reworking Forms

Your form design is one area where you'll likely need to do the greatest rework in your conversion of file-server to client/server. Access makes it easy to build grandiose forms and sub-forms that are bound to your application's data. It's so easy: simply make a change on the form, and the underlying table is modified, too. The drawback of this approach is clear: The entire data set needs to travel across the network to populate the form. As your data set grows, this approach becomes less efficient and less practical, especially when you start talking about SQL-linked applications.

A better way to design forms is to severely limit the amount of data bound to the form. One approach is to provide users with a means of filtering the data, such as deciding what authors they want to view by last name. Figure 23.3 shows a series of toggle buttons that users can choose from. The selected letter of the author's last name will be used to restrict the data set.



FIGURE 23.3

frmAuthors in Chap23.mdb helps users limit their data.

This requires some VBA code behind the option group's AfterUpdate event (see Listing 23.2). The code sets the record source based on your response and then unhides the detail and footer sections where you display the author information. To test this, you need to attach to the authors table in the PUBS database that comes with SQL Server. This code can be found in Chap23.mdb on this book's Web page at www.samspublishing.com.

LISTING 23.2 Chap23.mdb: The AfterUpdate Subroutine for the Letters Option Group

```
Sub grpLetters_AfterUpdate()
    'Modify record source
    Me.Recordsource= "SELECT * FROM dbo_authors WHERE au_lname " & _
        LIKE '" & Chr$(Me![grpLetters]) & "';"

    'Display selection
    If Not Me.Section(acDetail).Visible Then
        Me.Section(acDetail).Visible = True    'Show Detail section
        Me.Section(acFooter).Visible = True    'Show Footer section
        Me.NavigationButtons = True            'Show navigation buttons
        'Fit windows to form
        Docmd.RunCommand acCmdSizeToFitForm
    End If
End Sub
```

You should apply this same idea to your combo boxes to limit selection. To accomplish this, replace your combo boxes with pop-up forms that provide a means of limiting data.

Developing Advanced Applications

After upsizing your data to SQL Server (see “Upsizing Access Databases” later in this chapter), you can take advantage of advanced SQL Server features to further improve the performance of your application. These include SQL Pass-Through queries, server views, and stored procedures.

Using SQL Pass-Through

The term *SQL Pass-Through* literally means that SQL statements are passed directly through to the server. SQL Pass-Through queries bypass any ODBC and Jet manipulation. They provide a good option for Access-to-SQL Server client/server development when using a SQL-linked application; because saved queries aren’t available in an Access Project, use stored procedures or VBA code against ADO to achieve the same result. SQL Pass-Through queries execute more quickly than queries against linked tables and are the means of tapping into SQL Server’s stored procedures mentioned later in this chapter. SQL Pass-Through statements can be directed at any ODBC data source and aren’t limited to SQL Server.

SQL Pass-Through queries generally are easier to debug because they’re passed to the server unaltered. When queries are processed through ODBC, you need to consult some tracing mechanism to know what message is really passed to SQL Server.

Some costs are associated with using SQL Pass-Through. SQL Pass-Through queries are always non-updateable, but you can program updates to the underlying tables via VBA if necessary.

To create a SQL Pass-Through query, follow these steps:

1. Choose Query from the Insert menu, or create a new query from the Database window.
2. In the New Query dialog, select Design View and click OK.
3. Select the tables and/or queries to be used in the SQL Pass-Through query, and then click Add.
4. Add any fields and criteria you need in the query.
5. From the Query menu, choose SQL Specific and then Pass-Through. The SQL window displays the SQL Pass-Through query (see Figure 23.4).



FIGURE 23.4

The *SQL Pass-Through Query* window shows the *SQL* that will be sent to the back-end server.

6. Modify the SQL statement as needed. If you want, try running the query to see whether it runs properly on SQL Server.
7. Save the query.

Using Views

In Microsoft Access, developers often use a query as a form's record source to limit the amount of data presented and to enforce security. Queries set up as *RWOP (Run With Owner's Permission)* allow end users to modify underlying tables with the owner's permissions, even if they have no permission on the underlying tables.

In SQL Server, you can consider a *view* as an Access query. Views, like queries, provide security by controlling what fields and records end users can see. For client/server applications, you need to pay particular attention to how much data travels across the network. You also need to remember that views aren't updateable by default, unlike their Access counterparts; you must include all fields that comprise the primary key if you want to make the view updateable.

To create a view in SQL Server, you have three options:

- Use the `CREATE VIEW` statement.
- Use the View Designer in an Access Project connected to a SQL Server database.
- Use the Manage Views window in the SQL Server Enterprise Manager. (Choose the database that you want to manage views for from the database objects list. Right-click Views, choose New View from the shortcut menu, and enter a view statement in the View Designer.)

To create a basic SQL Server view by using SQL syntax, use the following code:

```
CREATE VIEW all_titles
AS
SELECT title, type, price
FROM titles
```

After the statement is executed, SQL Server creates the view. You can create more complex views to restrict which rows are returned. The following statement creates a view that shows only books priced over \$20:

```
CREATE VIEW expensive_titles
AS
SELECT title, type, price
from TITLES
WHERE price > $20
```

You can actually link SQL Server views in your SQL-linked application and use them as though they were tables. If you want to be able to update the underlying table, you need to

create an index for the view. If you attach a view in Access, you're prompted to select the unique key for index creation. You can create other indexes as needed. To create a local index called `CustID` for a server view where the unique key is the field `CustomerID` for the `Customers` table, follow these steps:

1. Choose Query from the Insert menu, or create a new query from the Database window.
2. In the New Query dialog, select Design View and click OK.
3. Close the Add Table dialog—it's not needed to create a data definition language (DDL) query.
4. Choose SQL Specific from the Query menu, and then choose Data Definition.
5. Type **CREATE UNIQUE INDEX CustID index ON Customers (CustomerID)** in the Data Definition Query dialog.
6. Run the query.

In an Access Project, views are manipulated directly from the Database window, so you don't need to use SQL syntax to create, alter, or delete views in a Project. In fact, if you need to create server views to be used by a SQL-linked application (.mdb file) as described earlier, you can create an Access Project connected to the target SQL Server database, and use the design tools in the .adp (Project) file to help with your database development.

Using Stored Procedures

As in Access, SQL Server has its own programming language, Transact-SQL, that's far more powerful than simple query execution. This control-of-flow language allows you to program looping constructs, conditional statements, variable usage, and much more.

To optimize performance for a client/server application that uses SQL Server, consider using stored procedures. As a general rule of thumb, any process that your application runs frequently is a good candidate for a stored procedure.

A *stored procedure* is a precompiled set of SQL statements and control-of-flow language that greatly improves application performance. One type of stored procedure is a trigger, which executes when a certain event happens, such as an insertion, deletion, or data modification. Another type is a record-returning stored procedure, which returns one or more data recordsets to the calling application in much the same way as a server view.

SQL Server can run stored procedures without involving the client machine. When your data is stored in Access, on the other hand, all processes must be completed by the front-end application.

The following sample code works with the pubs sample database and shows the `myprocedure` stored procedure, which accepts two input parameters: `@lname` and `@fname`, the author's first

and last name. By default, if you run myprocedure, you'll get a listing of all authors and their books who have a first name beginning with *A* and a last name beginning with *R*.

```
CREATE PROCEDURE myprocedure @lname varchar(35)='R%', @fname varchar(15)='A%'
AS SELECT au_lname, au_fname, title
FROM authors, titleauthor, titles
WHERE au_fname LIKE fname
AND au_lname LIKE lname
AND authors.au_id = titleauthor.au_id
AND titles.title_id = titleauthor.title_id
```

In an Access application, stored procedures are created, modified, deleted, or executed by using SQL Pass-Through queries or VBA code. In an Access Project, stored procedures are manipulated directly from the Database window by using a stored procedure editor. In both cases, the stored procedure syntax is the same, but the editor and the model for binding to the database are slightly different.

Using In-Line Functions

In SQL Server 2000, you now can create and use user-defined functions, also called *in-line functions*. These queries take input parameters and return results, similar to stored procedures. To find out more about this new feature, see Chapter 24, “Developing SQL Server Projects Using ADPs.”

Working with Current Access Security

If you've implemented security in your applications, it requires some conversion. Yes, your Access security might work, but it protects your data only if users access the data from your Access front-end application. When your data is on the server, you need to protect it on the server for the same reason you secure data in your Access data database and not the end-user application.

SQL Server security was discussed earlier, in the section titled “Security.” Although it takes some effort, you can re-create some or all of your Access data security on SQL Server. You might consider writing a function, relying heavily on ADO, that automatically reads in your current Access users and re-creates them on the server.

Groups, however, are a different issue. In Access, you can belong to as many groups as you want, but in SQL Server you inherit rights from your user account and the one or more roles assigned to it. A SQL Server logon maps to a specified user account, which gains you certain permissions. Permissions can then be grouped into roles, and you can be assigned multiple roles. This provides a model for fine-tuning database permissions, but because this model varies from Access's approach, it's almost assured that you will make some changes to your Access application's security after you upsize it.

Upsizing Access Databases

In general, when developers talk about upsizing databases, they're referring to moving data to a more robust and efficient back-end server, such as SQL Server. In Access, after you move a database to SQL Server, you can connect to the data through a SQL-linked application or a Project (see the earlier section "Planning for Client/Server").

These basic steps are involved when moving to an Access application and are explained in more detail in subsequent sections:

1. Perform the preparatory steps.
2. Set up the ODBC data source.
3. Export the tables.
4. Rebuild the indexes and relationships.
5. Manually apply the defaults, rules, and triggers (referential integrity) with SQL or by using the SQL Server Enterprise Manager or Access Project UI.
6. Create attachments (table links) to the SQL data in the Access databases and delete the original local tables.

Preparing to Upsize

Before you can upsize to SQL Server, you must have appropriate server permissions. To upsize into an existing database, you need only `CREATE TABLE` and `CREATE DEFAULT` permissions for that database. If you are creating a new database, you also need `CREATE DATABASE` rights on the server, as well as `SELECT` permissions on system tables in the master database. You might need to have your SQL Server administrator set up a new device and database for your application tables. You also need the appropriate permissions in Access to proceed. For any object that you will upsize, you need read/design permission on that object.

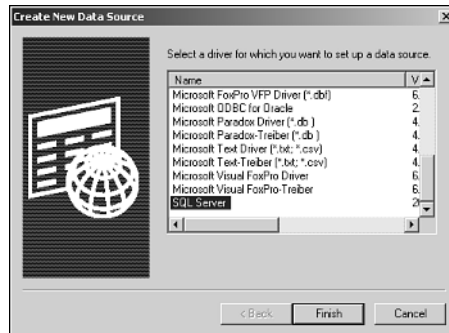
Finally, because the Upsizing Wizard saves its status report as an Access report snapshot, you must have a default printer installed and specified on the computer used to upsize.

Setting Up the ODBC Data Source

Before beginning the upsizing process, you need to ensure that ODBC is installed properly and then set up a data source for SQL Server. (For some applications, you might want to set up the data source automatically for end users; see the later section "Programmatically Setting Up an ODBC Data Source.") To set up the data source, follow these steps:

1. Double-click Control Panel's ODBC Data Sources (32bit) icon (Windows 9x) or ODBC icon (Windows NT), or choose Administration Tools, Data Sources (ODBC) from the Start menu in Windows 2000.

2. Check to see whether a data source is set up for the SQL Server you want to upsize to. If it is, you're all set for upsizing. If not, proceed with step 3.
3. Click the Add button.
4. In the Create New Data Source dialog, select SQL Server (see Figure 23.5) and click Finish.

**FIGURE 23.5**

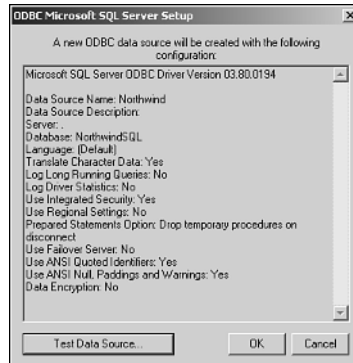
Use this dialog to create a new ODBC data source.

5. In the ODBC SQL Server Setup dialog, enter a Data Source Name to identify the data source you'll use when connecting to SQL Server tables.
6. Enter a description in the Description text box.
7. Select a server name from the list or enter the physical name of the server where your SQL Server database resides. Click Next.
8. Enter the name of the SQL Server database in the Database Name text box. This database is where your tables will be created when you go through the upsizing process.
9. Click Finish. The ODBC Microsoft SQL Server Setup dialog appears (see Figure 23.6).
10. Click OK.

Exporting Tables

Next, you re-create your Access tables on the back-end server. First determine whether you can upsize to an existing database or need to create a new database on your server. If this is the first time you've upsized, you need to create a new database. Consult your SQL Server documentation for more information.

Identify which tables to upsize. You might decide to have some tables remain locally rather than on the back-end server, or you might want to have some temporary local tables populated periodically by SQL Server. See the earlier section "Using Local Tables for Static Information" for details.

**FIGURE 23.6**

The ODBC Microsoft SQL Server Setup dialog shows how your data source is defined.

Access 2002 exports tables directly to ODBC data sources. This method is generally slow with large data tables, but ensures that your data adheres to all the server-based rules. Using bulk-copy routines is a significantly faster option for large data sets, but it doesn't bother with verifying your data.

TIP

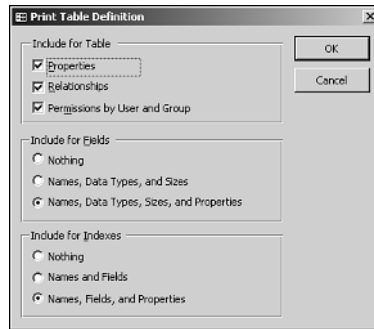
As of Access 2000, you can export AutoNumber-type fields, and they become Identity columns used by SQL Server.

Rebuilding Indexes and Relationships

Because indexing is critical to performance, you must also re-create any necessary indexes for your database. Be sure to index on all primary keys, foreign keys, and any field you believe will be used frequently for searches. Also, rebuild relationships between data tables as necessary. In SQL Server, this also includes creating triggers, as discussed in the next section.

If you aren't sure what indexes, defaults, and referential integrity rules exist on your Access database, use the Database Documenter to get a complete listing:

1. Choose Analyze from the Tools menu, and then choose Documenter.
2. Select the tables and other objects you want information on.
3. Click the Options button for additional table documentation settings.
4. The Print Table Definition dialog opens. Select what information you want to include for tables, fields, and indexes (see Figure 23.7), and then click OK.

**FIGURE 23.7**

The Print Table Definition dialog lets you choose what information you want documented.

Re-creating Defaults, Rules, and Triggers

You also need to manually set up any defaults, validation rules, and referential integrity on the back-end server. Fortunately with SQL Server, you can set up referential integrity via the user interface rather than write trigger upon trigger. For more information on creating SQL Server objects, see your SQL Server documentation.

Attaching to Server Tables

After your server tables are set up properly, you need to create a link to each table in your Access application and delete the original tables, if applicable, after transferring any data over to the server tables:

1. Choose Get External Data from the File menu, and then choose Link Tables.
2. Set the Files of Type selection to ODBC Databases.
3. In the Select Data Source dialog, select your SQL data source and click OK.
4. Enter the username and password in the Login dialog and click OK.
5. Select the tables to attach in the Link Tables dialog and click OK.

You can choose to create some alias queries for all upsized tables to ensure that your SQL-linked application continues working. For instance, your old tables might include fields with embedded spaces, such as Customer Name. Because SQL Server doesn't allow that, you likely created a Customer_Name field on your back-end server. You can choose to change any query that referenced these fields, or simply create an alias query that's used in all queries and acts as the base table. An *alias query* is an actual query saved under the name of the original table but referencing the newly created tables. In this alias query, columns would be aliased, as in Customer Name: Customer_Name.

Using the Upsizing Wizard

In my opinion, one of the greatest time-saving products Microsoft ever developed is the Upsizing Wizard. When your application is ready for the client/server world, upsizing your data table to SQL Server is quite simple.

The Upsizing Wizard is an alternative to manually upsizing a database. If you've ever upsized a database manually, you'll certainly appreciate these tools. The steps in this section upsize the Northwind.mdb application found in your \Program Files\Microsoft Office\Office\Samples folder.

Next, create a copy of the Northwind database to use for this experiment, name it NorthwindSQL.mdb, and place it in the same folder as the original.

Now you can put the upsizing tools to use:

1. Open the copy of the database you made, named NorthwindSQL.mdb.
2. From the Tools menu choose Database Utilities and then Upsizing Wizard to launch the Upsizing Wizard (see Figure 23.8).

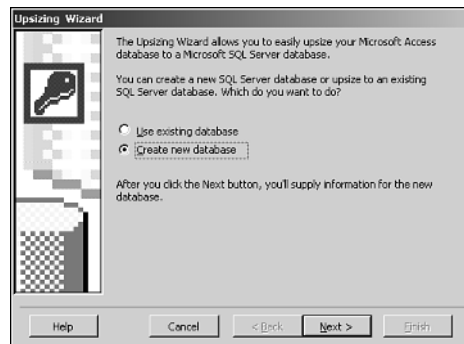


FIGURE 23.8

The first time you use the Upsizing Wizard on this database, a new database is created.

3. If you're upsizing to an existing SQL Server database, choose Use Existing Database, click Next, and pick the DSN to use from the Select Data Source dialog. You can then skip to step 5. Otherwise, choose Create New Database, and then click Next.

TIP

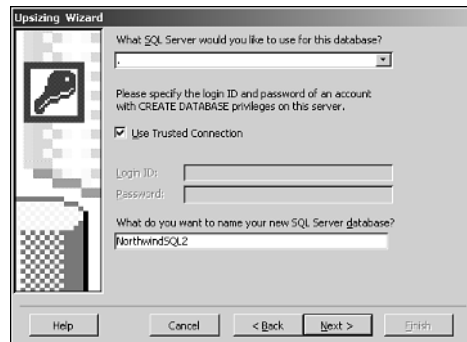
You might upsize a database more than once as you learn the process. When it comes time to upsize a database again, clean out the SQL Server database by creating an

Access Project against the database and deleting all objects in it. The next time you upsize, choose the Use Existing Database option.

NOTE

Upsizing and deleting objects against SQL Server 6.5 creates transaction logs. Make sure that you have adequate device space for the logs configured on the server, and dump them to disk so that the log device won't get full.

4. In the next dialog, select the destination server, and then enter the appropriate login information and the name for the new database (see Figure 23.9). Click Next.

**FIGURE 23.9**

Select the destination server and database name in this dialog.

5. You're asked which tables you want to upsize. This dialog is similar to other wizards. Click the double arrows to move all the tables in the Export to SQL Server column, and then click Next.
6. Specify the attributes you want to export in addition to the data. (Notice in Figure 23.10 that the upsizing tools do quite a bit of work for you. Re-creating all these options manually would be a major pain!) Click Next when everything is as you want it.

**FIGURE 23.10**

You can use the defaults as is in this dialog most of the time.

TIP

In most cases, the default settings in Figure 23.10 are perfectly appropriate for your upsizing task. However, if you're upsizing a large database for the first time, you can select the option **Only Create the Table Structure; Don't Upsize Any Data**. This allows you to do a quick "trial run" of the upsizing process without waiting for the wizard to move large volumes of data to SQL Server.

7. Specify how to configure the upsized application:

- **No Application Changes** creates a SQL Server database but doesn't change the source application. Use this option only if you want to make a server database equivalent to your Access database.
- **Link SQL Server Tables to Existing Application** updates the queries, forms, reports, and Data Access Pages in your application to work with the data moved to SQL Server. Each original table in your application is renamed with a `_local` extension; then the upsized table from the SQL Server is linked to the application. Select this option when you want your client/server application to continue to utilize local or linked Jet tables or existing DAO code.
- **Create New Access Client/Server Application** creates a new Access Project with the name and location that you specify. Select this option when you want your upsized application to use a true two-tier client/server model and are willing to significantly change your existing application to take full advantage of SQL Server's power. See the following section for more information.

After you make your selection, click Finish to begin the upsizing process. You will receive upsizing errors if you have names that conflict with SQL Server keywords, if you don't have appropriate permissions to the server, or if mechanical problems exist, such as insufficient disk space.

NOTE

The Upsizing Report appears at the end of the process in Print Preview, so you can see the completed tasks and detailed errors that occurred during the upsizing. The report is also stored as a report snapshot file (in this example, NorthwindSQL.snp) in the same location as the upsized Access file, so you can refer to it in the future.

The wizard continues past any errors that occur after you acknowledge the message box. Each table name is displayed as the table is upsized or exported to the SQL Server database. When it's complete, you've upsized your application and created a front-end/back-end application (in which you've upsized the data and created table links) or a true client/server application (in which you've created a new Access Project). The code described in Chapter 25, "Startup Checking System Routines Using ADO," relinks tables when the application is started from the front end to the new SQL Server back end.

Creating a Project from an Application

If you tell the Upsizing Wizard to convert your Access application to an Access Project by using the Create New Access Client/Server Application option, the wizard creates a new Access Project file and upsizes objects from the source database to the Project like this:

- **Queries.** Queries are converted to either views, stored procedures, or, if using SQL Server 2000, functions, depending on their purpose. In versions before SQL Server 2000, select queries without `ORDER BY` are converted to views. Any select query with `ORDER BY` must be converted to a query called from a stored procedure, because SQL Server views can't contain the `ORDER BY` clause. Queries containing parameters (include other queries that do so) are also converted to stored procedures, as are update and delete queries. A few query types (SQL Pass-Through queries, data-definition queries, union queries, and queries with many nesting levels) aren't directly supported by SQL Server and aren't upsized; you must contrive a programmatic alternative to these queries.
- **Forms and reports.** The wizard might need to modify form and report properties that contain the name of a table, query, or field or contain SQL syntax. The changes mostly involve updating the `ControlSource`, `RecordSource`, and `RowSource` properties to use the names of new views, table or field name aliases, or stored procedures created during upsizing.

- **Data Access Pages.** The wizard reviews your Data Access Pages and updates connection and data binding information in them to use the new SQL Server database. A copy of each changed Data Access Page file is made to the same location as the Access Project with `_CS` added to the end of the filename. The newly created pages retain the original name in the Access Project so that hyperlinks between the pages work correctly.
- **Macros and modules.** The wizard doesn't change macro commands or module code. After upsizing to a Project, you should review your code that uses `Recordset` objects from DAO and change the code to use ADO instead. See Chapter 5, "Introducing ActiveX Data Objects," for ADO information.

Distributing a Client/Server Solution

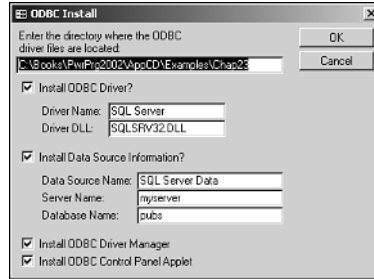
After your application is fully migrated to client/server, a few outstanding issues still exist, such as how to distribute the application. The largest challenge comes into play when you need to send it to someone geographically removed from you or determine that you have a salable commercial application. Consider creating the SQL Server database, setting up an ODBC data source, and loading any existing data into the SQL Server tables.

Programmatically Setting Up an ODBC Data Source

To implement a client/server solution, you need to set up the ODBC data source. Doing so manually isn't difficult, but if you're developing applications for the commercial market or large numbers of corporate users, consider accomplishing this programmatically; otherwise, you'll have to visit each desktop yourself and configure each PC individually, or teach your users how to install a data source themselves. Visiting every single desktop is generally impractical—especially in the case of a commercialized application—and your users might not be able to set up the ODBC data source. Therefore, creating your own custom installation program would be a good alternative.

You can use DLL functions to set up an ODBC data source. The ODBC Administrator itself is a DLL that contains a number of functions for installing drivers and adding, modifying, and removing data source definitions. The installation program uses these functions to install the ODBC components necessary to use the sample time and billing database.

Listing 23.3 shows the primary subroutine that runs the installation program, which is contained in the code module for `frmODBCInstaller` in `Chap23.mdb` on this book's Web page. This routine installs the SQL Server ODBC driver and sets up a data source; it also can optionally install ODBC Driver Manager and the Control Panel application. You can use this code as the basis for your own installation program. Figure 23.11 shows `frmODBCInstaller` in Form view.

**FIGURE 23.11**

Here is the front end for setting up the ODBC data source in Chap23.mdb.

LISTING 23.3 Chap23.mdb: Running the Installation Program

```
Private Sub DoInstallerFunctions()
    On Error GoTo DoInstallerFunctions_Error

    Dim strInfFile As String
    Dim strDriverInfo As String
    Dim strPath As String
    Dim lngBytes As Long
    Dim strDriver As String
    Dim strAttrib As String

    ' Install ODBC driver?
    If Me!chkInstDriver Then
        ' Assign null to INF file argument to force installer to use the driver info
        strInfFile = vbNullChar

        ' Build driver information
        strDriverInfo = Me!txtDriverName & vbNullChar & "Driver=" & _
            Me!txtDriverDLL & vbNullChar & "Setup=" & Me!txtDriverDLL & _
            vbNullChar & "APILevel=1" & vbNullChar & "ConnectFunctions=YYN" & _
            vbNullChar & "SQLLevel=1" & vbNullChar & vbNullChar

        ' Set buffer for destination path
        strPath = Space$(acsMaxPathLen)

        ' Call SQLInstallDriver installer function
        If acsSQLInstallDriver(strInfFile, strDriverInfo, _
            strPath, acsMaxPathLen, lngBytes) Then
            mstrSystemDir = Left(strPath, lngBytes)
            SafeCopy mstrSourceDir & Me!txtDriverDLL, _
                mstrSystemDir & "\" & Me!txtDriverDLL
        End If
    End If
End Sub
```

LISTING 23.3 Continued

```

Else
    Error 30001
End If
End If

' Configure data source?
If Me!chkInstDS Then
    ' Build data source attribute info
    strAttrib = "DSN=" & Me!txtDSN & vbNullChar & "Server=" & _
        Me!txtServer & vbNullChar & "Database=" & Me!txtDatabase & _
        vbNullChar & "Description=SQL Server Data" & vbNullChar & _
        "OEMtoANSI=No" & vbNullChar & vbNullChar

    ' Call SQLConfigDataSource installer function
    If Not acsSQLConfigDataSource(0&, odbcAddDSN, _
        Me!txtDriverName, strAttrib) Then
        Error 30002
    End If
End If

' Install driver manager?
If Me!chkInstDriverMan Then
    ' Set buffer for destination path
    strPath = Space$(acsMaxPathLen)

    ' Call SQLInstallDriverManager to get install path
    If Not acsSQLInstallDriverManager(strPath, _
        acsMaxPathLen, lngBytes) Then
        Error 30003
    Else
        mstrSystemDir = Left(strPath, lngBytes)
        SafeCopy mstrSourceDir & "odbc32.dll", mstrSystemDir & "\odbc32.dll"
        SafeCopy mstrSourceDir & "odbccr32.dll", mstrSystemDir & "\odbccr32.dll"
    End If
End If

' Install control panel applet?
If Me!chkInstAdmin Then
    ' Set buffer for destination path
    strPath = Space$(acsMaxPathLen)

    lngBytes = acsGetSystemDirectory(strPath, acsMaxPathLen)
    If lngBytes > 0 Then
        mstrSystemDir = Left(strPath, lngBytes)
    End If
End If

```

LISTING 23.3 Continued

```

        SafeCopy mstrSourceDir & "odbccp32.dll", mstrSystemDir & "\odbccp32.dll"
        acsWritePrivateProfileString "MMCPL", "odbc", mstrSystemDir & _
            "\odbccp32.dll", "CONTROL.INI"
    Else
        Error 30004
    End If
End If

MsgBox "Installation complete!", vbInformation, Me.Caption

DoInstallerFunctions_Exit:
Exit Sub

DoInstallerFunctions_Error:
Select Case Err
    Case 30001 To 30005
        Call acsErrODBCInst("DoInstallerFunctions")
    Case Else
        MsgBox Error, vbExclamation, Me.Caption
    End Select
Resume DoInstallerFunctions_Exit
End Sub

```

NOTE

Although this installation program is included as an Access 2002 database, the code is compatible with Visual Basic versions 4.0 and later if you prefer to write your application with that tool.

Re-creating a SQL Database with Server Scripts

SQL Server provides a great utility for re-creating SQL Server objects. By using Enterprise Manager, one of the administrative tools provided with SQL Server, you can select which objects you want to create, and SQL Server writes the scripts. This is especially useful for distributing your client/server application commercially, or for distributing to sites that don't have a resident SQL Server expert to set up the SQL Server device, set up the database, and then create all the tables with their necessary defaults, rules, and so forth.

NOTE

Enterprise Manager isn't included with the Desktop version of SQL Server that ships with Microsoft Office.

To create SQL scripts, follow these steps:

1. Start SQL Enterprise Manager. This application exists on the Windows NT Server machine running your SQL Server, but must be added to a Windows development workstation by installing the SQL Server Client Utilities from a SQL Server setup CD-ROM.
2. Select the database you want to create scripts for in Server Manager's tree view.
3. From the Tools menu, choose Database Scripting.
4. In the Generate SQL Scripts dialog, select the objects you want to create scripts for, set any scripting options, and then click OK.

Alternatively, you can use Automation to create your SQL Server database:

- An Automation feature since SQL Server 6.0 is Distributed Management Objects (DMO). The SQL-DMO object model includes objects, properties, methods, and collections that you can use to write programs to administer multiple SQL Server databases distributed across your network. You can write VBA or C++ programs that use SQL-DMO objects (from Access, create a reference in VBA to the Microsoft SQLDMO Object Library). Search the SQL Server Books Online for more information on DMO.
- SQL Server 7.0 and 2000 have an additional tool set—Data Transformation Services (DTS)—that you can script or program for various database management tasks.
- Use ADO to send DDL commands from your application to SQL Server to modify a database structure. See the Access help topic “Microsoft ADO Extensions for DDL and Security (ADOX) Programmer's Reference” for more information.

Loading Existing Data into SQL Server

After creating your tables in SQL Server, you should have a plan for loading any existing data onto the server. This can be as simple as a series of Access queries or as complex as an interactive, flexible data-conversion program.

Keeping Certain Issues in Mind with Access and SQL Server

You now should better understand not only what client/server applications are, but also how to implement them successfully. The most important points to take away from this chapter are the following:

- **Plan your file-server applications with client/server migration in mind.** It's easier to do it right the first time than be forced to rework your application later on.
- **Limit your data.** Always be conscious of how much data you're passing across your network. Move only as much data as is truly needed.
- **Take advantage of server processes.** Make sure that your queries are actually processing on the server. You can use some tracing mechanism to verify this, as mentioned in the section "Query Processing on the Server." Also consider the advantages of using SQL Pass-Through queries and stored procedures. Another new feature found in SQL Server 2000 is the ability to support cascading updates and deletes with relationships, as found with Access relationships.
- **Know what upsizing is all about.** Understand that upsizing means moving your data to a back-end server that specializes in query processing.

Summary

Using Access to connect to SQL Server data is a whole new world of development. Keeping this in mind when dealing with your Access application in the beginning can save you a great amount of frustration and work when upsizing down the road. Your knowledge of working with both products can someday save you that big project.

For more information on using Access as a front-end application for your client/server development, see the following chapters:

- Chapter 2, "Coding in Access 2002 with VBA," gives an overview of VBA and how to use its commands to their fullest.
- Chapter 5, "Introducing ActiveX Data Objects," looks at the ActiveX Data Objects (ADO), including recordset manipulation and creating database objects on-the-fly.
- Chapter 8, "Using Queries to Get the Most Out of Your Data," covers how to optimize your queries and to create query objects by using VBA.
- Chapter 24, "Developing SQL Server Projects Using ADPs," discusses the client/server model in more detail by further describing the capabilities of Access Projects.

Developing SQL Server Projects Using ADPs

CHAPTER

24

IN THIS CHAPTER

- Understanding Project File Architecture 780
- Working with Projects 789
- Building a Client/Server Application 795
- Working with Views 815
- Working with Stored Procedures 818

Prior to Access 2000, you could link data to your application from SQL Server or other server-based databases by using ODBC. When table links to a server were created, the server's data was available to your Access application, but the server's structure wasn't. The structure of the server database and the capability to manipulate it weren't previously available in the Access development environment.

Access 2000 changed this model. Although you can still create Jet-based applications that include server table links, you can also create a type of application known as an Access Database Project (ADP). A Project doesn't bind to a Jet database or rely on table links. It instead links exclusively to a SQL Server database. More important, a Project binds to the server database through the OLE DB technology, which exposes the design of the server's database objects (tables and so forth) to the Access development environment.

This chapter examines the primary features and uses of an Access Project that links an Access application to a SQL Server database.

NOTE

The Access documentation differentiates between Access Projects and traditional Jet-based applications by referring to applications with Jet tables as *Access databases*. In practice, this terminology can be confusing because an Access application has a database whether it uses Jet tables or SQL Server tables. This chapter uses the term *Jet-based application* to define an application that uses the traditional Access-to-Jet architecture (an .mdb file).

Understanding Project File Architecture

Before you create a Project, you need to understand a few basics:

- The OLE DB technology
- The data link feature
- The Microsoft SQL Server Desktop Engine
- The primary objects available in a server database

Understanding OLE DB

OLE DB is a set of data access utilities that expose data and database structures to your applications. In the past, you used ODBC to link an Access application to non-Jet data. Although ODBC is still part of the Microsoft Office architecture, the OLE DB components are more important than ODBC to your present and future development efforts.

With OLE DB, Microsoft defines a standardized architecture that provides a connectivity layer to various data types. OLE DB provides access to relational data like that in Jet and SQL Server databases. It also supports access to Microsoft Exchange data, flat files, spreadsheet data, and non-Microsoft server products.

Within OLE DB are three important terms to know:

- **Data source.** The storage location for the data records, such as a table, file, message, or document.
- **Data consumer.** The application that accesses data in the data source, such as an Access Project or C++ application.
- **Provider.** The OLE DB engine (essentially, a small program) that allows the data consumer to connect to the data source, such as the OLE DB Provider for SQL Server used by an Access Project.

In Access 200x, you don't need to know how to programmatically communicate with an OLE DB provider; instead, you use a Data Link object to provide data to your application. Of course, you can also write VBA code to retrieve and manipulate data; when you do this with ADO, you are using OLE DB via ADO.

Linking to Data

When you create an Access Project, the Project uses an OLE DB data link to bind data and objects in a SQL Server database to the Project. The OLE DB provider that you use to create a data link between an Access application and SQL Server 6.5, 7.0, or 2000 (Access 2002) is installed with Access 200x.

You can also use OLE DB providers to work with data from VBA program code by using ADO, as discussed in Chapter 5, "Introducing ActiveX Data Objects."

NOTE

Several OLE DB providers are installed by Access setup. To see the list of providers on your system, view the property sheet for a new or existing data link, as described in the following section and shown later in Figure 24.2. For the purposes of this chapter and SQL Server development in general, the provider you will use is called the Microsoft OLE DB Provider for SQL Server.

Data Links and Access Projects

The data link between an Access Project and its SQL Server database is managed from the user interface. The File menu's Connection option, available only from an Access Project, displays the Data Link Properties dialog. Here, you define a data link to the SQL Server data for your application (see Figure 24.1). An Access Project can use only SQL Server for the data and objects displayed in its Database window.

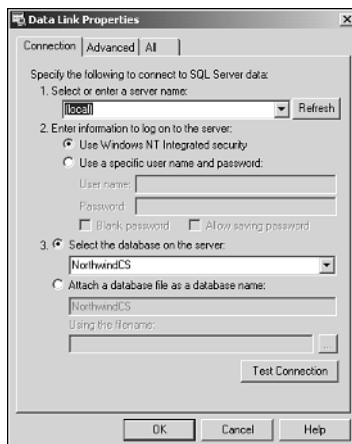


FIGURE 24.1

A Jet-based Access application doesn't show the File menu's Connection option.

Access .mdb files are still built on the Jet database engine and the Data Access Objects (DAO) technology, which uses the Microsoft Jet OLE DB Provider.

In a Jet-based application, each table in the application could be local to the application database or linked from another database. The Data Link Properties dialog in an Access Project enables you to identify the SQL server that will provide the application with data and database objects. As a result, there's no central way to manage all the links between the application and its data. In contrast, an Access Project binds to a single server database through a single connection—a data link. Thus, you can change the entire database for a Project through the Data Link Properties dialog.

TIP

The capability to change the entire database for a Project with a single data link change makes application development and enhancement easier. You can develop an

application while connected to a test database and then easily change the Project's data link to the live server to place it in production.

Data Links and VBA Code

You can also use the OLE DB data links by creating a data link disk file and using it from application code. Data link files, with a .udl file extension, can be used from an Access Project and a Jet-based application to define connection information for any OLE DB provider.

To create a data link file, follow these steps:

1. In Windows Explorer, select the folder in which to add the new data link.
2. From the File menu, choose New and then Text Document.
3. Name the new data link file with the .udl extension, and then press Enter. A message box will appear, warning you about changing the extension. Click Yes.
4. Click the link file to open the Data Link Properties dialog.
5. Click the Provider tab. A list of available providers appears (see Figure 24.2). Select a provider and click Next.

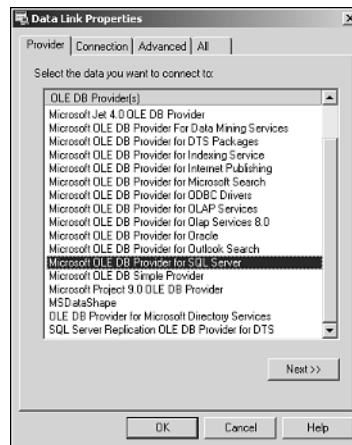
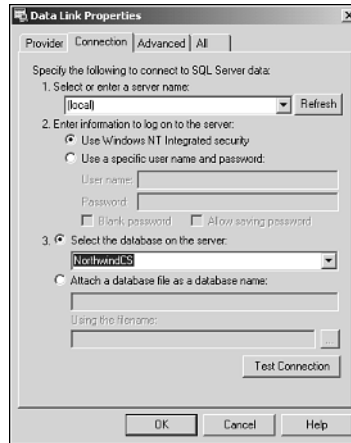


FIGURE 24.2

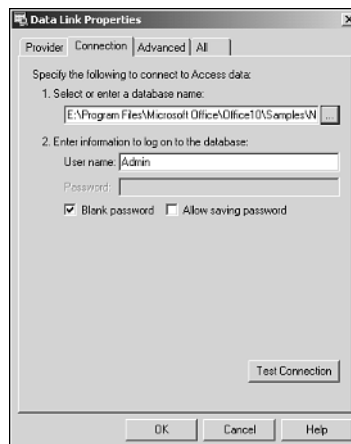
These OLE DB providers are installed with Access 2002.

6. On the Connection page, complete the required information. The type of information that must be entered varies based on the provider selected for the data link. For example, if

the Microsoft OLE DB Provider for SQL Server is selected, a server name, database name, username, and password must be entered on the Connection page (see Figure 24.3). On the other hand, if the Microsoft Jet 4.0 OLE DB Provider is chosen, a database file (.mdb) path and name must be entered (see Figure 24.4).

**FIGURE 24.3**

Specify these database properties for a data link that uses SQL Server.

**FIGURE 24.4**

Specify a database filename for a data link that uses Jet.

7. Always click the Test Connection button to determine whether the data link information you entered provides a successful data connection.

The value of encapsulating connection information in a data link file is similar to the value received from ODBC data sources. If you've worked with ODBC before, you know that you can use a data source to connect application code to a certain set of data, and later change the data source to point to different data without changing any code. Similarly, a data link file can simplify your development and coding efforts.

Assume that several remote warehouses each create a daily receiving information file as an Access .mdb and send it in e-mail to corporate headquarters. Sally, the inventory control manager at headquarters, has an Access Project that validates the warehouse files and loads the data into a SQL server. The Access code to do this in the Project looks something like this:

```
Dim cnn As New ADODB.Connection
Dim rst As New ADODB.Recordset

cnn.Open "File Name=C:\Whse\Load.Udl"
rst.Open "Receipts", cnn, , , adCmdTable

' ...
' In a Do loop, each record is validated and added to a server table here
' ...

rst.Close
cnn.Close
```

Notice that no Access database is specified. Instead, a data link file is named to provide connection information to the code. The data link file Load.Udl was created to use the Jet 4.0 OLE DB provider, which works with Access database files. In the data link file, a database name is specified, but you can easily change this in Explorer without changing any code or properties inside the Access application containing the code. Each time the code runs, it looks in the data link file for connection information. So when Sally gets a file from a remote warehouse, she simply does this:

1. Opens the property sheet for the Load.Udl data link file and clicks the Connection tab.
2. Changes the database name to match that of the most recently received file.
3. Clicks a button in the Access application to run the preceding code.
4. Extracts the next warehouse file from email and repeats this process.

Notice from Sally's workflow that she can use the same code chunk to talk to several different .mdb files in several different locations without changing any code; she changes only the connection information in the data link file. This approach to coding gives you significant flexibility when writing code to work with data that's not accessed via the default connection for a Project.

CAUTION

This illustration is intended to show one way that data link files can be used, not to provide the best example of a workflow for Sally's business need. If you code your application to depend on a key .udl file such as the one in this example, be sure that you include code to test for improper manipulation or deletion of the data link file.

The Microsoft Data Engine

Much of your Access Project development can be done rapidly and portably, thanks to a version of SQL Server that runs on your workstation. This version is the Microsoft SQL Server Desktop Engine (formerly known as Microsoft Data Engine), or MSDE for short.

This scaled-down version (but only barely so) of SQL Server 2000 is designed to run on workstations. You can install MSDE on Windows 9x and Windows NT/2000 workstations. Because MSDE is based on the same data engine as SQL Server, most Microsoft Access Projects or client/server applications run on either version unchanged.

Here are the limitations of MSDE as compared to a Windows NT Server-based SQL Server:

- A 2GB database size limit
- Supports Symmetrical Multiprocessing (SMP) for only two CPUs (only one CPU supported on Windows 9x)
- Different memory management model (MSDE attempts to minimize memory usage, which might reduce performance)
- Can't be a transaction replication publisher (but can be a subscriber)
- A limit of five simultaneous active query batches (also called *threads*)
- Doesn't support a few high-end features such as parallel queries and fail-over clustering

TIP

If you need to do workstation development against very large SQL Server databases, upgrade the MSDE to the full version of SQL Server 2000. Simply install MSDE on your workstation from the Office CD set and then run setup from the SQL Server CD set. SQL Server setup detects the MSDE and asks whether you want to upgrade. After the upgrade, your local Access Projects function the same, but the 2GB database size limit is gone. Another advantage to installing the full version of SQL Server, if you own it, is that Enterprise Manager is installed on your workstation, providing a broader toolset for managing SQL Server than is found in an Access Project.

MSDE isn't installed as part of the Office XP setup. You install it from the CD file \MSDE2000\Setup.Exe after you complete Access/Office setup. On a Windows NT workstation, MSDE runs as a service and starts automatically when you log on. On a Windows 9x workstation, the SQL Server engine and Service Manager are started with shortcuts in the Startup program group. You should see a small server icon for the Service Manager on the Windows taskbar that you can click to start SQL Server if it's not running after you complete setup. If you want SQL Server to start when you log on, be sure to select the Auto-Start Service When OS Starts option at the bottom of the dialog. Figure 24.5 shows the Service Manager dialog.

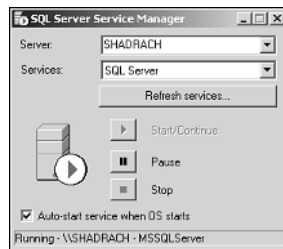


FIGURE 24.5

The SQL Server Service Manager.

Objects on a SQL Server

At times, you can use an Access Project to connect to an existing SQL Server to create forms, reports, or code against the data. In other cases, you might need to create or enhance a SQL Server database from the Access Project. Figure 24.6 shows the Database window in an Access Project; note that the list of objects you can work with varies from the list in a Jet-based application.

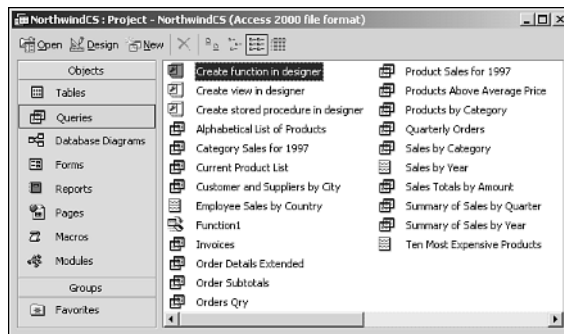


FIGURE 24.6

The Database window for an Access Project.

From an Access Project, you can work with the following SQL Server objects and tasks:

- **Tables.** SQL Server tables are much like Jet tables in that they have fields, properties, and relationships to other tables. You work with table designs and data directly from a Project's Database window.
- **Queries.** In an Access 2002 ADP, queries are made up of three types of objects:
 - **View.** A SQL Server *view* is what Access calls a select query, but of a very limited variety. A SQL Server view joins tables and optionally applies criteria to the result-set; it doesn't allow for actions (append, create table, delete, or update), versions before SQL Server 2000 didn't allow you to sort the results, and all versions don't support passed parameters. You work with view designs and data directly from a Project's Database window.
 - **Stored procedures.** These scripts contain program code and database instructions, mainly for recordset manipulation. Think of a stored procedure as compiled code that runs as fast as possible because it's stored in the database with the objects it affects, rather than in your application. You work with stored procedures directly from a Project's Database window.
 - **User-defined functions.** This object type, new to SQL Server 2000 and Access 2002, combines the best features of views and stored procedures. It takes these into a single query that you can nest, pass parameters to, sort, and return values. User-defined functions can be a better alternative to views and stored procedures because you can return a single table of data or scalar value, and simplify the complexity of your SQL statement syntax. They are not used for batch actions however.
- **Database diagrams.** A SQL Server database diagram is more than a visual representation of a database's structure—you can create and modify tables, columns, and their attributes by working with the diagram. You work with diagrams directly from a Project's Database window.
- **Triggers.** A trigger is a stored procedure attached to a table-level event and defined to run automatically when a record in its table is modified. Triggers can be defined to run when table data is deleted, inserted, or updated, and are used to apply validation rules or maintain data integrity. To create and manage triggers, choose the Tables object in the Database window and then select Triggers from a table's shortcut menu.
- **Replication.** You can establish and maintain replication settings that enable the SQL Server database to publish or subscribe to other SQL Server databases. SQL Server 7.0/2000 can also publish data to Jet 4.0 tables. To manage replication, choose Replication from the Tools menu. For more information on replication, see Chapter 22 (although it discusses replication with Access .mdb's).

- **Security.** To create and maintain a security model for the SQL Server database objects and data, choose Security from the Tools menu.

Working with Projects

You've probably already created Jet-centric Access applications consisting of two primary components: a front-end application database with queries, forms, reports, and modules, and a related back-end database with the data tables. Access Projects employ the front-end/back-end concept but in a different model. In the front end (an Access .adp file) are forms, reports, and modules with VBA code. The back end (a SQL Server database) consists of the views, user-defined functions, triggers, data tables, and more application code in stored procedures. Because the views and stored procedure code are in the back end, the processing of data retrieval and other data manipulation is optimized for maximum performance and minimum network traffic; thus, an Access Project fits the definition of a client/server application.

NOTE

The Access Project model fits the two-tier client/server model, defined as having front- and back-end components. A three-tier client/server model can also be used with Access but requires additional development tools (Visual Basic or Visual C++) to create the COM components that manage data in the middle tier. Using Jet-based applications is easier in a three-tier model than using Projects because Jet-based applications can provide temporary local tables and queries that can be used for bulk local data manipulation and with forms and reports. Access Projects don't provide any local storage for table data.

Creating a New Project

To create a new Access Project, follow these steps:

1. From the File menu, choose New.
2. In the New dialog, select Project (Existing Database) or Project (New Database) on the General page and click OK.
3. Enter a filename and location for the new Project (.adp) file in the File New Database dialog and click Create.

The next dialog that appears depends on whether you're linking to an existing database or creating a new database.

If you're linking a new Project to an existing database, continue with these steps:

1. In the Data Link Properties dialog, enter the server name for a SQL Server (6.5, 7.0, or 2000) on your domain or the name of your local server if you're using MSDE (see Figure 24.7). The local server's name is the same as your workstation name (the Computer Name on the Identification page of the Windows Network dialog in Control Panel).



FIGURE 24.7

Use the Data Link Properties dialog to connect a Project to a SQL Server.

TIP

If you don't know the name of your workstation, enter **(local)** as the server name. This trick is especially handy because the server name combo box doesn't always list the local MSDE server name, even after clicking the Refresh button.

2. Select the security model to use. You can select Windows NT/2000's Integrated security to pass your logon name and password to the SQL server, or you can use a name and password related to a SQL Server logon. The username sa, used to administer the database, is commonly used during development, especially when you are developing against MSDE on your local workstation.
3. Select a database on the server. If no databases are shown, your logon information isn't correct for the specified server. Test the connection and then click OK.

TIP

This dialog also lets you specify a database filename and local database name for the file rather than select an existing database. This technique, known as *attaching a database*, provides advanced functionality that's beyond the scope of this chapter. In a nutshell, SQL Server 7.0 databases are stored on disk in data (.mdf) and log (.ldf) files. A data file (and therefore its database) can be detached from a SQL server (using the system stored procedure `sp_detach_db`) and moved to a different machine. By specifying the filename of a detached .mdf file in this dialog, the file and its database are attached to the SQL server you specify and added to the server's catalog. This technique enables you to distribute the same database to multiple mobile users via e-mail, CD, or FTP, for example.

If you shut down a database before shipping it, you need to send only the data (.mdf) file. If the database is live and in use when moved, you must also send the log (.ldf) file and use the system stored procedure `sp_attach_db` to connect the data and log files to the receiving server.

When linking a new Project to a new database, continue with these steps:

1. In the Microsoft SQL Server Database Wizard dialog (see Figure 24.8), select or enter the name of a valid SQL server.

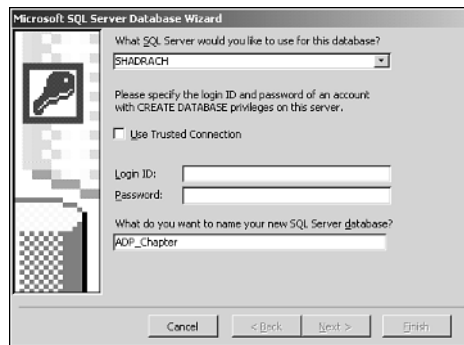


FIGURE 24.8

New databases are created with the Microsoft SQL Server Database Wizard.

2. Enter a username and password that gives you access to the specified SQL server with Create Database rights (if you know the password for the sa logon, use that account).

3. Enter a name for the new database and click Next. If you're creating a SQL Server 7.0/2000 database, the process is complete and you simply click Finish. If you're creating a SQL Server 6.5 database, a dialog appears that requires you to define device, file, and log size information before you click Finish.

NOTE

When you create a new Access database or Project file, Windows lets you overwrite an existing file of the same name. SQL Server, on the other hand, doesn't overwrite an existing database with a new one if you attempt to reuse a database name. You must specify a unique name for a new database; if you don't do so in this dialog, the wizard adds the string 1 to the end of the name you specify.

Project Properties

A Project contains different objects, properties, and architecture than a Jet-based application. However, both application types are similar in their use of forms, reports, and VBA modules. Table 24.1 summarizes the key differences.

TABLE 24.1 Key Differences Between Access .mdb Applications and .adp Projects

<i>Item</i>	<i>Application</i>	<i>Project</i>
AllDatabaseDiagrams object	Not available	Available
AllFunctions object	Not available	Available
AllQueries object	Available	Not available
AllStoredProcedures object	Not available	Available
AllViews object	Not available	Available
Append query	Query	Stored procedure
Delete query	Query	Stored procedure
Parameterized query	Query object	Stored procedure/function
Select query	Query	View/function
Update query	Query	Stored procedure
Relationships	Relationships	Database diagram
Startup properties	CurrentDb	CurrentProject

TABLE 24.1 Continued

<i>Item</i>	<i>Application</i>	<i>Project</i>
<i>File Menu Options</i>		
Connection	Available	Not available
Get External Data and then Link Tables	Available	Not available
<i>Tools Menu Options</i>		
Database Utilities and then Backup SQL Database	Not available	Available
Database Utilities and then Database Splitter	Available	Not available
Database Utilities and then Drop SQL Database	Not available	Available
Database Utilities and then Linked Table Manager	Available	Not available
Database Utilities and then Restore SQL Database	Not available	Available
Database Utilities and then Copy SQL Database	Not available	Available
Database Utilities and then Transfer SQL Database	Not available	Available
Database Utilities and then Upsizing Wizard	Available	Not available
Security	Jet security	SQL Server security

TIP

If your add-in or other executing code needs to know what type of Access application it's running in, use code like this:

```
bytType = Application.CurrentProject.ProjectType
```

This property returns a value of 1 (acADP) for a Project file and 2 (acMDB) for a Jet-based application file.

Securing a Project

The last item in Table 24.1 highlights a difference in the security models between an Access application and a Project. In Jet-based applications, you can apply user/group security to application objects such as forms, reports, and modules. In Access Projects, all security exists on the server, and the objects in Project .adp files can't be secured by user or group.

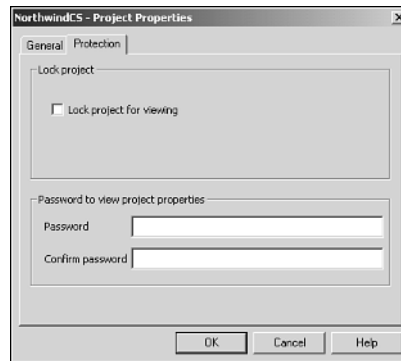
Fortunately, code and application objects can still be protected from user review and modification. In the same way as an Access .mdb file can be saved as an .mde file with code and objects precompiled and locked, an .adp file can be saved as an .ade with similar restrictions. From the Tools menu, choose Database Utilities and then Make ADE File. The resulting file won't allow the following:

- Importing or exporting of forms, reports, or modules
- Opening forms, reports, or modules in Design view
- Manipulating VBA source code, references, or libraries
- Adding, deleting, or changing references to object libraries or databases

NOTE

An .ade (locked) file can't be decompiled back into an .adp (open source) file. You must save a copy of the original .adp file for each version of an .ade that you create to perform application upgrades and maintenance.

The VBE provides an alternative security level that you can apply to an Access Project (or application). On the Protection page of the Project Properties dialog (available from the VBE's Tools menu), configure the password to be used when VBA code security is toggled on (see Figure 24.9). The Lock Project for Viewing check box provides the toggle to enable/disable the security enforcement. When enabled, the option prevents anyone from viewing or editing any module, form, or report code without entering the password. A secured module also can't be exported or deleted. To read more about securing VBA code, reread Chapter 20, "Securing Your Application."

**FIGURE 24.9**

Use this dialog to prevent viewing and editing of VBA source code.

Building a Client/Server Application

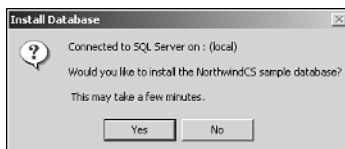
Up to now, I've laid the pieces of the Access Project puzzle on the table and clarified what they are. Now it's time to start fitting them together to make a usable, productive application.

Architecturally, an Access Project combines application objects and pointers to server objects through a data link. The Project (.adp) file stores the forms, reports, Data Access Pages, macros, and modules. On the server reside the tables, views, database diagrams, and stored procedures. By using the data link, the Project causes the server to execute code in stored procedures or to return records from a view or table.

An Access Project retrieves data from the server in one of several ways:

- By selecting records into a table or view datasheet
- By retrieving records for use in form or report controls
- By creating recordset objects from server data through VBA code calling the ActiveX Data Objects (ADO) components

The rest of this chapter uses the sample Project file `NorthwindCS.Adp` that ships with Access. Locate and open the Project now if you haven't done so already; it's in the `\Program Files\Microsoft Office\Office10\Samples` folder. When you first open this Project file, it will look for a matching database (`NorthwindCS`) on your local MSDE SQL Server installation. If your local SQL Server service isn't started, Access will attempt to start it for you. If you don't have MSDE or a SQL Server Desktop installation on your computer, Access will ask you for the server name, user name, and password needed to log on to a server on your domain. If Access doesn't find the sample database on the server you specify, it will ask whether you want the database to be created (see Figure 24.10).

**FIGURE 24.10**

The NorthwindCS Project will create its own SQL Server database if one isn't found.

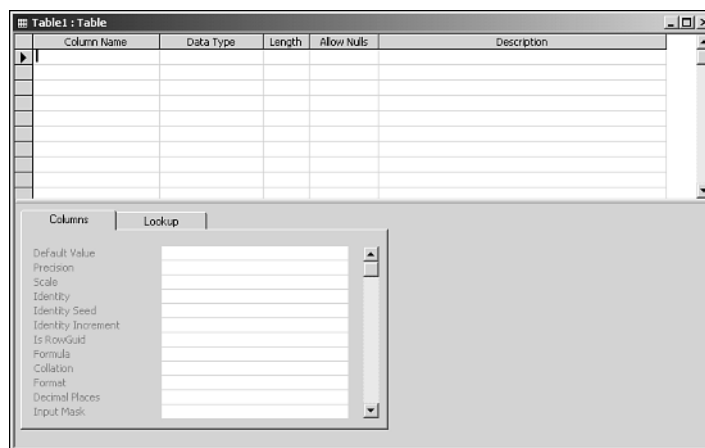
TIP

Northwind uses the NorthwindCS.SQL script file to create its sample database. You can open this script file with Notepad, WordPad, or Word from the \Program Files\Microsoft Office\Office10\Samples folder and review its syntax if you want to learn more about SQL Server's Data Definition Language (DDL).

Working with Tables

The Access Database window for a Project enables you to design and modify tables on the linked SQL Server database. Simply highlight the Tables icon in the Database window and click New. Unlike working in a Jet-based application, you must enter a valid table name in the pop-up Choose Name dialog *before* you can design the table (see the following section “Naming Conventions for Objects” for information on table naming conventions).

Figure 24.11 shows the table design window.

**FIGURE 24.11**

You create and modify tables in this layout.

Enter a column name and data type for each column you want to create. Press Tab to move around in the design window. Each column requires a name and data type; the other attributes are optional. Table 24.2 describes a table column's attributes.

TABLE 24.2 Column Attributes in SQL Server

<i>Attribute</i>	<i>Description</i>
Column Name	The column's name, which must follow the rules defined in the following section. A column name must be unique within its table; however, a generally accepted convention for relational database design is that a column name has the same usage and data type throughout a single database.
Data Type	The column's data type, using the data types defined later in the section "Server Data Types." The default data type in column design view is char (character).
Length	The maximum number of characters that can be entered, or the storage size for a data type, depending on the data type. For non-character data types, the value displayed for this attribute is the field's storage size and can't be changed. For char and nchar data types, the default value for this attribute is 10 characters; for varchar and nvarchar data types, the default is 50.
Allow Nulls	Determines whether a column can allow null values. This attribute is selected by default. If you're adding a column to an existing table and clear this check box (to disallow nulls), first make sure that the column has no null values.
Default Value	Specifies a value to be entered into a column when the record is saved and no other value is entered. If a column doesn't allow nulls, no value is entered before the record is saved, and a default value exists, Access places the default value in the column. If a column doesn't allow nulls, no value is entered before the record is saved, and a default value doesn't exist, the record can't be saved. The default value for this attribute is blank.
Precision	The total number of digits that can be entered in a numeric data type, without punctuation. For example, the number 6543.21 has six digits of precision. The default value for this attribute is 0.
Scale	The maximum number of digits that can be entered to the right of a decimal point for a numeric data type. For example, the number 654.321 has a scale of three. The default value for this attribute is 0.

TABLE 24.2 Continued

<i>Attribute</i>	<i>Description</i>
Identity	This attribute determines that a column will autonumber from a starting value (the Identity Seed attribute) by adding an increment (the Identity Increment attribute) to each subsequent record. When used with a data type of <code>int</code> and an increment value of 1, this attribute is identical to the <code>AutoNumber</code> column type in Jet-based applications. There can be only one identity type column per table, and the data type must be <code>decimal</code> , <code>int</code> , <code>numeric</code> , <code>smallint</code> , or <code>tinyint</code> , and the <code>Scale</code> attribute must be 0. This attribute is unselected by default.
Identity Seed	The first row in the table receives this seed value in its identity type column, if any. The default value for this attribute is blank, which is translated to the value 1 if an identity column exists and the property isn't set.
Identity Increment	The second row in the table receives the Identity Seed value plus the Identity Increment value; for each subsequent row added, the prior identity column value is incremented by this attribute's value, or 1 if this attribute is left blank (the default).
Is RowGuid	Whereas an identity column generates an ID value unique to the table, a column of type <code>uniqueidentifier</code> stores only GUID (globally unique ID) values that are unique around the world. You can select only the Is RowGuid attribute for a column of type <code>uniqueidentifier</code> . When you select the attribute, Access adds the default expression <code>newid()</code> to the Default Value attribute. SQL Server doesn't automatically populate a <code>uniqueidentifier</code> column; the value must be added via a default value or program code.
Formula	This property displays the calculated value if the field is computed.

CAUTION

The Access table design window relies on an ADO connection to the SQL server to manipulate the table design. Consequently, the error messages returned when you try to save a table aren't Access-specific and can be frustratingly useless (for example, Errors occurred).

TIP

You will come to dislike the feature that clears the table design Clipboard in a Project when its source table is closed. If you want to copy or cut the properties for a table column or columns to the Clipboard, keep the source table design open until you paste the information into the destination table's design window.

You can also create and modify tables from a database diagram. Open the Relationships diagram in the NorthwindCS sample Project to see an sample database diagram and experiment with table design.

Naming Conventions for Objects

Object names in SQL Server vary slightly from those for Jet-based Access Projects. Generally, the rules for SQL Server object names are broader than those in Access/Jet, so if you are familiar with the limitations and naming conventions from past Access releases, you'll have no trouble adapting to SQL Server's tastes or upsizing any existing databases to SQL Server.

In SQL Server, the term *identifier* refers to any object that can have a name, including tables, columns, views, indexes, constraints, rules, triggers, and stored procedures. Identifiers are either regular or delimited:

- A *regular identifier* conforms to the naming conventions described in this section and is used verbatim in table design, stored procedures, or VBA code.
- A *delimited identifier* doesn't conform to the SQL Server naming conventions but is allowed in the database as long as every instance is quoted (with ") or bracketed (with []).

For example, the object name `Order` is a reserved word in SQL Server (part of the `Order By` clause syntax). If you change the table name `Orders` to `Order` in the sample database and then try to create a new stored procedure by using the new table name `Order`, a syntax error results. On the other hand, if you delimit the object name `Order` in your stored procedure, the code runs as expected. The following is an example of this:

```
Alter Procedure prcIdentifiers
As
/* This table name is acceptable per SQL Server's naming conventions */
Select * From Orders
/* This table name is not acceptable and must be delimited */
Select * From [Order]
Select * From "Order"
Return
```

These object name rules for SQL Server 7.0/2000 names qualify as regular (non-delimited) identifiers:

- Object names (even if delimited) must be 128 characters or fewer if they are regular identifiers, or 126 characters if they're delimited (to allow two characters for the delimiter as part of the name).
- The first character must be a letter from A to Z (either case, or their equivalent in a foreign language) or one of these characters: `_`, `@`, or `#`. Other characters are disallowed.
- Subsequent characters can be any allowed first character, a decimal number, or `$`.
- The identifier can't be a Transact-SQL reserved word (Transact-SQL is the scripting language used in SQL Server stored procedures).
- The identifier can't contain spaces.

Keep these considerations in mind when naming objects in your databases:

- Variables, parameters (arguments), and names in stored procedures use the strings `@` and `@@` in defined ways, so using this string at the beginning of an object name might be confusing.
- SQL Server uses the `#` or `##` character strings to begin the name of a temporary table or temporary procedure, so a naming convention that begins object names with this character is discouraged.

Server Data Types

SQL Server column data types are conceptually similar to those in Jet, but the data type names vary, and a SQL Server data type's characteristics might not perfectly mirror those of the related Jet data type. Table 24.3 describes the SQL Server data types.

TABLE 24.3 SQL Server Column Data Types

<i>Data type</i>	<i>Description</i>
<code>bigint</code>	8 bytes (64 bits) that store whole numbers in the range of -2^{63} ($-9,223,372,036,854,775,808$) through $2^{63}-1$ ($9,223,372,036,854,775,807$).
<code>binary</code>	Fixed-length binary data with an assigned length from 1 to 8,000 bytes.
<code>bit</code>	Integer data with a 0, 1, or Null value. You can't index <code>bit</code> columns. This data type indicates true/false information in a Project. With Jet, Access expects to see 0 and -1 for False and True, but in a Project Access correctly translates the value 1 in a <code>bit</code> field to True.

TABLE 24.3 Continued

<i>Data type</i>	<i>Description</i>
char	Fixed-length character data with an assigned length from 1 to 8,000 bytes.
datetime	A date and time ranging from 1/1/1753 to 12/31/9999. The time accuracy for this data type is 3.33 milliseconds, or 1/300th of a second, rounded as needed.
decimal	A number with a fixed precision and scale that can hold values from $(-10^{38} - 1)$ to $(10^{38} - 1)$; the maximum precision attribute value for this data type is 28.
float	A floating-point number that can hold values from $(-1.79E + 308)$ to $(1.79E + 308)$.
image	Variable-length binary data whose length can't exceed $(2^{31} - 1)$, or about two billion bytes. This column type's name doesn't imply that only image files can be embedded in the column, merely that the column holds a binary image of its contents and isn't aware of what's in the image (in other words, the data can't be searched or sorted).
int	A whole number ranging from (-2^{31}) (about two billion) to $(2^{31} - 1)$.
money	Currency values ranging from (-2^{63}) to $(2^{63} - 1)$. The accuracy for this data type is .0001.
nchar	Fixed-length Unicode character data with a length from 1 to 4,000 bytes.
ntext	Variable-length Unicode data with a maximum length of $(2^{30} - 1)$, or about one billion, characters.
numeric	Identical to decimal, defined earlier in this table.
nvarchar	Variable-length Unicode character data with a length from 1 to 4,000 bytes.
real	A floating-point number that can hold values from $(-3.40E + 38)$ to $(3.40E + 38)$. A real is the same as a float with a precision value of 24.
smalldatetime	A date and time ranging from 1/1/1900 to 6/6/2079. This data type is accurate to the minute and doesn't store seconds.
smallint	An integer from -32,768 to 32,767.
smallmoney	Currency values ranging from -214,748.3648 to +214,748.3647. The accuracy for this data type is .0001.

TABLE 24.3 Continued

<i>Data type</i>	<i>Description</i>
text	Variable-length data with a maximum length of $(2^{31} - 1)$, or about two billion, characters.
timestamp	A number of type binary (8) that's unique across the database and provides a record identifier that SQL Server uses for concurrency and transaction control. This value doesn't actually correlate to a date or time, but instead indicates the sequence in which data modifications (deletes, inserts, and updates) occur in a database. The value of this column type isn't directly useful during application development.
tinyint	Integer data from 0 through 255.
uniqueidentifier	A 32-character GUID in the display format xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx that's unique in the world.
varbinary	Fixed-length binary data with an assigned length from 1 to 8,000 bytes.
varchar	Variable-length character data with an assigned length from 1 to 8,000 bytes.

Although Access/Jet table columns are now Unicode-enabled by default (meaning that they can contain characters from non-English display languages), you must explicitly use the Unicode-aware data fields in SQL Server if you intend to store non-English data. The SQL Server data types beginning with n (nchar, ntext, and nvarchar) are used for this purpose. However, although you may be tempted to use the Unicode-enabled data types in all cases to enable your application to handle any type of data in the future, be aware that each character stored in the Unicode data types consumes 2 bytes of storage instead of 1, even if the data isn't Unicode. This means that the disk space requirements of your database will grow much more quickly when Unicode data types are used.

NOTE

You might already know that Access stores any table data of types Memo and OLE Object outside the table's data pages and refers to the long-value data by using a pointer. This technique involves a "double-jump" to retrieve the data values and can negatively affect performance. Similarly, SQL Server stores data of types text, ntext, and image outside the host table, with a similarly negative performance impact. Use these data types as sparingly as possible in your Projects.

Table 24.4 shows the SQL Server data types and compares the data type to the nearest Jet type.

TABLE 24.4 Comparing SQL Server Column Data Types to Access

<i>SQL Server (Project) Data Type</i>	<i>Access (Using Jet) Data Type</i>
bigint	N/A
bit	Yes/No
char	N/A
datetime	Date/Time
decimal	Number (Decimal)
float	Number (Double)
image	OLE Object
int	Number (Long Integer)
int (as identity)	AutoNumber
money	Currency
nchar	N/A
ntext	Memo
numeric	Number (Decimal)
nvarchar	Text
real	Number (Single)
smalldatetime	N/A
smallint	Number (Integer)
smallmoney	Currency
text	Memo
timestamp	N/A
tinyint	Number (Byte)
uniqueidentifier	Replication ID (GUID)
User-defined	N/A
varbinary	N/A
varchar	Text

Using Constraints

In SQL Server, constraints provide the functionality that you get from validation rules in a Jet-based application—they test and limit the values that can be entered into a column. The

constraint model is quite complex and, therefore, very powerful. Properly used, constraints can save quite a bit of validation coding on your forms.

You can create five types of constraints for a table column.

Check Constraints

A *check constraint* limits the value that can be entered into a column. Generally, a check constraint takes a column value and compares it to another column value, a literal, or an expression by using this syntax:

`[columnname] operator (comparisonvalue_or_expression)`

Table 24.5 lists valid SQL Server expression operators.

TABLE 24.5 Operators Used in Column Check Constraints

Operator	Meaning
—	Subtract
%	Modulo
*	Multiply
/	Divide
+	Add
!<	Not less than
!=	Not equal to
!>	Not greater than
&	Bitwise And
^	Bitwise Exclusive Or
	Bitwise Or
<	Less than
<=	Less than or equal to
<>	Not equal to
=	Equal
>	Greater than
>=	Greater than or equal to
All	True if all of a set of comparisons are true
And	True if both Boolean expressions are true
Any	True if any one of a set of comparisons is true
Between	True if the operand is within a range

TABLE 24.5 Continued

<i>Operator</i>	<i>Meaning</i>
Exists	True if a subquery contains any rows
In	True if the operand equals one of a list of expressions
Like	True if the operand matches a pattern
Not	Reverses the value of any other Boolean operator
Or	True if either Boolean expression is true
Some	True if some of a set of comparisons are true
–	Negative
~	Bitwise Not
+	Positive

If a check constraint names a single column, it's a column-level constraint and is bound to the named column. A check constraint that references more than one column (even if the same column name is repeated) is bound at the table level.

The expression portion of a check constraint can use wildcard characters that are valid for a SQL `Like` phrase, including `%` (match any number of characters), `_` (match a single character), and `[range]` (match a single character in the range if the syntax is `[x-x]` or in the set if the syntax is `[xxxx]`). Because the constraint runs on the server, it can use Transact-SQL syntax but can't reference VBA expressions or procedures.

For example, a column-level check constraint named `CK_BirthDate` in the NorthwindCS sample database's Employees table references the SQL Server `getdate` function:

```
([BirthDate] < getdate())
```

Another example is a table-level (multi-column) check constraint named `CK_Discount` in the NorthwindCS database's Order Details table:

```
([Discount] >= 0 And [Discount] <= 1)
```

Here's an example of a column-level check constraint that could be built into the Order Details table in the NorthwindCS sample database. The constraint allows only state abbreviations for Washington and West Virginia by specifying that the first character of `Region` is `w` and the second character is `a` or `v`:

```
([Region] Like '[w][av]')
```

Here are some basic considerations for check constraints:

- A single column can have any number of check constraints.
- A check constraint for a column can reference only the column itself in the test expression, not other columns or tables.
- If multiple check constraints exist for a single column, they execute in the order they were created.
- A single table can have any number of check constraints.
- A table-level check constraint can reference only columns in the same table.

To create a new constraint, follow these steps:

1. Open a table in Design view and view the Properties dialog. You can find the constraint information on the Check Constraints page (see Figure 24.12).

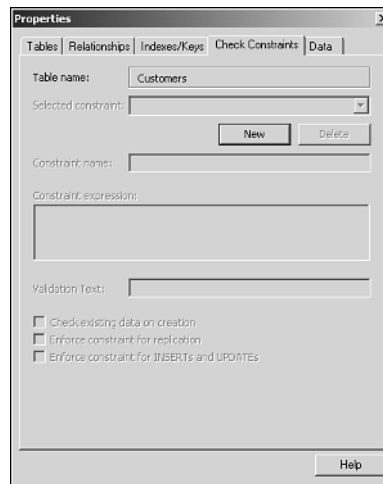


FIGURE 24.12

Constraints are defined in the table's Properties dialog.

CAUTION

A table's Properties dialog doesn't have the standard OK, Cancel, and Apply buttons found in other Windows property dialogs. Every change you make in this dialog is permanent, and closing the dialog with the Esc key or the Close box doesn't undo your changes as it would in similar Windows dialogs.

2. Click the New button. The dialog preassigns a suggested name in the Selected Constraint text box. These names always begin with CK_ and include the table name.
3. In the Constraint Expression box, type a valid constraint for a column or for the table. Text values in a constraint should always be delimited with single quote marks (' ').
4. Rename the constraint from the system-suggested name to your own by typing the new name in the Constraint Name text box.

Default Constraints

A *default constraint* is entered into a column if no other value is provided before the record is inserted into the table. You enter default constraints into the Default Value attribute of the table design window.

A default constraint can contain a constant literal value (delimited with single quote marks or specified with the keyword `Null`) or an expression that evaluates to the data type of the constraint's column. When using an expression to create a default, you can use any SQL Server function, such as `user_name()` or `getdate()`. See the SQL Server Books Online topic "Functions (T-SQL)" for a complete list.

For SQL Server 7.0 and earlier, you can apply defaults to columns of any data type except a timestamp column or any column marked as an identity column. However, with SQL Server 2000, some constraints can be performed using the extended properties.

NOTE

You might already be familiar with the behavior for the default value attribute in a Jet-based application, which displays the default value in an unsaved record on a datasheet or form. This behavior isn't available in a Project; there's no way to force a table datasheet to display a column's default value before a new record is saved. The same is true of forms bound to SQL Server data if you rely on the Default Value column attribute. However, you can create a `DefaultValue` property expression on a form control rather than use a default; the form property will cause the display of the default value even before the record is saved.

Primary Key Constraints

A *primary key constraint* is a column or multiple columns that, taken together, uniquely identify a single record in a table. Each table can have only one primary key constraint, and generally no table should be without a primary key because the Access Project uses the primary key when updating records in the server table.

Columns that participate in a primary key constraint can't allow Null values, and the table design window creates a unique index for the combination of columns in the constraint.

A primary key constraint is also used in *referential integrity*, the process of enforcing relationships between records in different but related tables.

To create a primary key constraint, select one or more column definitions in Table Design view; then click the Primary Key toolbar button or choose Primary Key from the right-click menu.

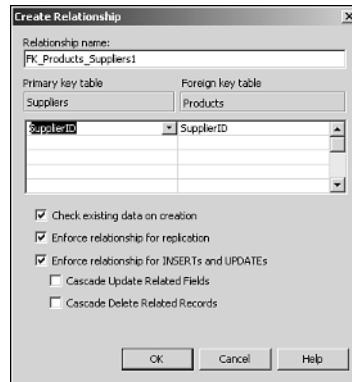
Foreign Key Constraints

A *foreign key constraint* is a column or multiple columns that, taken together, identify a single record in a table for the purpose of relating it to the primary key of a record in a different table. A foreign key constraint maps a primary key in one table to a foreign key in a different table and enforces the relationship between the tables. It can, for example, ensure that no record entered into the foreign key table has a foreign key value that doesn't correspond to a primary key value in the related table.

You create a foreign key constraint in the database diagram designer by defining a relationship, which automatically creates the constraint. To define a relationship, follow these steps:

1. Open an existing or new database diagram. For a new diagram, add a table to the diagram by selecting Show Table from the View menu and dragging a table name to the design surface.
2. Select a primary key definition by clicking in the box(es) to the left of one of its columns.
3. The mouse cursor changes to a pointer, which you drag to the foreign key columns in a different table. The Create Relationship dialog appears. Figure 24.13 shows the relationship's database diagram from the NorthwindCS sample database. The relationship between the Suppliers and Products tables has been deleted and is in the process of being re-created.
4. Enter a relationship name.
5. The primary key column or columns is usually defaulted for you in this dialog in the Primary Key Table column. You must select the column(s) that make up the foreign key in the Foreign Key Table column.

After you create a foreign key constraint through a relationship, you can't delete a record from the table in the primary key side of the relationship until all related records (those with a matching foreign key value) are deleted from the table on the foreign key side.

**FIGURE 24.13**

Use the *Create Relationship* dialog during relationship definition.

TIP

Unlike Jet, SQL Server (before version 2000) doesn't provide a property setting that enforces cascading deletions from a parent table into a child table. To create cascading deletions in SQL Server 7.0 and earlier, you must add a delete trigger to the parent table.

However, you can see in Figure 24.13 that you can now set cascade updates and deletes on relationships in SQL Server 2000.

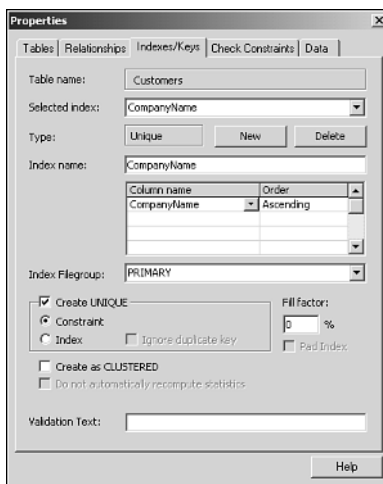
Unique Constraints

A *unique constraint* validates that a value within the specified column is unique in the same table. A primary key, by definition, is also a unique constraint. A table can have multiple unique constraints.

To create a unique constraint, follow these steps:

1. Open a table in Design view and view the Properties dialog.
2. On the Indexes/Keys page, click the New button.
3. Select one or more column names from the Column Name list.
4. Change the index name to an appropriate name for the constraint.
5. Select the Create UNIQUE check box and the Constraint option button.

Figure 24.14 shows this process in the table's Properties dialog.

**FIGURE 24.14**

Define a unique constraint for a table on the Indexes/Keys page.

NOTE

You can use this same page to create indexes as well as constraints. If you want to create a unique index, pay close attention to the Ignore Duplicate Key check box. If this option isn't selected, a bulk insert operation will fail completely if any record being inserted in the transaction attempts to create a non-unique value in the unique index. However, if the option is selected, the bulk insert will succeed for all non-offending records but simply won't add the offending record(s) to the table.

TIP

You can see from Figure 24.14 that in SQL Server 2000 you can now set the text displayed for a validation constraint, using the Validation Text text box at the bottom of the page.

Using Triggers

A *trigger* is stored procedure code that's run (triggered) by a delete, insert, or update event on a table record. A trigger applies validation rules or maintains data integrity. Triggers can

contain most Transact-SQL statements allowed in stored procedures. If you are familiar with the event model for an Access form, you can think of a trigger as a `BeforeUpdate` event procedure.

The code in a trigger can affect rows in other tables, even if this causes other triggers to fire in a sort of chain-reaction cascade. As a result, an insert trigger can be used to provide validation that exceeds the capabilities of a table-level check constraint.

To create a trigger, choose Triggers from the shortcut menu for a table in the Database window. In the trigger maintenance dialog that appears, click `New`; the trigger design window appears, as shown in Figure 24.15.

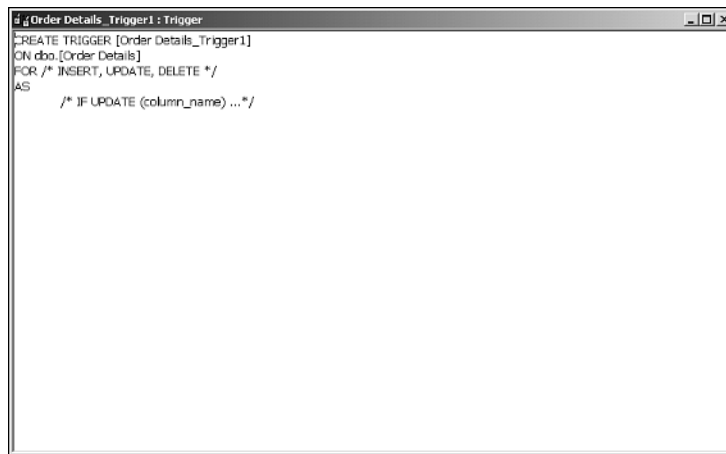


FIGURE 24.15

Here is the design window for a trigger.

In the trigger design window, remove the comment markers around the string `/* INSERT, UPDATE, DELETE */` and keep one or more of the three values to define the type of trigger; the same trigger can be set to fire on more than one of the three trigger-able events. Enter a name for the trigger on the first line, and then modify the trigger code to achieve the desired result.

Listing 24.1 shows a sample trigger that has been added to the sample database's `OrderDetails` table. This code fires whenever a new record is deleted, inserted, or updated in the table. The trigger updates (increments or decrements) the `UnitsOnOrder` field in the `Products` table for the product being affected by the order record. Review the code comments in the listing for more details.

LISTING 24.1 A Trigger That Increments a Quantity Value in a Related Table

```
Create Trigger trgdiuOrderDetails
On dbo.[Order Details]
For Delete, Insert, Update
As
/*
    Increment the on-order quantity in Products
    based on the order quantity in the insert buffer.
    The virtual table 'inserted' contains the records that were inserted or
    updated; if empty, then this is a delete trigger.
*/
If Exists(Select * From inserted)
    Update p
        Set p.UnitsOnOrder = p.UnitsOnOrder + i.Quantity
        From inserted i, Products p
        Where p.ProductID = i.ProductID

/*
    Decrement the on-order quantity in Products
    based on the order quantity in the delete buffer.
    The virtual table 'deleted' contains the records that
    were deleted, if empty then this is an insert trigger.
*/
If Exists(Select * From deleted)
    Update p
        Set p.UnitsOnOrder = p.UnitsOnOrder - d.Quantity
        From deleted d, Products p
        Where p.ProductID = d.ProductID
```

Notice in Listing 24.1 that the name *inserted* represents the actual record(s) being inserted because it refers to a virtual table called the *insert buffer*. Whenever new records are added to a SQL Server table or updated (basically, deleted and added), the virtual table *inserted* is available in your triggers. Similarly, you can reference the delete buffer's virtual table by using the name *deleted* in an update or delete trigger to refer to the deleted record(s).

NOTE

When you enter Design view for a new trigger, the first line of the trigger code begins with the `Create Trigger` statement. When you enter Design view to edit an existing trigger, the first line of trigger code begins with `Alter Trigger`.

Obviously, triggers can be a powerful tool for Access developers. A single table can have multiple triggers in any combination of the three types: delete, insert, and update. You will find that much of the complex data manipulation that you've been doing with VBA code and form expressions in Jet-based Projects can now be done with table triggers in your Projects.

Because you must change a table record or records to test a trigger, your development might be easier if you write your trigger code in a stored procedure first. You can test and debug the stored procedure by running it from the Database window, and then paste the code into a trigger when the development is completed. However, if you're writing a trigger that relies on the deleted or inserted virtual tables, as shown earlier in this section, you can't test code that refers to those two tables within your stored procedure. You can, however, create dummy tables that represent the deleted or inserted tables for testing purposes. For example, create a table called `deleted_test` with the same structure as the table that will receive the trigger, and test your stored procedure against the dummy table. When moving the tested stored procedure code to a trigger, simply remember to rename the references from `deleted_test` to `deleted`.

Optimizing Data Access

As with your Jet-based applications, a good database design is critical to the maintenance and performance of the SQL Server databases attached to your Projects. The following sections provide some basic guidelines.

Use Standard, Normalized Database Structures

You'll probably find that multiuser applications run much more quickly on SQL Server than under a Jet-based model. Consequently, you'll probably find yourself cheating for performance less than you are used to. In a Jet-based database environment, "cheating" for performance usually means to

- Denormalize a database (flatten the structure) to produce fewer table joins that each `Select` statement will process.
- Break larger tables into several smaller tables so that fewer index pages and records are passed across the network to process a `Select` statement.

In a SQL Server model, all processing of table joins, indexed lookups, and record aggregation takes place on the server, thereby minimizing network and workstation burden. Consequently, the database design compromises that you might make to improve performance in your Jet-based applications might actually reduce the performance of your Access Project-based application.

Use Relationships

Creating relationships between associated tables generally provides the best performance, and data integrity goals are well served by the use of primary and foreign key constraints in related tables. If you are familiar with creating Jet-based relationships in the Access Relationships window, creating SQL Server relationships in a database diagram is only slightly different.

Use Stored Procedures and Triggers

Build as much of the code on the server as is practical. The term *is practical* is relative to your business needs and development style, but a good rule of thumb is that you should place code on the server that manipulates one or more *existing* database records. In contrast, you should place code in the application that validates or manipulates unsaved records or otherwise leverages the richness of the VBA language.

Storing application code on the server has three distinct benefits:

- **A single copy of the code exists.** Assume that you have a Jet-based Access application with 100 users, and a validation rule or maintenance routine requires a small change. You must change the code in a master copy of the application and then distribute the updated application to all 100 users. If the code is on the server, however, you change the code in one central place, and all users instantly begin to see the new code, with no workstation-based installation tasks required at all.
- **The code is processed on the server.** Code acting against table data will run most quickly if it's stored in the database with that same table data. Also, servers tend to have more horsepower than workstations, so even a busy server might well execute code more quickly than a single workstation with far more limited horsepower.
- **The code is closer to the database.** Logistically, it makes good sense to have the code for a database travel with that database onto backup devices, be transferred to test machines, or be replicated across an enterprise.

Balance Your Use of Indexes

Some developers take the approach that “you can’t have too many indexes.” This isn’t at all a truism in the SQL Server environment. SQL Server’s indexing models are documented quite well, and in many scenarios, an extra index will cost performance rather than improve it. If you have the opportunity, read about indexing in the SQL Server Books Online before you create your first complex Project. As a general rule, however, you will want to create an index on any field commonly used in a join, a Where clause, or an Order By clause.

Use the Most Appropriate Data Types

Select the simplest storage type for numeric data and the smallest data width for textual data that you can, as dictated by each data column’s specific needs. For example, if time accuracy

to the minute will suffice and information about seconds isn't required, using a `smalldatetime` column format over a `datetime` format will save 4 bytes per record, which can add up to a measurable savings of disk and memory space on a large table.

Always Put Timestamp Fields in Project Tables

A timestamp field provides a unique record identifier that's shared between Access and SQL Server to optimize concurrency and transaction control. Not only will the performance of deletes, inserts, and updates improve when a timestamp exists in a table, but the speed of simple tasks such as filling and refreshing a table datasheet or form also is improved.

Working with Views

You should be familiar with the concept of views just by working with Access queries. In an Access Project, a *view* is a SQL statement saved to the server that selects, filters, or groups data from one or more tables. Views have various uses in your Project:

- **They simplify the way users look at data.** If your application users are allowed to create their own Access reports, work with data in a datasheet window, or perform ad hoc queries from Excel or a query tool, a view can provide a way to customize the columns or rows exposed to various users. For example, you could create different views for different user types, based on security needs or aptitude. Views are a great way to summarize data into specific subsets without having to teach users how to write SQL. Or, you could create different views as record sources for different forms to simplify application maintenance.
- **They can be used to transfer or transform data.** Sometimes data is moved from one database to another, as in the daily routines that summarize data warehouse information into its related data mart for use by executive query and reporting systems. In other cases, data must be rearranged for download into other products or formats, as in the example of a scheduled event that extracts server data into delimited files for use by Excel or Word. In both scenarios, a view is the best tool for grouping specific columns and records into an appropriate format.
- **They can help with security.** Views are often used to enforce a security model. User permissions frequently are removed from server tables so that users can't connect to server tables and edit the data directly. Then, various views are created to give specific user groups (known as *roles*) access to certain rows and columns as their needs require. Security is added to each view to restrict access to the targeted user group. To prevent users from modifying or seeing a view's SQL statement, you can encrypt the view to lock everyone (including the creator/owner) out of its design.

NOTE

Access and SQL Server documentation use the terms *query* and *view* as mostly interchangeable. For example, the Access help topic “Designing Queries” discusses using the Query Designer to create server views.

When you design a view, bear in mind these strengths and limitations:

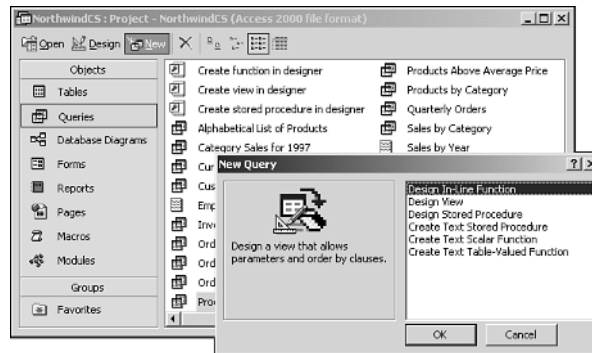
- A view can’t have the same name as a table in the same project.
- You can include tables or other views in a view’s definition. In fact, SQL Server is so powerful that you can nest views up to 32 levels deep.
- Because views (before SQL Server 2000) can’t contain the `Order By` clause, they aren’t useful for sorting records.
- You can use SQL Server expressions and functions only in the view syntax; you can’t use any Access features or VBA expressions or functions.
- Most views you create that join multiple tables aren’t updateable in Access datasheets. You generally have to create a form to update data in a multitable view because the form allows you to specify which index to use to make the view updateable (done via the form’s Unique Table property). Views that use `Distinct`, `Group By`, `Top`, or `Union` can’t be made updateable.

TIP

You can add the `Distinct` keyword to a single table view to change it from updateable to non-updateable if you want users to have read-only views of the results.

To create a new view, follow these steps:

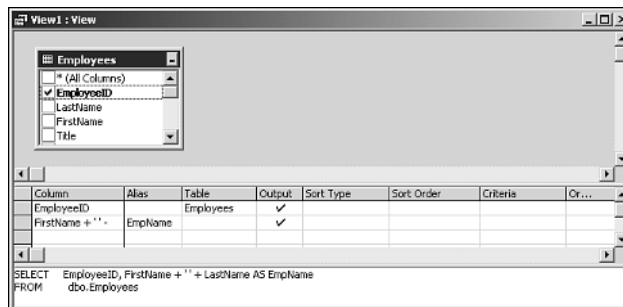
1. Select Queries in the Database window and click the New button. The New Query dialog appears (see Figure 24.16).
2. Select Design View and click OK.
3. With the View design window open, click the Show Table toolbar button. Drag a table or view from the Show Table dialog to the design window.
4. Select the column(s) to display in the design grid.

**FIGURE 24.16**

New query options exist for Access 2002 and SQL Server 2000.

- For each column in the grid, set the appropriate attributes, such as Alias and Criteria. You can enter a valid SQL Server expression, such as `Employees.FirstName + ' ' + Employees.LastName` for the Column attribute (see the Invoices view in the NorthwindCS sample database for examples).
- To view the SQL syntax for a view under construction, click the SQL toolbar button.
- Before saving your view, click the Verify SQL Syntax toolbar button to have the syntax reviewed for flagrant syntax errors.
- Save the view—you must save it before it can be run—and check the results by clicking the Datasheet View toolbar button.

Figure 24.17 shows a simple view in the design window.

**FIGURE 24.17**

Creating a simple view.

To create a multitable view in the View design window, repeat the steps earlier in this section, starting with step 2. If proper relationships exist, the design window will show the default join lines between the tables, and you won't need to do any manual table joins. Because SQL Server views can be complex and difficult to debug, familiarize yourself with the process of using the Access query designer to create multitable queries before you begin learning how to build SQL Server views.

Working with Stored Procedures

If you've been writing VBA functions and event procedures as you read this book, you understand the concept of creating a block of code that achieves a specific result and may include flow-control statements and accept or return argument values. This concept describes SQL Server *stored procedures*, which are named, saved code routines that perform operations on server data. They're written in the Transact-SQL language (T-SQL for short) and have these primary attributes:

- It's compiled at the time it's saved on the server so that it executes quickly.
- Because it's a database object, you can apply security permissions/restrictions to it through SQL Server's security model.
- It can accept parameters and can return values (variables and recordsets) to the user interface or calling routine.
- It can include `If . . Else` logic to control code flow conditionally.
- It can contain multiple SQL statements that operate on data.
- It can call other stored procedures.

Neither the Access interface nor the SQL Server compiler are very helpful or forgiving when you write stored procedure code. You won't find the delightful Auto List Members or Auto Quick Info features of the VBE in the Stored Procedure design window. You won't think that syntax errors generated by the compiler are particularly helpful. You also won't be able to use the Windows Undo feature! And you will dislike the lack of debugging tools for stored procedures.

So, how do you write a stored procedure? The same way you eat an elephant—one bite at a time. You will have the best success at server-side coding if you break your stored procedure development into small tasks, and then do your development one task at a time. Create your primary stored procedure, define the parameters, and outline the structure. Then, create another stored procedure to serve as your temporary workspace. Write code in the temporary workspace, compiling and checking syntax frequently. When you've completed the coding on the task at hand and tested your work, move the code to the real stored procedure, clear out your workspace, and continue coding the next task. In this way, you can build a larger stored procedure a few lines at a time.

Creating Stored Procedures

Access 2002 and SQL Server 2000 let you create stored procedures in much the same way as views—by using a Stored Procedure design window. Select Design Stored Procedure from the New Query dialog.

To create a stored procedure with all versions of SQL Server, follow these steps:

1. In the Database window, select Queries and click the New button.
2. Select Create Text Stored Procedure and click OK. The Stored Procedure design window appears with a code template inside (see Figure 24.19).

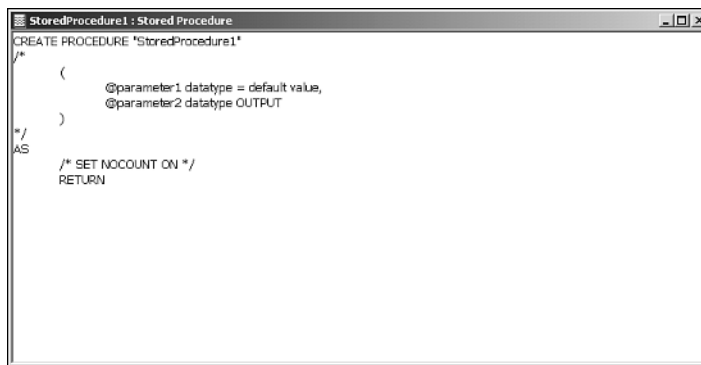


FIGURE 24.18

The Stored Procedure text editor isn't as robust as the Visual Basic Editor.

3. Change the suggested name on the first line to the name you want to use.
4. Enter one or more T-SQL commands and click the Save toolbar button to compile your code and check the syntax. T-SQL commands include SQL statements such as `Select`, SQL Server expressions, and flow-control statements such as `If`.
5. Click the Datasheet View toolbar button to execute the stored procedure. If it returns a recordset, a datasheet is displayed; if it doesn't, Access provides a message that the stored procedure did or didn't complete execution successfully.
6. Repeat steps 3 and 4 until the stored procedure is complete.

If the stored procedure returns a recordset, you can display that recordset as an Access datasheet and use it as the record source for forms and reports. If you want a stored procedure to return records, place the `Select` statement directly after `Create Procedure "procname"` `AS`. The following simple stored procedure returns a recordset:

PART IV

```
Create Procedure prcProdList
AS
    SELECT Products.ProductName, Products.UnitPrice
    FROM Products
    ORDER BY Products.ProductName ASC
```

If you want to pass information to a stored procedure or receive values back, you create parameters on the procedure. A stored procedure can have many parameters, with which a stored procedure can pass values back to your ADO code or to a calling stored procedure.

To define parameters at the top of the procedure, prefix each one with the @ character (as in @prmProdName) and define the data type. Refer to Figure 24.19 to see how Access inserts placeholders for parameters in the stored procedure code template.

You can use an input parameter in stored procedure code anywhere a literal value might appear; it can't be used to replace code syntax or an object name (such as a table name). The following stored procedure accepts a parameter, uses it in the Where clause, and displays a recordset with the price for a specified product:

```
Create Procedure prcProdPrice
    @prmProdName varchar(40)
AS
    SELECT Products.ProductName, Products.UnitPrice
    FROM Products
    WHERE Products.ProductName = @prmProdName
```

When you run a stored procedure that has one or more parameters from the Database window, Access displays the Enter Parameter Value dialog once for each parameter (see Figure 24.19). When running a stored procedure from code or a form, the parameter values must be passed in as form properties or ADO properties, respectively.

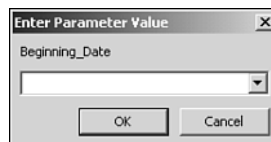


FIGURE 24.19

Enter a stored procedure parameter.

A stored procedure can return values instead of, or in addition to, recordsets. These values, known as *output parameters*, are declared with the Output keyword. Like VBA functions, stored procedures can also have one return value; your code usually sets this value to indicate an error condition or successful completion.

The stored procedure in Listing 24.2 contains one input parameter and one output parameter and sets the return value. It accepts a product name as the input parameter, returns the price for the specified product as the output parameter, and sets the return value to 0 if successful or -1 if the product wasn't found.

LISTING 24.2 A Stored Procedure with an Output Parameter and Return Value

```
Create Procedure prcProdPrice
    @prmProdName varchar(40),
    @rprmProdPrice money OUTPUT
AS
SELECT @rprmProdPrice = Products.UnitPrice
FROM Products
WHERE Products.ProductName = @prmProdName
IF (@rprmProdPrice > 0)
    RETURN 0
ELSE
    RETURN -1
```

NOTE

The stored procedure in Listing 24.2 presumes that the `Select` statement will return only one record, which is true in this case because the criteria uniquely identifies a single product record. However, in many types of stored procedures, the `Select` statement might return one or more records. To work in such an environment, T-SQL provides *multirecord cursors* (roughly equivalent to VBA recordsets). A description of cursors is beyond the scope of this chapter.

To use the stored procedure in Listing 24.2, some ADO code must be written to set and test the parameters, because the procedure doesn't return records to the user interface. Listing 24.3 shows how such code would look.

LISTING 24.3 Setting and Testing a Stored Procedure's Parameters

```
Private Sub PrcTest()
    Dim cmd As New ADODB.Command
    Dim cnn As New ADODB.Connection
    Dim fld As ADODB.Field
    Dim prmIn As Parameter
    Dim prmOut As Parameter
    Dim prmRet As Parameter
```

LISTING 24.3 Continued

```

Dim rst      As New ADODB.Recordset
Dim varProd As Variant

varProd = Trim(TextBox("Enter product name:"))

cnn.ConnectionString = CurrentProject.BaseConnectionString
cnn.Open
cmd.ActiveConnection = cnn
cmd.CommandType = adCmdStoredProc
cmd.CommandText = "prcProdPrice"

' Bind parameters; this must occur in the order they are declared
Set prmRet = cmd.CreateParameter("Return", adInteger, adParamReturnValue)
cmd.Parameters.Append prmRet
Set prmIn = cmd.CreateParameter("Input", adVarChar, adParamInput, 40)
cmd.Parameters.Append prmIn
prmIn.Value = varProd
Set prmOut = cmd.CreateParameter("Output", adCurrency, adParamOutput)
cmd.Parameters.Append prmOut
cmd.Execute

Debug.Print "Return value: " & cmd.Parameters(0)
Debug.Print "Product price: " & cmd.Parameters(2)

cnn.Close

End Sub

```

Comparing Stored Procedure and Access Syntax

Access and SQL Server don't use identical syntax elements (such as operators) to achieve the same result. You have to learn the differences between Access and SQL Server syntax to work with stored procedures. Table 24.6 shows the difference in certain characters between a Jet-based application and an Access Project.

TABLE 24.6 Comparing Access and SQL Server Syntax Elements

<i>Item</i>	<i>Access</i>	<i>SQL Server</i>
Bitwise And	And	&
Bitwise Exclusive Or	XOr	^
Bitwise Or	Or	
Concatenation	&	+

TABLE 24.6 Continued

<i>Item</i>	<i>Access</i>	<i>SQL Server</i>
Date delimiter	#date#	'date'
False	No/False/0	0
Modulus	mod	%
Multiple-character wildcard	*	%
Not Equal To	<>	<> or !=
Not Greater Than	Not >	!>
Not in list	!	^
Not Less Than	Not <	!<
Single character wildcard	?	_
String delimiter	'string' or "string"	'string'
True	Yes/True/-1	1

The Transact-SQL language used to write stored procedures has many functions similar in purpose to functions you may already use or know about in VBA, but with a different syntax. Table 24.7 lists T-SQL functions that you can use in stored procedures and their corresponding VBA functions.

TABLE 24.7 VBA and T-SQL Functions Compared

<i>VBA</i>	<i>Transact-SQL</i>
asc(x)	ascii(x)
ccur(x)	convert(money,x)
cdbl(x)	convert(float,x)
chr\$(x)	char(x)
cint(x)	convert(smallint,x)
clng(x)	convert(int,x)
csng(x)	convert(real,x)
cstr(x)	convert(varchar,x)
cvdate(x)	convert(datetime,x)
date(x)	convert(datetime,convert(varchar,getdate(x)))
dateadd(datepart, x, y)	dateadd(datepart, x, y)
datediff(datepart, x, y)	datediff(datepart, x, y)

TABLE 24.7 Continued

<i>VBA</i>	<i>Transact-SQL</i>
<code>datepart(datepart, x)</code>	<code>datepart(datepart, x)</code>
<code>day(x)</code>	<code>datepart(dd,x)</code>
<code>hour(x)</code>	<code>datepart(hh,x)</code>
<code>int(x)</code>	<code>floor(x)</code>
<code>lcase\$(x)</code>	<code>lower(x)</code>
<code>len(x)</code>	<code>datalength(x)</code>
<code>ltrim\$(x)</code>	<code>ltrim(x)</code>
<code>mid\$(x,y,z)</code>	<code>substring(x,y,z)</code>
<code>minute(x)</code>	<code>datepart(mi,x)</code>
<code>month(x)</code>	<code>datepart(mm,x)</code>
<code>now(x)</code>	<code>getdate(x)</code>
<code>right\$(x,y)</code>	<code>right(x,y)</code>
<code>rtrim\$(x)</code>	<code>rtrim(x)</code>
<code>second(x)</code>	<code>datepart(ss,x)</code>
<code>sgn(x)</code>	<code>sign(x)</code>
<code>space\$(x)</code>	<code>space(x)</code>
<code>str\$(x)</code>	<code>str(x)</code>
<code>ucase\$(x)</code>	<code>upper(x)</code>
<code>weekday(x)</code>	<code>datepart(dw,x)</code>
<code>year(x)</code>	<code>datepart(yy,x)</code>

Summary

You can see from this chapter that developing applications with SQL Server and an ADP is a whole new ball game from using Jet and .mdbs. But if you keep at it, it will become just as familiar to you.

The whole topic of using Microsoft Access ADPs with SQL Server is a book by itself. In fact, Mary Chipman and Andy Baron have written one titled *Microsoft Access Developer's Guide to SQL Server*, also published by Sams. I can't recommend it enough to fill any ADP and SQL Server gaps you may have.

Check out these chapters on using ADO and client/server information:

- In Chapter 5, “Introducing ActiveX Data Objects,” you learn about the syntax necessary to take advantage of ADO.
- Chapter 23, “Moving Workgroup Applications to Client/Server,” covers more issues about creating applications by using client/server applications with Access as the front end.

Adding Finishing Touches

PART V

IN THIS PART

- 25 Startup Checking System Routines
Using ADO 829**
- 26 Creating Maintenance Routines 861**

Startup Checking System Routines Using ADO

CHAPTER

25

IN THIS CHAPTER

- Performing Startup System Checks 830
- Setting and Retrieving System Settings 835
- Notifying and Logging Users Out of an Application 836
- Testing Table Links at Startup 843
- Testing and Repairing Corrupted Jet Back-End Databases 852
- Checking and Notifying Users of a New Version 857

All of Access's powerful features really don't mean much unless they're put to practical use. This also can be said of all the books concerning development in the Access environment. One goal of this book is to give you practical examples as well as advanced concepts.

This chapter also gives you routines that you can add to your systems to save time and money in support and maintenance. When starting your application and creating useful routines, ask yourself these questions:

- What's a graceful way to keep track of the files needed for linking and unlinking tables?

NOTE

In Access versions before Office 95, the linking and unlinking of tables from other databases was referred to as *attaching* and *unattaching*, which can be somewhat confusing when you look at someone else's code.

- How often have you had to hunt down everybody on a network to ask them to log out?
- How can a corrupted back-end database be repaired automatically from within your system?
- How do you notify users that a new version of the application is available and force them to upgrade?

This chapter answers these questions and more.

NOTE

The routines described in this chapter are to be used with native Access back ends and might not work with SQL Server. For information on using Access as a front end for SQL Server, refer to Chapter 24, "Developing SQL Server Projects Using ADPs."

Performing Startup System Checks

When you're creating applications in the real world, it's important to make them the most robust systems possible. To do so, you must start when the application is first loading.

It's highly recommended that you perform startup checking at various levels, especially regarding the back-end database and its connections. Many problems can occur for which you can provide a way to resolve automatically or at least have the system back users out gracefully.

Some problems that can occur, and for which no simple programming solution exists, happen when Windows causes an application error or when the front-end database is corrupt.

If Windows causes an application error (commonly referred to as a General Protection Fault, or GPF), it usually displays a message box and returns you to the main desktop (see Figure 25.1). No more having to shut down Windows and reboot!

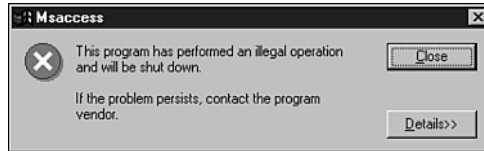


FIGURE 25.1

The Windows 9x/NT version of the GPF doesn't require rebooting Windows.

If the front end becomes corrupted, the user or administrator must repair the database. To do so from Access, follow these steps:

1. Open Access without specifying a database (click Cancel in the Open Database dialog).
2. From the Tools menu, choose Database Utilities and then Repair Database.
3. In the Compact and Repair Database dialog, select the front-end .mdb file to repair and click Compact.

With the back end, it's important to trap all situations that can occur. This chapter can't cover every situation, but it provides solutions to some of the major ones. First, however, look at the routine that does the startup checking, `ap_AppInit()`, in Listing 25.1. The code is broken up with comments, and each code segment is explained throughout the rest of this chapter. All routines shown in this chapter can be found in the `modGlobalUtilities` module of `VideoApp(ADO).mdb`, which can be found on this book's Web page at www.sampublishing.com.

LISTING 25.1 VideoApp(ADO).mdb: The Main Startup System Checking Routine

```
Function ap_AppInit()
    Dim cnnLocal As New ADODB.Connection
    Dim rstSharedTables As New ADODB.Recordset
    Dim lngCurrError As Long, strCurrError As String

    DoEvents
    DoCmd.Echo True, "Checking Connections..."

    flgLeaveApplication = False
    Set cnnLocal = CurrentProject.Connection
```

LISTING 25.1 Continued

```
'-- Section 1: Initialize database properties
ap_InitializeDBProps cnnLocal

'-- Section 2: User requested to logout, quit the application
If ap_LogOutCheck(pstrBackEndPath) Then
    Beep
    MsgBox "Maintenance is being performed on the backend" & vbCrLf _
        & vbCrLf & "All users are requested to log out at this time.", _
        vbOKOnly + vbCritical, "Logging Out for Maintenance"
    Application.Quit
    Exit Function
End If

'-- Section 3: Open the table containing the list of linked tables
' Open employee table.
rstSharedTables.Open "tblSharedTables", cnnLocal, adOpenStatic

On Error Resume Next

rstSharedTables.MoveLast

Dim rstTestTable As New ADODB.Recordset
Dim strTableName As String

strTableName = rstSharedTables!TableName
rstTestTable.Open strTableName, cnnLocal, adOpenStatic

lngCurrError = Err.Number

If lngCurrError = 0 Then
    Dim varTest As Variant
    varTest = rstTestTable(0).Name
    lngCurrError = Err.Number
End If
strCurrError = Err.Description

Do Until lngCurrError = 0
    On Error GoTo Error_ap_App_Init
    Select Case lngCurrError
        Case apErrInvalidSQL
            '-- Section 4: If the Data MDB is found in the App Directory,
            '-- link the files.
            If Dir(pstrAppPath & "\" & pstrBackEndName) = pstrBackEndName Then
                ap_LinkTables rstSharedTables, pstrAppPath & "\" & pstrBackEndName
```

LISTING 25.1 Continued

```

        pstrBackEndPath = pstrAppPath & "\"
    Else
        '-- Section 5: Allow the user to locate the BackEnd MDB
        If Not ap_LocateBackend(rstSharedTables, _
            strCurrError) Then
            flgLeaveApplication = True
        End If
    End If
Case apErrDBCorrupted1
    '-- Section 6: Backend Corrupted. Repair?
    Beep
    If MsgBox("The Backend Database is Corrupted." & vbCrLf & _
        vbCrLf & "Would you like to log users out and " & _
        " attempt to compact/repair it?", vbYesNo + vbCritical, _
        "Corrupted Backend!") = vbYes Then
        DoCmd.OpenForm "ap_CompactDatabase", acForm
    Else
        flgLeaveApplication = True
    End If
End Select
 '-- Section 7: Leave the application if requested
If flgLeaveApplication Then
    Application.Quit
    Exit Function
End If
On Error Resume Next
 '-- Section 8: Let's try and open the first table again.
rstTestTable.Open strTableName, cnnLocal, adOpenStatic
lngCurrError = Err.Number
strCurrError = Err.Description
Loop

On Error GoTo Error_ap_App_Init

 '-- Section 9: Check the version of the front end,
 '--             and point to a new one if necessary.
If Not CheckFEVersion() Then
    Application.Quit
End If

 '-- Section 10: Save the BackEnd path in the BackEndPath property
 '--             for future use.
ap_SetDatabaseProp "LastBackEndPath", pstrBackEndPath

```

LISTING 25.1 Continued

```

'-- Check for locally logged errors
ap_ErrorCheckLocal

'-- Check for Replicated Tables
ap_CheckReplicatedTables

'-- Turn on the Monitor Form
DoCmd.OpenForm "UserLogOutMonitor", , , , , acHidden

DoCmd.Close acForm, "SplashScreen"
DoCmd.Echo True
rstSharedTables.Close

Exit_ap_App_Init:
Exit Function

Error_ap_App_Init:
Beep
MsgBox "The following error occurred: " & Err.Description & vbCrLf & _
    vbCrLf & "The system will now be closed down!"
Application.Quit

End Function

```

The first order of business accomplished by `ap_AppInit` is initializing specific variables that can be used elsewhere in the application. The following routine performs this task:

```

Sub ap_InitializeDBProps(cnnLocal As ADODB.Connection)
    Dim rstDBProps As New ADODB.Recordset
    rstDBProps.Open "ztblDatabaseProperties", cnnLocal
    pstrAppPath = CurrentProject.Path
    pstrBackEndName = rstDBProps!BackEndName
    pstrBackEndPath = rstDBProps!LastBackEndPath
    rstDBProps.Close
End Sub

```

Because you can't manipulate Access custom database properties in ADO, the following variables are placed in a table named `ztblDatabaseProperties`, a single-entry table that contains the various system properties you want to track:

- The front-end path to the variable `pstrAppPath`
- The back-end name to `pstrBackEndName`
- The last good back-end path to `pstrBackEndPath`

Figure 25.2 shows the values stored in `ztblDatabaseProperties`.

NOTE

Although using `CurrentProject.Path` to get the current database's path is quite simple, for consistency it's stored in a variable.

BackEndName	LastBackEndPath	FrontEndVersion	ExportTypeDefault
VideoDat.mdb	C:\Books\PwrPrg2002\AppCD\Examples\	1.0	HTML File

FIGURE 25.2

You can track the information stored in this table through the Windows Registry.

Setting and Retrieving System Settings

The routines in Listing 25.2 are `ap_SetDatabaseProp` and `ap_GetDatabaseProp`.

LISTING 25.2 VideoApp(ADO).mdb: The Main Startup System Checking Routines

```
Sub ap_SetDatabaseProp(strPropertyName As String, varValue As Variant)
    Dim rstDBProps As New ADODB.Recordset
    rstDBProps.Open "ztblDatabaseProperties", CurrentProject.Connection, _
        adOpenDynamic, adLockOptimistic
    rstDBProps(strPropertyName) = varValue
    rstDBProps.Update
    rstDBProps.Close
End Sub

Function ap_GetDatabaseProp(strPropertyName As String) As Variant
    Dim rstDBProps As New ADODB.Recordset
    rstDBProps.Open "ztblDatabaseProperties", CurrentProject.Connection, _
        adOpenStatic
    ap_GetDatabaseProp = rstDBProps(strPropertyName)
    rstDBProps.Close
End Function
```

Notice between these routines that when storing an updated value into the `ztblDatabaseProperties` table in `ap_SetDatabaseProp`, the recordset is opened dynamically, whereas in `ap_GetDatabaseProp`, the recordset is opened statically.

Next, the `ap_AppInit` routine performs the check to notify users whether they have to leave the application.

Notifying and Logging Users Out of an Application

When dealing with live systems, you need a way to log and keep people out of an application's back end. As with most system problems, several solutions exist.

One solution to keeping a property on the back end is to keep track of whether users are to be logged out. With this solution, you have to use the back-end database rather than the front-end database. You can examine this property from VBA and act accordingly.

A problem with using this “property on the back-end database” method is that if you use it for logging someone out and the back-end database is corrupted, you get an error. The solution is to create a file in the back-end folder that the user's front end looks at and then acts accordingly. The file created, `LogOut.flg`, has its attribute flag set as Hidden so that there isn't a problem with users deleting the file arbitrarily to get back into the system.

Keeping Users Out at Startup Time

Section 2 of `ap_AppInit()` calls the function `ap_CheckLogOut()`, passing the variable `pstrBackEndPath`, which contains the last known path to the back end:

```
'-- Section 2: User requested to logout, quit the application
If ap_LogOutCheck(pstrBackEndPath) Then
    Beep
    MsgBox "Maintenance is being performed on the backend" & vbCrLf _
        & vbCrLf & "All users are requested to log out at this time.", _
        vbOKOnly + vbCritical, "Logging Out for Maintenance"
    Application.Quit
    Exit Function
End If
```

Figure 25.3 shows the message that occurs when users need to stay out.

You might wonder why the application passes the last known path rather than checks the current path from one of the linked tables. If a problem is encountered with the back end or someone is already trying to get everybody out, it's not necessary to make users wait to relink all the tables just to find out they need to leave the system.

The following code for `ap_LogOutCheck()`, in the `modGlobalUtilities` module, performs the check whether the user needs to be logged out. Notice that the function's error handling just skips to the next line, thereby passing back the false value for `ap_LogOutCheck()`.

```
Function ap_LogOutCheck(strBackEndPath As String) As Integer
    On Error Resume Next
    ap_CheckLogOut = Dir(strBackEndPath & "\LogOut.FLG", vbHidden) = "LogOut.FLG"
End Function
```

**FIGURE 25.3**

This log out message for the World Wide Video Application occurs on entry into the application.

TIP

Like variables, functions default to zero when Dim'd as an integer. Always explicitly declare your functions and variables. Otherwise, they're implicitly declared as Variant, which is slower to process.

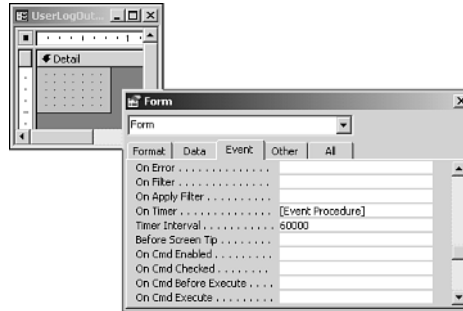
Notice that this function uses the `Dir()` function with the `vbHidden` attribute flag designated. This allows you to look for a file even when it's hidden. The average user can't see it through Windows Explorer.

Logging Users Out in the Middle of the Application

The preceding section describes how to log users out of the system when first starting up. Sometimes you need to log users out when they're in the middle of the application. This method requires a hidden form that contains nothing but a timer event procedure. You can use this procedure with another form if you have one that you keep open at all times.

The form used is `UserLogOutMonitor`. Figure 25.4 shows the form with its property sheet.

The `Timer Interval` property lets you specify an interval of time in milliseconds. Therefore, the 60000 shown in Figure 25.4 is 60 seconds. So every 60 seconds, Access triggers the supplied `OnTimer` event.

**FIGURE 25.4**

This form plays a vital role in system maintenance for the World Wide Video application.

NOTE

Because the LogOut.flg file is in the same folder as the back end, it might be that checking the network once a minute causes too much network traffic. You have to work with this property to see what timing works for your situation.

Listing 25.3 shows the code for the OnTimer event for the UserLogOutMonitor form.

LISTING 25.3 VideoApp(ADO).mdb: Checking Whether to Log Out the User

```
Private Sub Form_Timer()
    If ap_LogOutCheck(ap_GetDatabaseProp("LastBackEndpath")) Then
        If Not ap_FormIsOpen("UserLogOutCountdown") Then
            DoCmd.OpenForm "UserLogOutCountdown"
        End If
    Else
        If ap_FormIsOpen("UserLogOutCountdown") Then
            DoCmd.Close acForm, "UserLogOutCountdown"
            Beep
            MsgBox "The Logout Countdown has been canceled." & vbCrLf & vbCrLf & _
                "You may go on with your work.", vbInformation, "Logout Canceled"
        End If
    End If
End Sub
```

The Form_Timer procedure performs these steps:

1. It tests to see whether the LogOut.flg file is found with the function ap_LogOutCheck().

- If the file is found, the routine checks whether the UserLogoutCountdown form is already opened. If UserLogoutCountdown isn't opened, the routine opens it. (This form is examined in a moment.)

If LogOut.flg isn't found and the UserLogoutCountdown form is opened, the routine cancels the countdown.

The last part of step 2 is in case the administrator who told people to log out changes his mind. You can see the message for this cancellation in Figure 25.5.

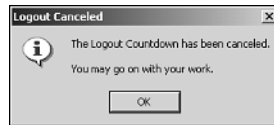


FIGURE 25.5

Watch the status of the logout file closely to be sure to cancel tasks as soon as possible.

Only two events are attached to the UserLogoutCountdown form. The code in UserLogoutCountdown keeps track of a countdown, starting at five minutes. The form also informs users that they'll be logged out of the system at the end of the countdown. This time span allows users to finish any work they have to do before time is up. When time is up, the system shuts them down, no matter where they are at this time.

UserLogoutCountdown is set up as a pop-up non-modal form, which lets users clean up without the countdown form getting lost beneath other windows. Figure 25.6 shows the form in Design view.

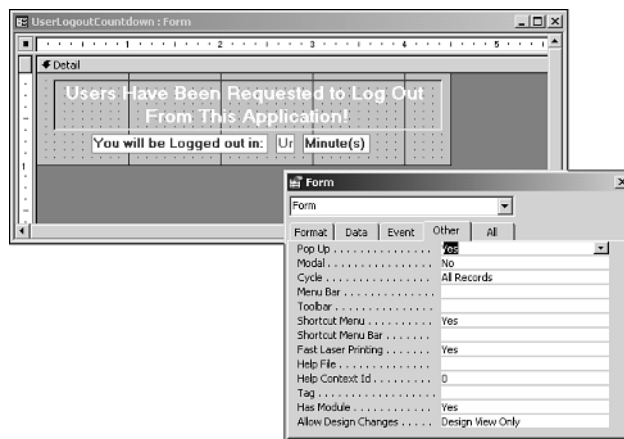


FIGURE 25.6

The Pop Up property for UserLogoutCountdown is set to Yes, whereas the Modal property is set to No.

CAUTION

Be careful of setting your form's Pop Up and Modal properties while removing all Windows control buttons. You could end with a difficult-to-close form. Always be sure to leave a way out of a form, either through a Close command button or the window's close (X) button. The other way to close the form is to press Ctrl+G to open the Immediate window, and then type **DoCmd.Close**.

Although it's not shown in Figure 25.6, this form's Timer Interval property is set to 60000 milliseconds, just as it was on the UserLogoutMonitor form. This property drives the countdown process and calls the OnTimer event procedure that contains the routine. Listing 25.4 shows the declarations for the form module, the Form_Open event procedure, and then the Form_Timer routine.

LISTING 25.4 VideoApp(ADO).mdb: Counting Down Before Logging Out the User

```
'-- Declarations Section
Option Compare Database
Dim intCountDown As Integer

Private Sub Form_Open(Cancel As Integer)
    intCountDown = 5
End Sub

Private Sub Form_Timer()
    On Error GoTo Error_Form_Timer
    intCountDown = intCountDown - 1
    If intCountDown <= 0 Then
        Application.Quit
    Else
        Beep
        Me!txtMinutesToGo = intCountDown
    End If
End Sub

Exit_Form_Timer:
Exit Sub

Error_Form_Timer:
MsgBox "An error has occurred!" & vbCrLf & vbCrLf _
    & "The application will now quit.", vbCritical, "Timing Error"
Application.Quit
End Sub
```

When the form is opened, the following events occur:

1. The `intCountDown` variable is initialized to 5 (minutes) in the `Form_Open` subroutine.
2. Every minute the `Form_Timer` subroutine is called, 1 is subtracted from `intCountDown`.
3. If the five minutes are up, the application quits; otherwise, a beep sounds, and the unbound `txtMinutesToGo` TextBox control is updated for display.

CAUTION

When you use timer events, the system can give an error when transactions are used. If you're performing new file tasks inside the timer event, your work could conflict with the transaction processing. You might need to close the `UserLogOutMonitor` form when performing transactions.

Setting the Flag File to Log Users Out of the Back End

Setting the flag file isn't much tougher than looking for it. You can find the code for this routine on the `cmdLogInOut` command button on the `ap_SystemUtilities` form, attached to the `OnClick` event. The `ap_SystemUtilities` form is used for handling various system functions and should be set up with some kind of security. The code also calls some public subroutines that are used to deal with creating and removing the flag file, as Listing 25.5 shows.

LISTING 25.5 VideoApp(ADO).mdb: Creating a Flag File for Logging Users Out

```
Private Sub cmdLogInOut_Click()
    On Error GoTo Error_cmdLogInOut_Click

    pstrBackEndPath = ap_GetDatabaseProp("LastBackEndPath")

    If Dir(pstrBackEndPath & "\LogOut.FLG", vbHidden) = _
        "LogOut.FLG" Then
        ap_LogOutRemove
    Else
        ap_LogOutCreate
    End If
    DisplayLogInOut
Exit_cmdLogInOut_Click:
Exit Sub
```

LISTING 25.5 Continued

```
Error_cmdLogInOut_Click:
    MsgBox Err.Description
    Resume Exit_cmdLogInOut_Click

End Sub

Public Sub ap_LogOutRemove()
    On Error Resume Next
    SetAttr pstrBackEndPath & "LogOut.FLG", vbNormal
    Kill pstrBackEndPath & "LogOut.FLG"
End Sub

Public Sub ap_LogOutCreate()
    On Error Resume Next
    '-- Create flag file
    Open pstrBackEndPath & "LogOut.FLG" For Output Shared As #1
    Close #1
    SetAttr pstrBackEndPath & "LogOut.FLG", vbHidden
End Sub
```

The same button is used to toggle between creating the flag file and removing it. If the `Dir()` function comes back with the `LogOut.flg` file existing, the `ap_LogOutRemove` routine is called with the `Kill` command. Otherwise, you call the `ap_LogOutCreate` routine, which uses a good old-fashioned `Open` command from the ancient BASIC days. The `SetAttr` command toggles the visibility of the file before the file is deleted and after it's created.

TIP

Use `Dir()` to create a list of files in a folder. By passing a file "skeleton" with wild cards and a path, you can repeat the call to `Dir()` until it returns the empty string.

To view the syntax of the `Dir()` function in the Object Browser, follow these steps:

1. While in any module, press F2.
2. Click the Object Browser toolbar button, or choose Object Browser from the View menu.
3. Select VBA - VBA from the Project/Library combo box.
4. Select FileSystem from the Classes list.
5. Select Dir from the Members list.

You now see the syntax for the `Dir()` function. You can click the question mark button for further help.

The `DisplayLogInOut` subroutine checks whether the `LogOut.flg` file exists. If it does, it toggles the `Caption` property on the `cmdLogInOut` command button, as Listing 25.6 shows.

LISTING 25.6 VideoApp.mdb: Changing a Button's Caption

```
Sub DisplayLogInOut()
    On Error GoTo Error_DisplayLogInOut

    If Dir(ap_GetDatabaseProp("LastBackEndPath") & _
        "\LogOut.FLG", vbHidden) = "LogOut.FLG" Then
        Me!cmdLogInOut.Caption = "&Allow Users Into System"
    Else
        Me!cmdLogInOut.Caption = "&Log Users Out Of System"
    End If

Exit_DisplayLogInOut:
    Exit Sub

Error_DisplayLogInOut:
    MsgBox Err.Description
    Resume Exit_DisplayLogInOut

End Sub
```

The routines for logging out users are used throughout the rest of this chapter.

Testing Table Links at Startup

As you continue to examine the sections of the `ap_AppInit()` function from Listing 25.1, now look at section 3, which opens and tests the connection for a table located in the back end, with the connection being a link in the front end (see Listing 25.7.)

LISTING 25.7 VideoApp(ADO).mdb: Testing the Links

```
'-- Section 3: Open the table containing the list of linked tables
' Open employee table.
rstSharedTables.Open "tblSharedTables", cnnLocal, adOpenStatic

On Error Resume Next

rstSharedTables.MoveLast
```


LISTING 25.7 Continued

```
Dim rstTestTable As New ADODB.Recordset
Dim strTableName As String

strTableName = rstSharedTables!TableName
rstTestTable.Open strTableName, cnnLocal, adOpenStatic

lngCurrError = Err.Number

If lngCurrError = 0 Then
    Dim varTest As Variant
    varTest = rstTestTable(0).Name
    lngCurrError = Err.Number
End If
strCurrError = Err.Description

Do Until lngCurrError = 0
    On Error GoTo Error_ap_App_Init
    Select Case lngCurrError
        Case apErrInvalidSQL
```

In section 3, the link is tested and opened at the field level. Opening the link might not trigger an error, but accessing the field will. This code performs these actions:

1. The `tblSharedTables` table is opened and assigned to `rstSharedTables`, which is a static type ADO recordset data type variable.

NOTE

`tblSharedTables` contains two Text-type fields: `TableName` and `ExportFileName`. Figure 25.7 shows the table in datasheet mode.

2. The code sets error handling to ignore errors and just moves to the next line by using the `On Error Resume Next` statement.
3. It tries to open the last table in `rstSharedTables` as a static recordset, and then stores the error codes (if any) to variables for later use.
4. If no error has occurred, the first field name in the table is read to see whether an error occurs at this level.
5. The code begins a loop that performs another test (at the bottom) as in step 3. Inside this loop are the corrections for the various errors. When one error is solved—for example, a moved back end—the loop retests and resends the code execution through the error list.



TableName	ExportFileName
tblCategories	Categories
tblCustomerRelations	CustRel
tblCustomers	Customer
tblEmployees	Employee
tblInvoices	Invoices
tblMovieStars	Mstars
tblMovieTitles	Movies
tblMovieTitlesCategories	MTCats
tblNewReleases	NewRel
tblRelationCodes	Relation
tblRentalHistory	RentHist
tblStars	Stars
tblTapes	Tapes

FIGURE 25.7

tblSharedTables is used for linking, unlinking, and exporting tables.

- The code turns on error handling again and examines the possible errors that occurred. Constants are used to represent the errors you want to look for.

NOTE

The only error code constant supplied is `apErrInvalidSQL`, which seems to cover quite a bit of ground. Whenever the database is moved, the error number stored in this constant, in the declarations section of the `modGlobalUtilities` modules, is raised. If you receive other error codes that don't get trapped, add them to the list of constants.

Linking and Unlinking Tables in a Jet Back End in the Application's Folder

In section 4, the `ap_AppInit()` function checks whether the back-end database is in the same folder as the front end and, if so, links the tables automatically:

```
'-- Section 4: If the Data MDB is found in the App Directory,
'--          link the files.
If Dir(pstrAppPath & "\" & pstrBackEndName) = pstrBackEndName Then
    ap_LinkTables rstSharedTables, pstrAppPath & "\" & pstrBackEndName
    pstrBackEndPath = pstrAppPath & "\"
Else
```

This code calls the `ap_LinkTables` routine and passes three arguments:

- `rstSharedTables`, the recordset containing the list of tables to link
- `pstrAppPath` and `pstrBackEndName`, the front-end database path concatenated to the back-end database name

Finally, this code section stores the front end's file path into the public variable `pstrBackEndPath`.

It's time to see what's done to relink to a new back end through the `ap_LinkTables` routine. Listing 25.8 shows the code for `ap_LinkTables`.

LISTING 25.8 VideoApp(ADO).mdb: Linking Tables in the Back End

```
Public Sub ap_LinkTables(rstSharedTables, strDataMDB As String)
    Dim catCurr As New ADOX.Catalog
    Dim intTotalTbls As Integer
    Dim intCurrTbl As Integer
    Dim strCurrTable As String
    On Error GoTo Err_LinkTables

    '-- Get the total number of linked tables, then display the progress meter.
    rstSharedTables.MoveLast
    intTotalTbls = rstSharedTables.RecordCount
    rstSharedTables.MoveFirst

    SysCmd acSysCmdInitMeter, "Linking Tables...", intTotalTbls

    catCurr.ActiveConnection = CurrentProject.Connection

    intCurrTbl = 1
    Do Until rstSharedTables.EOF
        '-- Update the progress meter
        SysCmd acSysCmdUpdateMeter, intCurrTbl

        '-- Attempt to open the current link
        On Error Resume Next

        strCurrTable = rstSharedTables!TableName
        catCurr.Tables.Delete strCurrTable
        catCurr.Tables.Refresh

        On Error GoTo Err_LinkTables

        Dim tblCurr As New ADOX.Table
        Set tblCurr.ParentCatalog = catCurr

        tblCurr.Name = rstSharedTables!TableName

        tblCurr.Properties("Jet OLEDB:Link Datasource") = strDataMDB
        tblCurr.Properties("Jet OLEDB:Create Link") = True
    
```

LISTING 25.8 Continued

```

tblCurr.Properties("Jet OLEDB:Remote Table Name") = _
    rstSharedTables!TableName

catCurr.Tables.Append tblCurr

Set tblCurr = Nothing

rstSharedTables.MoveNext
intCurrTbl = intCurrTbl + 1

Loop

Exit_LinkTables:
    SysCmd acSysCmdRemoveMeter
    Exit Sub

Err_LinkTables:
    Resume Exit_LinkTables

End Sub

```

The ap_LinkTables routine accomplishes these tasks:

1. It declares a catalog variable, and then moves to the last record in the shared tables recordset to get an accurate record count. After grabbing the record count, the code moves back to the first record. A status bar is displayed. Then the catalog's (catCurr) ActiveConnection property is set to the current project's connection.

```

Dim catCurr As New ADOX.Catalog
Dim intTotalTbls As Integer
Dim intCurrTbl As Integer
Dim strCurrTable As String
On Error GoTo Err_LinkTables

'-- Get the total number of linked tables, then display the progress meter
rstSharedTables.MoveLast
intTotalTbls = rstSharedTables.RecordCount
rstSharedTables.MoveFirst

SysCmd acSysCmdInitMeter, "Linking Tables....", intTotalTbls

catCurr.ActiveConnection = CurrentProject.Connection

```

TIP

If you look at the `RecordCount` property without first moving to the end of the recordset with a `MoveLast` method, you will get a `-1` back in some cases for the `RecordCount`. To see whether an ADO recordset has any records, use something similar to the following:

```
If (rstBOF And rst.EOF()) Then '-- no records found
```

One reason to avoid using the `MoveLast` method is that it can slow performance.

2. Use the `Catalog` object to delete the current link for a table:

```
'-- Attempt to open the current link  
On Error Resume Next
```

```
strCurrTable = rstSharedTables!TableName  
catCurr.Tables.Delete strCurrTable  
catCurr.Tables.Refresh
```

```
On Error GoTo Err_LinkTables
```

3. The new (linked) table is created in the application database:

```
Dim tblCurr As New ADOX.Table  
Set tblCurr.ParentCatalog = catCurr  
  
tblCurr.Name = rstSharedTables!TableName  
  
tblCurr.Properties("Jet OLEDB:Link Datasource") = strDataMDB  
tblCurr.Properties("Jet OLEDB:Create Link") = True  
tblCurr.Properties("Jet OLEDB:Remote Table Name") = _  
    rstSharedTables!TableName  
  
catCurr.Tables.Append tblCurr  
  
Set tblCurr = Nothing
```

CAUTION

You must create a new ADOX table, and then set the reference to `Nothing` when you're done with it. In DAO, you can create multiple tables with variables, as long as you append each table you create to the `TableDefs` collection. This isn't the case with ADO. If you don't clear and re-create the reference each time, you get an innocuous error message and spend hours trying to figure it out.

Finding the Jet Back End with the OpenFile API Call

In section 5 of the `ap_AppInit()` function, things are simple because the back-end database and the application are in the same folder. But most of the time, this won't be the case. Frequently, the application's back end will be out on a network drive, and the front end will be on a local drive. Section 5 of `ap_AppInit()` covers this situation:

```
'-- Section 5: Allow the user to locate the Backend MDB
If Not ap_LocateBackend(rstSharedTables, _
    strCurrError) Then
    flgLeaveApplication = True
End If
End If
```

Listing 25.9 shows the code for the `ap_LocateBackend()` function.

LISTING 25.9 VideoApp(ADO).mdb: Locating the Back End with a Call to an API Routine

```
Public Function ap_LocateBackend(rstSharedTables, strCurrError) As Boolean
    Dim ocxDialog As Object

    ap_LocateBackend = True

    DoCmd.Echo True
    Beep

    If MsgBox("A problem has occurred accessing the linked tables." & _
        vbCrLf & vbCrLf & "The error was: " & strCurrError & vbCrLf & _
        vbCrLf & "Would you like to locate the backend?", vbCritical + _
        vbYesNo, "Error with Backend") = vbYes Then
        Dim strFileName As String
        strFileName = ap_OpenFile(pstrBackEndName, "Locate backend database")
        If Len(strFileName) <> 0 Then
            DoEvents
            ap_LinkTables rstSharedTables, strFileName
            pstrBackEndPath = Left$(strFileName, InStrRev(strFileName, "\"))
        Else
            ap_LocateBackend = False
        End If
    Else
        ap_LocateBackend = False
    End If

Exit_ap_LocateBackend:
    Exit Function

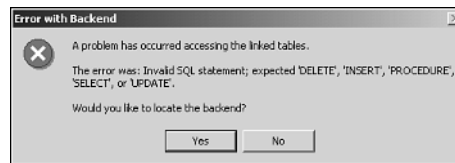
Error_ap_LocateBackend:
```

LISTING 25.9 Continued

```
ap_LocateBackend = False  
Resume Exit_ap_LocateBackend
```

End Function

First, users are told that the back end can't be found. They then are asked whether they want to locate it (see Figure 25.8). If they click Yes, the Locate Backend Database dialog appears (see Figure 25.9).

**FIGURE 25.8**

Try to give users as much polite information as possible when displaying message boxes.

**FIGURE 25.9**

The Locate Backend Database dialog is one of many you can use through API calls.

TIP

The message box in Figure 25.8 breaks up the message with blank lines, to make it easier to read. The `vbCrLf` constant actually places `Chr$(13)` and `Chr$(10)` (carriage return/line feed), so two `vbCrLf` constants together finish the first line and then add a blank line. You can also use an ampersand (&) to break the text into sections with a heading.

The following code line calls `ap_OpenFile`, an API wrapper function located in `modWindowsAPICalls`:

```
strFileName = ap_OpenFile(pstrBackEndName, "Locate backend database")
```

Listing 25.10 shows the code for the `ap_OpenFile()` function.

LISTING 25.10 VideoApp(ADO).mdb: Making the API Call

```
Public Function ap_OpenFile(Optional ByVal strFileNameIn As String = "", _
    Optional strDialogTitle As String = "Name of File to Open")
    Dim lngReturn As Long
    Dim intLocNull As Integer
    Dim strTemp As String
    Dim ofnFileInfo As OPENFILENAME
    Dim strInitialDir As String
    Dim strFileName As String

    '-- if a file path passed in with the name, parse it and split it off.
    If InStr(strFileNameIn, "\") <> 0 Then
        strInitialDir = Left(strFileNameIn, InStrRev(strFileNameIn, "\"))
        strFileName = Left(Mid$(strFileNameIn, _
            InStrRev(strFileNameIn, "\" ) + 1) & String(256, 0), 256)
    Else
        strInitialDir = CurrentProject.Path
        strFileName = Left(strFileNameIn & String(256, 0), 256)
    End If

    With ofnFileInfo
        .lStructSize = Len(ofnFileInfo)
        .lpstrFile = strFileName
        .lpstrFileTitle = String(256, 0)
        .lpstrInitialDir = strInitialDir
        .hwndOwner = Application.hWndAccessApp
        .lpstrFilter = "All Files (*.*)" & Chr(0) & "*.*)" & Chr(0)
        .nFilterIndex = 1
        .nMaxFile = Len(strFileName)
        .nMaxFileTitle = ofnFileInfo.nMaxFile
        .lpstrTitle = strDialogTitle
        .flags = cd1OFNFileMustExist Or cd1OFNHideReadOnly Or cd1OFNNoChangeDir
        .hInstance = 0
        .lpstrCustomFilter = String(255, 0)
        .nMaxCustFilter = 255
        .lpfnHook = 0
    End With
```


LISTING 25.10 Continued

```

lngReturn = ap_GetOpenFileName(ofnFileInfo)
If lngReturn = 0 Then
    strTemp = ""
Else
    '-- Trim off any null string
    strTemp = Trim(ofnFileInfo.lpstrFile)
    intLocNull = InStr(strTemp, Chr(0))
    If intLocNull Then
        strTemp = Left(strTemp, intLocNull - 1)
    End If
End If

ap_OpenFile = strTemp

```

End Function

An explanation of this code and other API calls can be found in Chapter 15, “Extending the Power of Access with API Calls.”

Testing and Repairing Corrupted Jet Back-End Databases

Section 6 of the `ap_AppInit()` function in Listing 25.1 deals solely with handling the repair of the back end if required:

```

Case apErrDBCorrupted1
    '-- Section 6: Backend Corrupted. Repair?
    Beep
    If MsgBox("The Backend Database is Corrupted." & vbCrLf & _
        vbCrLf & "Would you like to log users out and " & _
        " attempt to compact/repair it?", vbYesNo + vbCritical, _
        "Corrupted Backend!") = vbYes Then
        DoCmd.OpenForm "ap_CompactDatabase", acForm
    Else
        flgLeaveApplication = True
    End If
End Select

```

NOTE

A couple of changes with Jet 4.0 is that you don't use Compact and Repair to perform the separate functions. They're combined in the CompactDatabase method.

Also, the CompactDatabase method has been added to the Application object in Access 2002. The example code uses this method to repair the database. Because you are using the Application object, chances are you won't have to worry about changing it in future versions of Access, regardless of which data access method you use.

Section 6 starts by handling the situation where lngCurrError is equal to apErrDBCorrupted1, the constant used for corrupted database error. Next, a message box informs users of the situation and asks whether they want to try to compact/repair it.

After displaying the message box, the ap_CompactDatabase form opens. The OnLoad event of ap_CompactDatabase first checks whether anyone has already created the LogOut.flg file. If the file has been created, the routine exits the system. Listing 25.11 shows the OnLoad procedure for ap_CompactDatabase.

LISTING 25.11 VideoApp(ADO).mdb: Attempting to Compact/Repair the Database

```
Private Sub Form_Load()
    Dim strBackEndPath As String
    Dim strBackEndName As String
    Dim intCurrError As Integer

    '-- Grab the backend and path
    strBackEndPath = ap_GetDatabaseProp("LastBackEndPath")
    strBackEndName = ap_GetDatabaseProp("BackEndName")

    '-- If something else is up, exit
    If ap_LogOutCheck(strBackEndPath) Then
        Beep
        MsgBox "Somebody has already requested users to log out." & vbCrLf & _
            vbCrLf & "All users are requested to logout at this time.", _
            vbOKOnly + vbCritical, "Logging Out for Maintenance"
        GoTo Exit_Form_Load
    End If

    '-- Create the flag file to log users out
    ap_LogOutCreate

    On Error GoTo Error_Form_Load

    DoCmd.SelectObject A_FORM, "ap_CompactDatabase"

    intCurrError = -1
```

LISTING 25.11 Continued

```
'-- Keep trying until no error
Do While intCurrError <> 0
    DoCmd.Echo True, "Attempting To Compact/Repair BackEnd Database..."

    '-- This lets windows catch up
    DoEvents

    '-- This flag is set in the cmdCancel click event
    If intCancel Then
        GoTo Exit_Form_Load
    End If

    On Error Resume Next

    ' Make sure that a file doesn't exist with the name of
    ' the compacted database.
    '-- Try to compact into a temp file
    Application.CompactRepair strBackEndPath & strBackEndName, _
        strBackEndPath & "CompTemp.mdb"

    intCurrError = Err.Number

    '-- if no error, continue on
    If intCurrError = 0 Then
        On Error GoTo Error_Form_Load
        '-- Delete the original database,
        '-- then rename the temp to the original name
        Kill strBackEndPath & strBackEndName
        Name strBackEndPath & "CompTemp.mdb" As strBackEndPath & strBackEndName
    End If

Loop

'-- Delete the flag file
ap_LogOutRemove

MsgBox "Backend Compacted/Repaired Successfully!"

Exit_Form_Load:
    DoCmd.Close acForm, "ap_CompactDatabase"
    Exit Sub

Error_Form_Load:
```

LISTING 25.11 Continued

```

MsgBox "The backend has not been compacted/repared. The request for " & _
    "users to log out will be canceled!" & vbCrLf & vbCrLf & _
    "Please notify the system administrator.", vbCritical, "Compact Canceled"
ap_LogOutRemove

Resume Exit_Form_Load

End Sub

```

The `Form_Load` event procedure for `ap_CompactDatabase` ends up doing double duty in that, in addition to testing for and creating the flag file (if necessary), it allows attempts to repair the database. Figure 25.10 shows the `ap_CompactDatabase` form.

**FIGURE 25.10**

Feedback, as well as a way to cancel, is necessary when a task will take time.

It takes approximately five minutes for users to be logged out from the system after the logout flag file is created. It might take a few tries before the system can perform the compact/repair command on the back end. The following section of code attempts to repair the database:

```

intCurrError = -1

'-- Keep trying until no error
Do While intCurrError <> 0
    DoCmd.Echo True, "Attempting To Compact/Repair BackEnd Database..."

    '-- This lets windows catch up
    DoEvents

    '-- This flag is set in the cmdCancel click event
    If intCancel Then
        GoTo Exit_Form_Load
    End If

    On Error Resume Next

    ' Make sure that a file doesn't exist with the name of
    ' the compacted database.

```

```

'-- Try to compact into a temp file
Application.CompactRepair strBackEndPath & strBackEndName, _
    strBackEndPath & "CompTemp.mdb"

intCurrError = Err.Number

'-- if no error, continue on
If intCurrError = 0 Then
    On Error GoTo Error_Form_Load
    '-- Delete the original database,
    '-- then rename the temp to the original name
    Kill strBackEndPath & strBackEndName
    Name strBackEndPath & "CompTemp.mdb" As strBackEndPath & strBackEndName
End If

```

Loop

In the loop, error handling is set to `Resume Next` so that the error can be examined and the routine can keep trying to compact/repair the back end. After the database is repaired successfully, the system removes the logout flag file with a call to the `ap_LogOutRemove` routine, explained earlier in the section “Setting the Flag File to Log Users Out of the Back End.”

TIP

Set the `Err` value to 0 before performing the action to be tested. In a loop, if the action returns no errors, the `Err` value retains the value from the previous action.

If any other errors occur in the routine, a message box appears, explaining that the back-end database wasn’t repaired. The logout flag file is then erased so that others have a chance to correct the problem. The user is then sent out of the application.

The only other routine used for the `ap_CompactDatabase` form is the `OnClick` event of the `btnCancel` command button. If users decide not to wait any longer, they can click `Cancel` and get the same message given in the error routine of the `OnCurrent` event (see Figure 25.11).

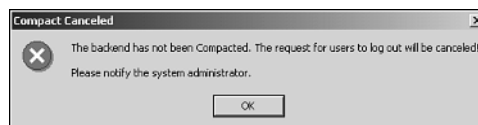


FIGURE 25.11

By using the event-driven model of Access, you can cancel just about any task, if necessary.

The next two sections of the `ap_AppInit()` function are pretty straightforward, as you can see:

```
'-- Section 7: Leave the application if requested
If flgLeaveApplication Then
    Application.Quit
    Exit Function
End If
On Error Resume Next
'-- Section 8: Let's try and open the first table again.
rstTestTable.Open strTableName, cnnLocal, adOpenStatic
lngCurrError = Err.Number
strCurrError = Err.Description
```

Loop

```
On Error GoTo Error_ap_App_Init
```

In section 7, the `flLeaveApplication` variable was set in prior sections to see whether leaving the application is needed. Section 8 rechecks the initial table in `tblSharedTables` to see whether the error loop can be exited safely.

The next section of real interest is section 9, where the user is kept up-to-date with the current version of the application.

Checking and Notifying Users of a New Version

When you're dealing with a large number of installations, it's not always feasible to try to personally update everyone's local machine with the most current version of the front end. Even letting users know that a new version is available with a network message doesn't always do the job. Users don't always tend to pay attention to network or system messages—although they're quick to call when the application blows up. What you can do to handle this situation is to build version checking into the startup routine. This is what section 9 of the `ap_AppInit()` function is all about:

```
'-- Section 9: Check the version of the front end,
'--           and point to a new one if necessary.
If Not CheckFEVersion() Then
    Application.Quit
End If
```

This code calls the `CheckFEVersion()` function, found in `modGlobalRoutines` and shown in Listing 25.12.

LISTING 25.12 VideoApp(ADO).mdb: Double-Checking the Version of the Front End

```

Public Function CheckFEVersion() As Boolean
    Dim cnnNet As New ADODB.Connection
    Dim rstBEDefault As New ADODB.Recordset

    '-- Open a connection to the backend database
    cnnNet.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & _
        pstrBackEndPath & pstrBackEndName

    '-- Open the database properties table
    rstBEDefault.Open "zstblDatabaseProperties", cnnNet

    '-- Compare frontend version
    If rstBEDefault!FrontEndVersion <> ap_GetDatabaseProp("FrontEndVersion") Then
        MsgBox rstBEDefault!FrontEndMessage, vbCritical
        CheckFEVersion = False
    Else
        CheckFEVersion = True
    End If

    '-- Clean up
    rstBEDefault.Close
    cnnNet.Close

    Set rstBEDefault = Nothing
    Set cnnNet = Nothing

End Function

```

This function uses the `zstblDatabaseProperties` table, stored on both the front and back ends. The table found on the back end has two fields: `FrontendVersion` and `FrontEndMessage`. The front-end table has just the `FrontEndVersion` field. The back-end table is opened with the `rstBEDefault` recordset and compared to the value returned from the `ap_GetDatabaseProp()` function. If the two fields don't match, a message box appears, using the value found in the `FrontEndMessage` field.

After the message box appears with the new version message, the code returns a false value; then the `ap_AppInit` routine quits the application. This way, users are not only informed that they need to set up a new version, but they are also kept from going any further into the application.

The final section is section 10:

```

'-- Section 10: Save the BackEnd path in the BackEndPath property
'--             for future use.
ap_SetDatabaseProp "LastBackEndPath", pstrBackEndPath

```

```
'-- Check for locally logged errors
ap_ErrorCheckLocal

'-- Check for Replicated Tables
ap_CheckReplicatedTables

'-- Turn on the Monitor Form
DoCmd.OpenForm "UserLogOutMonitor", , , , , acHidden

DoCmd.Close acForm, "SplashScreen"
DoCmd.Echo True
rstSharedTables.Close
```

This code performs the following tasks:

1. It stores the current back-end path in the custom database property LastBackEndPath.
2. The next two called routines perform tasks that are discussed in other chapters. The `ap_ErrorCheckLocal` routine, discussed in Chapter 7, “Handling Your Errors in Access with VBA,” helps log errors into a table. The `ap_CheckReplicatedTables` routine, discussed in Chapter 26, “Creating Maintenance Routines,” replicates tables from the back end to the front for performance purposes.
3. The code opens the `UserLogOutMonitor` form. This form, used to monitor whether users need to log out, is discussed in “Logging Users Out in the Middle of the Application” earlier in this chapter.
4. It closes the splash screen.
5. The code closes the `rstSharedTables` recordset.

Summary

Startup system checking is very important to creating robust applications. The more work you do up front to handle problems on the way into the system, the fewer errors will occur when users are in the middle of performing their jobs. For more information about some of the commands seen in this chapter, review the following chapters:

- Chapter 2, “Coding in Access 2002 with VBA,” covers the commands included in Visual Basic for Applications, including information about the Object Browser, With command, and many more.
- Chapter 15, “Extending the Power of Access with API Calls,” shows what other kinds of API calls are available and how to take advantage of them.

Creating Maintenance Routines

CHAPTER

26

IN THIS CHAPTER

- **Creating an Export Dialog to Export Tables 862**
- **Compacting and Repairing the Back End on Demand 868**
- **Creating a Generic Code Table Editor 871**
- **Replicating Tables from Back End to Front End for Better Performance 874**

This chapter presents some useful routines, combining ideas that I hope you've gleaned from the rest of book. One goal of this chapter is that, by examining and using these routines, you can continue to learn new techniques. These routines aren't in any particular order and have diverse uses.

You can find each example in the VideoApp(ADO).mdb database in the \Examples folder on the book's Web page at www.sampublishing.com.

Creating an Export Dialog to Export Tables

Often, it's a good idea to give users—or at least administrators—an option for exporting tables. That way, they can do so without the hassle of having them get to the database container.

To take this a step further, how about giving users the choice of exporting to various file formats? Sometimes giving users the flexibility they need with the control that's necessary can be a challenge.

The `ap_SystemUtilities` form can export various tables in the applications. Figure 26.1 shows this form with multiple tables selected to export.

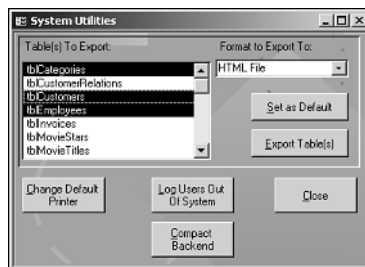


FIGURE 26.1

This utility form allows you to export multiple tables.

This example of exporting tables does the following:

- It uses a multi-select list box and shows how to code for it.
- It allows users to get or set a default export type by using default values stored in a local table.
- It lets users choose from a list of different file types, including exports to various database formats, spreadsheets, text files, and HTML.

Examining the Export Utility's Features

The export utility uses a table and a form. The table, `tblSharedTables`, was originally used for linking tables in Chapter 25, “Startup Checking System Routines Using ADO.” It’s used again here to track a filename for each exportable table in the application. Figure 26.2 shows this table.



TableName	ExportFileName
tblCategories	Categories
tblCustomerRelations	CustRel
tblCustomers	Customer
tblEmployees	Employee
tblInvoices	Invoices
tblMovieStars	Mstars
tblMovieTitles	Movies
tblMovieTitlesCategories	MTCats
tblNewReleases	NewRel
tblRelationCodes	Relation
tblRentalHistory	RentHist
tblStars	Stars
tblTapes	Tapes

FIGURE 26.2

This table serves two purposes: helping relink tables and exporting tables.

The `ap_SystemUtilities` form in this example uses the `lboTablesToExport` multiple-selection list box, which allows users to select the tables they want to export. (Multiple-selection list boxes are covered in more detail in Chapter 10, “Expanding the Power of Your Forms with Controls.”)

The `lboTablesToExport` list box uses the `tblSharedTables` table as its row source. Figure 26.3 shows the property sheet of `lboTablesToExport` with the rest of the `ap_SystemUtilities` form.

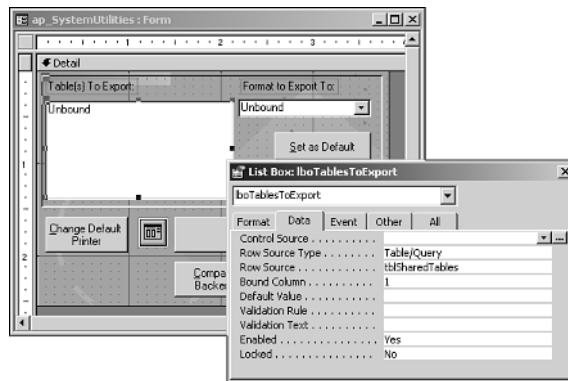


FIGURE 26.3

Users can pick tables to export from this form's multiple-selection list box.

Users choose the export file type through the `cboExportFileType` combo box. This combo box lists the file types supported for exporting as a row source. The export utility supports the following file types:

<i>File Type</i>	<i>Category</i>
dBASE 5.0	Database
Excel 97–2002	Spreadsheet
HTML file	HTML
XML document	XML
Text	Delimited text

You can add other file types as needed. To see which types can be exported, look at the `DoCmd` method's `TransferSpreadsheet`, `TransferText`, and `OutputTo`, as well as the `Application` method called `ExportXML`. By looking at the corresponding macro actions, selecting these methods, and then clicking the Database Type action argument, you get a list of possible file types. This can be seen in Figure 26.4, where `TransferDatabase` has been selected. You can then take the text given for the file type and use it with your VBA code, as will be shown shortly in Listing 26.1.

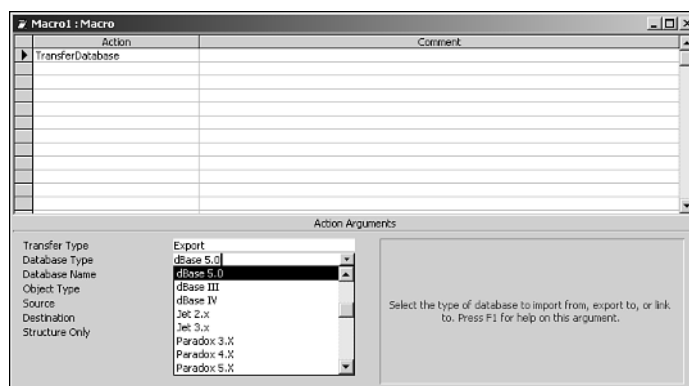


FIGURE 26.4

Check the Transfer macro actions to see which file types are supported in exporting tables.

Before getting into the code that performs the actual work, look at the `cmdSetDefault_Click` routine, which sets a default export type. This routine is attached to the `cmdSetDefault` command button's `OnClick` event. It calls `ap_SetDatabaseProp`, which Chapter 25 discusses in more detail.

```
Private Sub cmdSetDefault_Click()
    ap_SetDatabaseProp "ExportTypeDefault", Me!cboExportFileType
End Sub
```

The `ExportTypeDefault` field is in the `ztblDatabaseProperties` table, which stores defaults. Figure 26.5 shows this field and other default fields used by the `VideoApp(ADO).mdb` database.

BackEndName	LastBackEndPath	FrontEndVersion	ExportTypeDefault
VideoDat.mdb	C:\Books\PwrPrg2002\AppCD\Examples\	1.0	HTML File

FIGURE 26.5

Storing values in a table is a good way to keep track of values that pertain to the overall database.

To set the default export type, follow these steps:

1. Select the export type to set as a default in the `cboExportFileType` combo box.
2. Click the `cmdSetDefault` command button.

The next time someone uses this form, whatever export type that user selects as the default will be initially displayed.

Before examining the code for the actual exporting of files in Listing 26.1, look at the export utility in action in Figure 26.6.

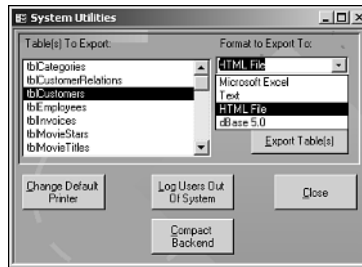


FIGURE 26.6

The export utility allows users to export multiple tables to a number of different file formats.

Examining the Code Behind the Export Utility

The code that does the actual exporting is attached to the `cmdExportTables` command button. Listing 26.1 shows the code attached to the `OnClick` event.

LISTING 26.1 VideoApp(ADO).mdb: Performing the Actual Export

```

Private Sub cmdExportTables_Click()
    Dim strAppPath As String
    Dim strExportFileName As String
    Dim varCurrent As Variant

    On Error GoTo Error_cmdExportTables_Click

    DoCmd.Hourglass True

    '-- Establish application path
    strAppPath = CurrentProject.Path

    Dim lboTablesToExp As ListBox
    Set lboTablesToExp = Me!lboTablesToExport

    For Each varCurrent In lboTablesToExp.ItemsSelected
        '-- Perform export if table is Selected
        DoCmd.Echo True, "Exporting " & lboTablesToExp.ItemData(varCurrent) & "..."
        '-- Get the export file name
        strExportFileName = DLookup("ExportFileName", "tblSharedTables", _
            "[TableName]= '" & lboTablesToExp.ItemData(varCurrent) & "'")
        '-- Export to text file
        If cboExportFileType = "Text" Then
            DoCmd.TransferText acExportDelim, "", _
                lboTablesToExp.ItemData(varCurrent), strAppPath & _
                "\" & strExportFileName & ".txt"
        '-- Export to spreadsheet
        ElseIf InStr(cboExportFileType, "Excel") <> 0 Then
            DoCmd.TransferSpreadsheet acExport, acSpreadsheetTypeExcel9, _
                lboTablesToExp.ItemData(varCurrent), strAppPath & "\" & _
                strExportFileName
        '-- Export to HTML
        ElseIf cboExportFileType = "HTML File" Then
            DoCmd.OutputTo acOutputTable, lboTablesToExp.ItemData(varCurrent), _
                acFormatHTML, strAppPath & "\" & strExportFileName & ".htm"
        '-- Export to XML
        ElseIf cboExportFileType = "XML Document" Then
            Application.ExportXML acExportTable, _
                lboTablesToExp.ItemData(varCurrent), _
                strAppPath & "\" & strExportFileName & ".xml"
        '-- Otherwise export to a database
        Else
            DoCmd.TransferDatabase acExport, cboExportFileType, strAppPath, _

```

LISTING 26.1 Continued

```

        acTable, lboTablesToExp.ItemData(varCurrent), strExportFileName
    End If
Next

'-- Clean up and notify
DoCmd.Hourglass False
DoCmd.Echo True
Beep
MsgBox "Table(s) Exported Successfully into " & strAppPath & "!"

Exit Sub

Error_cmdExportTables_Click:
    DoCmd.Hourglass False
    DoCmd.Echo True
    MsgBox Err.Description
    Exit Sub

End Sub

```

The routine in Listing 26.1 uses a number of interesting commands:

- One command cycles through each table selected in the lboTablesToExport list box. In the following code snippet, the For Each statement cycles through the ItemsSelected collection off the list box. The ItemData property displays the value of the current selected item.

```

Dim lboTablesToExp As ListBox
Set lboTablesToExp = Me!lboTablesToExport

For Each varCurrent In lboTablesToExp.ItemsSelected
    '-- Perform export if table is Selected
    DoCmd.Echo True, "Exporting " & lboTablesToExp.ItemData(varCurrent) & _
        "..."
```

- The DLookup() function is used to retrieve the DOS file name stored in the SharedTables table:


```

'-- Get the DOS export file name
strExportFileName = DLookup("ExportFileName", "tblSharedTables", _
    "[TableName]= '" & lboTablesToExp.ItemData(varCurrent) & "'")
```
- The type of exporting is examined, and the appropriate command is used. You can examine the various Transfer-type (and OutputTo) methods in Listing 26.1.

Compacting and Repairing the Back End on Demand

Although the front-end application can perform the compact/repair itself, it's necessary to log other users out of the back end and test to ensure that no one else is in the application. The form used for compacting the back end is `ap_CompactDatabase` (see Figure 26.7). Most work performed by this form is in the `OnLoad` event.

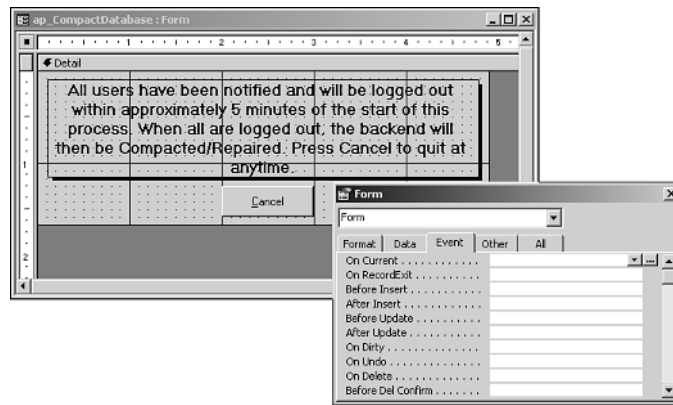


FIGURE 26.7

This form appears while the application tries to compact the back end.

Three routines called in Listing 26.2—`ap_LogOutCheck`, `ap_LogOutCreate`, and `ap_LogOutRemove`—check, create, and remove the flag file in the back-end folder. The flag file logs other users out of the back end. Chapter 25, “Startup Checking System Routines Using ADO,” fully covers these routines.

LISTING 26.2 VideoApp(ADO).mdb: Compacting a Database

```
Private Sub Form_Load()
    Dim strBackEndPath As String
    Dim strBackEndName As String
    Dim intCurrError As Integer

    '-- Grab the backend and path
    strBackEndPath = ap_GetDatabaseProp("LastBackEndPath")
    strBackEndName = ap_GetDatabaseProp("BackEndName")

    '-- If something else is up exit
    If ap_LogOutCheck(strBackEndPath) Then
```

LISTING 26.2 Continued**26**

```

Beep
    MsgBox "Somebody has already requested users to log out." & vbCrLf & _
        vbCrLf & "All users are requested to logout at this time.", _
        vbOKOnly + vbCritical, "Logging Out for Maintenance"
    GoTo Exit_Form_Load
End If

'-- Create the flag file to log users out
ap_LogOutCreate

On Error GoTo Error_Form_Load

DoCmd.SelectObject A_FORM, "ap_CompactDatabase"

intCurrError = -1

'-- Keep trying until no error
Do While intCurrError <> 0
    DoCmd.Echo True, "Attempting To Compact/Repair BackEnd Database..."
    '-- This lets windows catch up
    DoEvents
    '-- This flag is set in the cmdCancel click event
    If intCancel Then
        GoTo Exit_Form_Load
    End If

    On Error Resume Next

    ' Make sure that a file doesn't exist with the name of
    ' the compacted database.
    '-- Try to compact into a temp file
    Application.CompactRepair strBackEndPath & strBackEndName, _
        strBackEndPath & "CompTemp.mdb"

    intCurrError = Err.Number

    '-- if no error, continue on
    If intCurrError = 0 Then
        On Error GoTo Error_Form_Load
        '-- Delete the original database,
        '-- then rename the temp to the original name
        Kill strBackEndPath & strBackEndName
        Name strBackEndPath & "CompTemp.mdb" As strBackEndPath & strBackEndName
    End If

```

LISTING 26.2 Continued

```

    Loop

    '-- Delete the flag file
    ap_LogOutRemove

    MsgBox "Backend Compacted/Repaired Successfully!"

Exit_Form_Load:
    DoCmd.Close acForm, "ap_CompactDatabase"
    Exit Sub

Error_Form_Load:
    MsgBox "The backend has not been compacted/repared. The request for " & _
        "users to log out will be canceled!" & vbCrLf & vbCrLf & _
        "Please notify the system administrator.", vbCritical, "Compact Canceled"
    ap_LogOutRemove
    Resume Exit_Form_Load

End Sub

```

The main idea behind the routine in Listing 26.2 is to continue to try compacting and repairing the back end until it succeeds or Cancel is clicked. The following code performs this:

```

'-- This flag is set in the cmdCancel click event
If intCancel Then
    GoTo Exit_Form_Load
End If

```

If the compact is successful, the next code section deletes the original database and renames the temporary database to the original filename:

```

On Error Resume Next

' Make sure that a file doesn't exist with the name of
' the compacted database.
'-- Try to compact into a temp file
Application.CompactRepair strBackEndPath & strBackEndName, _
    strBackEndPath & "CompTemp.mdb"

intCurrError = Err.Number

'-- if no error, continue on
If intCurrError = 0 Then
    On Error GoTo Error_Form_Load
    '-- Delete the original database,

```

```
'-- then rename the temp to the original name
Kill strBackEndPath & strBackEndName
Name strBackEndPath & "CompTemp.mdb" As strBackEndPath & strBackEndName
End If
```

NOTE

Kudos to Microsoft for adding the CompactRepair method to Access 2002's Application object. In prior versions, you had to go off either Jet's dbEngine object or pull in ADO's JetEngine object. Now you can constantly use the Application object even if Microsoft changes data access methods again.

The code in Listing 26.3 is attached to the Cancel button shown on the ap_CompactDatabase form in Figure 26.7. When the database is compacted/repaired, the logout flag file is removed and life goes on.

LISTING 26.3 VideoApp(ADO).mdb: Cleaning Up If Canceled

```
Private Sub Cancel_Click()
    Beep
    MsgBox "The backend has not been compacted/repaired. The request for " & _
        "users to log out will be canceled!" & vbCrLf & vbCrLf & "Please " & _
        "notify the system administrator.", vbCritical, "Compact/Repair Canceled"
    ap_LogOutRemove
    intCancel = True
End Sub
```

Creating a Generic Code Table Editor

Most applications use a number of *code tables*, which list the values used in a set field. This list is usually static but can be dynamic. tblRatings is an example of a code table. Coming up with a method for handling code tables is helpful so that when a new one is added, minimal work needs to be done.

A generic code table editor is probably one of the simplest techniques shown in this book. It consists of one main form and whatever subforms are needed for the various code tables. Figure 26.8 shows the main form, with the tblCategories codes displayed in the subform.

**FIGURE 26.8**

The same main form is used with multiple subforms.

The main form used for this task, `ap_StandardCodeMain`, contains a subform control, `subStandardCode`, with no `SourceObject` property set. The `ap_StandardCodeMain` form also has a combo box control that lists the code tables to edit. The combo box, `cboTablesToEdit`, is populated in the form's `OnLoad` event. Listing 26.4 shows the routine attached to the `OnLoad` event for `ap_StandardCodeMain`.

LISTING 26.4 VideoApp(ADO).mdb: Loading a Combo Box with Code Table Names

```
Private Sub Form_Load()
    Dim strFileToEdit As String
    Dim accObj As AccessObject

    On Error GoTo Error_Form_Load

    '-- Go through each of the form objects in the Current Project
    For Each accObj In CurrentProject.AllForms
        '-- If the form is one of the code table subforms,
        '-- add it to the string that makes up the combo list source
        If InStr(accObj.Name, "CodeSubform") <> 0 Then
            strFileToEdit = strFileToEdit & Left$(accObj.Name, _
                InStr(accObj.Name, "CodeSubForm") - 1) & ";"
        End If
    Next

    '-- Assign the row source of the combo box.
    Me!cboTablesToEdit.RowSource = strFileToEdit

    Exit Sub

Error_Form_Load:
    MsgBox Err.Description
    Exit Sub

End Sub
```

TIP

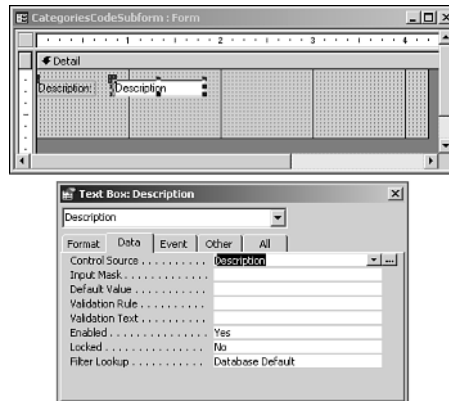
This code is a great example of using the AllForms collection off the CurrentProject. Notice the way it cycles through the Forms container with the For Each command. For more information on both objects, see Chapter 4, "Working with Access Collections and Objects."

26**CREATING
MAINTENANCE
ROUTINES**

This routine first cycles through each form object in the AllForms collection by using the For Each statement:

```
'-- Go through each of the form objects in the Current Project
For Each accObj In CurrentProject.AllForms
```

For the next piece of code to work, you need to set up a subform for each code table you want to edit. You then need to name each subform to match the name of the code table added to the literal CodeSubform. For example, a subform for the tblCategories table would be named CategoriesCodeSubform (see Figure 26.9).

**FIGURE 26.9**

This form is used with the ap_StandardCodeMain form.

Next, the following code checks for the word *CodeSubform* in the name of the current form. If it's found, the table portion of the name is added to the string strFileToEdit.

```
If InStr(accObj.Name, "CodeSubform") <> 0 Then
    strFileToEdit = strFileToEdit & Left$(accObj.Name, _
        InStr(accObj.Name, "CodeSubForm") - 1) & ";"
End If
```

Finally, the routine assigns the temporary string, `strFileToEdit`, to the row source of the combo box:

```
Me!cboTablesToEdit.RowSource = strFileToEdit
```

NOTE

This method for creating the row source for a combo box works only when the string is less than 2K. If you get more tables than will fit, you should use an alternative method, such as creating a table and then basing the row source off that table.

The next section uses a modified code table editor to edit the tables on the back end, and then update a timestamp.

Replicating Tables from Back End to Front End for Better Performance

Some tables are updated only once or twice a week, if that often. These semi-static tables could be brought into the local database and improve performance because local tables generally are faster to access than linked tables. Such semi-static tables aren't quite static enough to keep them only in the front end. The solution to this is to replicate them from the back end to the front when the back-end version is updated.

NOTE

This type of "home made" replication works great for Access-based back ends, but performance will probably be better if you leave all the tables on a SQL Server back end. When query data is located in more than one location, Access brings down all the data down locally to process.

Also, Access and Windows support another kind of replication, using the Briefcase utility, from a copy of a complete database to another. This varies from what is discussed here and is fully covered in Chapter 22, "Welcome to the World of Database Replication."

To replicate tables from the back end to the front end, follow these general steps (specifics will be explained shortly):

1. Edit the back-end version of a replicated table.
2. Store the date and time of the edit in another table that tracks replicated tables.
3. Replicate the local version immediately.
4. In the system startup routines, add a routine that checks the date of the last replication, comparing front- and back-end date and times.
5. If the back-end version has been updated since the last replication, download it again. Then update the front-end version's date/timestamp.

The first place to start is by creating the editor that will edit the back-end version of replicated tables and then update the back-end date/timestamp.

Creating a Replicated Table Editor

This editor works similarly to the generic code editor discussed earlier, except that it also creates a temporary link to the back end. Figure 26.10 shows this editor, `ap_UpdateReplicatedBackend`, with the `tblUsers` table being edited.

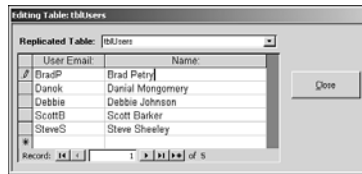


FIGURE 26.10

The values being edited are in the back end and will be replicated the next time the system is started at other stations.

Unlike with the earlier code editor, the combo box used to choose which table to edit doesn't have to be populated with code. This makes the `OnLoad` event for the form more straightforward, as shown in Listing 26.5.

LISTING 26.5 VideoApp(ADO).mdb: Setting Up the Environment

```
Private Sub Form_Load()
    On Error GoTo Error_Form_Load

    '-- Initialize the edit flag
    intRepEdited = False

    '-- Grab the backend and path
    mstrBackEndPath = ap_GetDatabaseProp("LastBackEndPath")
    mstrBackEndName = ap_GetDatabaseProp("BackEndName")
```


LISTING 26.5 Continued

```

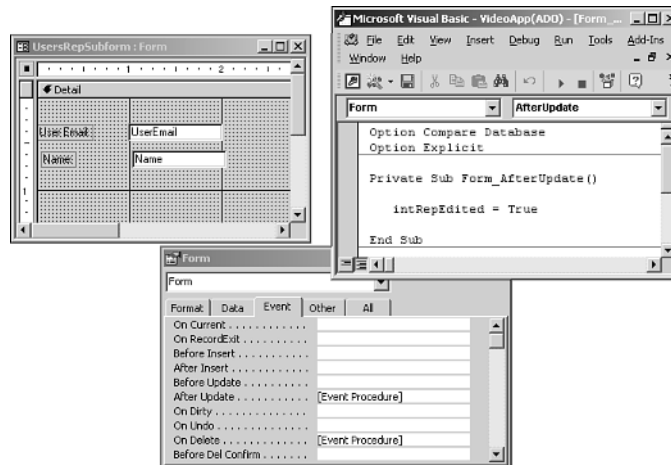
Exit Sub

Error_Form_Load:
    MsgBox Err.Description
Exit Sub

End Sub

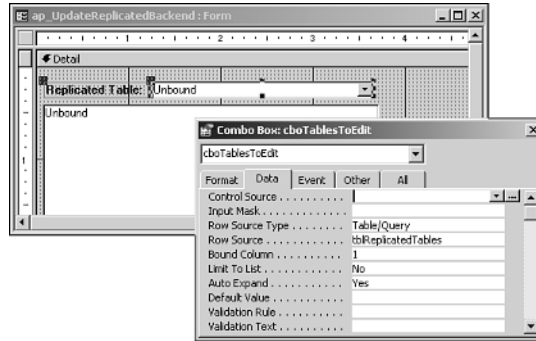
```

The `intRepEdited` variable is declared in the `ap_GlobalRoutines` module. It's initialized here in the `OnLoad` event as `False` and will be flagged as `True` in each subform's `AfterUpdate` and `OnDelete` flags. Figure 26.11 shows the `UserRepSubform` subform, its property sheet, and the routine for the `AfterUpdate` event.

**FIGURE 26.11**

intRepEdited must be set to True in each subform's AfterUpdate and OnDelete events.

The row source for the `cboTablesToEdit` combo box is set to the table `tblReplicatedTables` and stored in the front end. The form's real power is when a table is selected from the `cboTablesToEdit` combo box. Figure 26.13 shows this combo box, along with its property sheet, in Design view.

**FIGURE 26.12**

This combo box is populated more easily than the earlier code editor's combo box.

Listing 26.6 shows the code that executes when a table is chosen from the list of replicated tables, in the BeforeUpdate event procedure.

LISTING 26.6 VideoApp(ADO).mdb: Performing the Replication

```
Private Sub cboTablesToEdit_BeforeUpdate(Cancel As Integer)
    Dim catLocal As New adox.Catalog

    catLocal.ActiveConnection = CurrentProject.Connection

    On Error Resume Next

    DoCmd.Echo False

    If Len(mstrOldValue) > 0 Then
        '-- Delete the replicated link from the frontend
        catLocal.Tables.Delete "RPL" & mstrOldValue
    End If

    On Error GoTo Error_cboTablesToEdit_BeforeUpdate

    '-- Check and see if an edit has occurred
    CheckUpdateTime

    '-- Link the new table to edit and add RPL on the front of the name
    DoCmd.TransferDatabase acLink, "Microsoft Access", mstrBackEndPath & _
        mstrBackEndName, , Me!cboTablesToEdit, "RPL" & Me!cboTablesToEdit
    catLocal.Tables.Refresh

    '-- Assign the subform based on the table chosen
    '-- then update the form's caption.
```

LISTING 26.6 Continued

```

Me!subReplicatedTable.SourceObject = Mid$(Me!cboTablesToEdit, 4) & _
    "RepSubform"
Me.Caption = "Editing Table: " & Me!cboTablesToEdit

mstrOldValue = cboTablesToEdit

DoCmd.Echo True

Exit Sub

Error_cboTablesToEdit_BeforeUpdate:
DoCmd.Echo True
MsgBox Err.Description
Exit Sub

End Sub

```

Here's what occurs in this code:

1. If a table was selected before the current one, that link is deleted from the current project's table collection. The `On Error Resume Next` line has the code continue if there wasn't an error, and the `mstrOldValue` variable stores the name of the prior table:


```

On Error Resume Next

DoCmd.Echo False

If Len(mstrOldValue) > 0 Then
    '-- Delete the replicated link from the frontend
    catLocal.Tables.Delete "RPL" & mstrOldValue
End If

```
2. The `CheckUpdateTime` routine checks whether the prior table was updated. It also timestamps the back end, copies the replicated table to the local database, and updates the local replicated timestamp. This routine is performed after the `cboTablesToEdit_AfterUpdate` routine is completed.
3. The new table is linked with the literal `RPL` in front of the table name, and the `Tables` collection is refreshed through the `Refresh` method off an `ADOX` catalog. These lines of code accomplish this:


```

'-- Link the new table to edit and add RPL on the front of the name
DoCmd.TransferDatabase acLink, "Microsoft Access", mstrBackEndPath & _
    mstrBackEndName, , Me!cboTablesToEdit, "RPL" & Me!cboTablesToEdit
catLocal.Tables.Refresh

```

TIP

Refreshing the various collections is always a good idea when you append or delete objects to or from them, especially the Tables collection. New or deleted objects might not be seen immediately unless a Refresh method is performed. For more information on the Tables collection, see Chapter 4.

26**CREATING
MAINTENANCE
ROUTINES**

4. Finally, the subform control SourceObject and the caption for the main form are assigned:

```
'-- Assign the subform based on the table chosen
'-- then update the form's caption.
Me!subReplicatedTable.SourceObject = Mid$(Me!cboTablesToEdit, 4) & _
    "RepSubform"
Me.Caption = "Editing Table: " & Me!cboTablesToEdit
```

Again, the cboTablesToEdit_BeforeUpdate event procedure occurs whenever a new table is selected from the cboTablesToEdit combo box. Something similar, although not as dramatic, occurs when the form is closed. The UnLoad event of the ap_UpdateReplicatedBackend form occurs, and the code shown in Listing 26.7 is performed as documented.

LISTING 26.7 VideoApp(ADO).mdb: Cleaning Up on the Way Out

```
Private Sub Form_Unload(Cancel As Integer)
    Dim catLocal As New adox.Catalog
    catLocal.ActiveConnection = CurrentProject.Connection
    On Error Resume Next
    '-- Delete the replicated link from the frontend
    catLocal.Tables.Delete "RPL" & Me!cboTablesToEdit
    catLocal.Tables.Refresh
    '-- Check to see if anything was edited.
    CheckUpdateTime
    DoCmd.Echo True
End Sub
```

Figure 26.14 shows the ap_UpdateReplicatedBackend form after a table is edited and is being replicated.

**FIGURE 26.13**

This routine automatically replicates the changes to the front end immediately.

Finally, it's time to look at the CheckUpdateTime routine (see Listing 26.8).

LISTING 26.8 VideoApp(ADO).mdb: Actual Replication Takes Place

```
Sub CheckUpdateTime()
    Dim catLocal As New adox.Catalog
    Dim cmdUpdateRep As New ADODB.Command

    Set catLocal.ActiveConnection = CurrentProject.Connection

    '-- If current table is edited update backend timestamp, update the
    '-- frontend data and timestamp for local, then reset edit flag.

    mstrBackEndPath = ap_GetDatabaseProp("LastBackEndPath")
    mstrBackEndName = ap_GetDatabaseProp("BackEndName")

    On Error GoTo Error_CheckUpdateTime

    If intRepEdited Then
        DoCmd.Echo False, "Updating Backend..."
        DoCmd.Hourglass True
        DoCmd.SetWarnings False
        DoCmd.Echo True, "Replicating " & Me!cboTablesToEdit.OldValue & _
            ", Please wait..."
        '-- Timestamp the backend
        DoCmd.RunSQL "UPDATE tblReplicatedTables IN '" & mstrBackEndPath & _
            mstrBackEndName & "' SET tblReplicatedTables.LastUpdate = Now() _
            Where (((tblReplicatedTables.TableName) = '" & _
            & Me!cboTablesToEdit.OldValue & "'));"
        '-- Replicate the modified table
        DoCmd.Echo False, "Replicating " & Me!cboTablesToEdit.OldValue & _
            ", Please wait..."
    End If
End Sub
```

LISTING 26.8 Continued

```

'-- Delete the current local table,
'-- and import the backend table
On Error Resume Next

catLocal.Tables.Delete Me!cboTablesToEdit.OldValue
catLocal.Tables.Refresh

On Error GoTo Error_CheckUpdateTime

DoCmd.TransferDatabase acImport, "Microsoft Access", mstrBackEndPath & _
    mstrBackEndName, acTable, Me!cboTablesToEdit.OldValue, _
    Me!cboTablesToEdit.OldValue

catLocal.Tables.Refresh

Set cmdUpdateRep = catLocal.Procedures("qryUpdateLastReplication").Command
cmdUpdateRep.Parameters("CurrReplicatedTable") = _
    Me!cboTablesToEdit.OldValue
cmdUpdateRep.Execute

intRepEdited = False

DoCmd.Hourglass False
DoCmd.SetWarnings True

End If

Exit Sub

Error_CheckUpdateTime:
DoCmd.Hourglass False
DoCmd.SetWarnings True
DoCmd.Echo True
MsgBox Err.Description
Exit Sub

End Sub

```

These tasks take place in Listing 26.8 if the current table in the subform has been edited:

1. The entry in the table's back end is updated. This table consists of two fields: TableName and LastUpdate (see Figure 26.14). The following code lines update this table:

```

'-- Timestamp the backend
DoCmd.RunSQL "UPDATE tblReplicatedTables IN '" & mstrBackEndPath & _

```

```
mstrBackEndName & "' SET tblReplicatedTables.LastUpdate = Now() _
Where (((tblReplicatedTables.TableName) = '" _
& Me!cboTablesToEdit.OldValue & "')");"
```

TableName	LastReplication
tblRatings	4/14/2001 4:05:41 PM
tblUsers	4/14/2001 5:03:46 PM

FIGURE 26.14

At this time, `tblUsers` and `tblRatings` are the only replicated tables kept in the back-end version of `tblReplicatedTables`.

2. The current replicated table is deleted, and the updated table is imported from the back end:

```
'-- Delete the current local table,
'-- and import the backend table
On Error Resume Next
```

```
catLocal.Tables.Delete Me!cboTablesToEdit.OldValue
catLocal.Tables.Refresh
```

```
On Error GoTo Error_CheckUpdateTime
```

```
DoCmd.TransferDatabase acImport, "Microsoft Access", mstrBackEndPath & _
    mstrBackEndName, acTable, Me!cboTablesToEdit.OldValue, _
    Me!cboTablesToEdit.OldValue
```

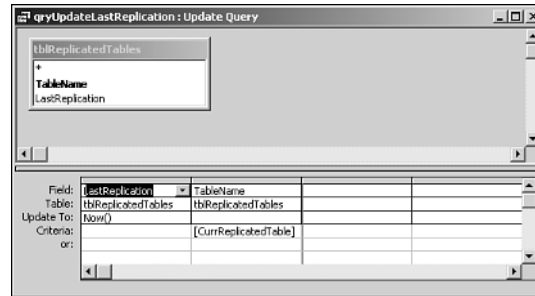
```
catLocal.Tables.Refresh
```

3. Because the table has been replicated, the local version of `tblReplicatedTables` is updated with a timestamp, using the `qryUpdateLastReplication` query (see Figure 26.15). Here's the code that uses this query:

```
Set cmdUpdateRep = catLocal.Procedures("qryUpdateLastReplication").Command
cmdUpdateRep.Parameters("CurrReplicatedTable") = _
    Me!cboTablesToEdit.OldValue
cmdUpdateRep.Execute
```

NOTE

The code in step 3 updates the front end of the current application making the changes. The front ends of the other stations will have a routine run at startup system checking time.

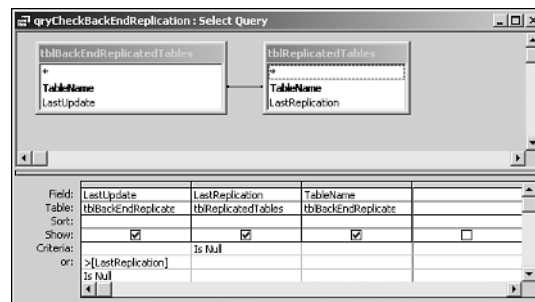
**FIGURE 26.15**

This query updates the current entry for LastReplication in the front-end version of the tblReplicatedTables table with the Now() system variable.

Looking at Startup Routines for Replicating Semi-Static Data

The pieces that make up the replication check routine are a query and a routine. The query, qryCheckBackEndReplication, compares the front-end and back-end table, tblReplicatedTables, and looks to see whether two things have occurred (see Figure 26.16):

- The front-end LastReplication field is Null, meaning that replication has never been performed for this particular table.
- The LastUpdated field in the back-end tblReplicatedTables (which is linked as BackEndReplicatedTables for the query in Figure 26.16) is more recent than the LastReplication field of the front-end tblReplicatedTables.

**FIGURE 26.16**

This query is useful for comparing the back and front ends for semi-static table replication.

The `ap_CheckReplicatedTables` routine performs the check. This routine, found in the `modGlobalRoutines` module, performs some of the same tasks as the `CheckUpdateTime` routine (refer to Listing 26.8) in that it replicates updated tables and timestamps the `LastReplication` field in the local `tblReplicatedTables`. The code, shown in its entirety in Listing 26.9, is examined more closely later in this section.

LISTING 26.9 VideoApp(ADO).mdb: Checking for Updated Tables

```

Sub ap_CheckReplicatedTables()
    Dim catLocal As New adox.Catalog
    Dim rstCheckRep As New ADODB.Recordset
    Dim cmdUpdateRep As New ADODB.Command
    Dim strTableName As String

    On Error GoTo Error_ap_CheckReplicatedTables

    Set catLocal.ActiveConnection = CurrentProject.Connection

    DoCmd.Echo True, "Checking for Replicated Tables..."

    '-- Attach the backend replicated table
    '-- and open the query that shows updated replicated tables
    On Error Resume Next

    catLocal.Tables.Delete "tblBackEndReplicatedTables"
    catLocal.Tables.Refresh

    On Error GoTo Error_ap_CheckReplicatedTables

    ap_CreateLinkedTableWithADO catLocal, "tblBackEndReplicatedTables", _
        "tblReplicatedTables", pstrBackEndPath & pstrBackEndName
    catLocal.Tables.Refresh

    Set cmdUpdateRep = catLocal. _
        Procedures("qryUpdateLastReplication").Command

    rstCheckRep.Open "qryCheckBackEndReplication", _
        CurrentProject.Connection, adOpenStatic

    '-- If a table has been updated, loop through
    Do Until rstCheckRep.EOF
        DoCmd.Echo True, "Replicating " & rstCheckRep!TableName & _
            ", Please wait..."
    
```

LISTING 26.9 Continued

```

'-- Delete the current local table, and import the backend table
On Error Resume Next
strTableName = rstCheckRep!TableName

catLocal.Tables.Delete strTableName
catLocal.Tables.Refresh

On Error GoTo Error_ap_CheckReplicatedTables

DoCmd.TransferDatabase acImport, "Microsoft Access", pstrBackEndPath & _
    pstrBackEndName, acTable, rstCheckRep!TableName, rstCheckRep!TableName

catLocal.Tables.Refresh

cmdUpdateRep.Parameters("CurrReplicatedTable") = strTableName
cmdUpdateRep.Execute

rstCheckRep.MoveNext

Loop

catLocal.Tables.Delete "tblBackEndReplicatedTables"

DoCmd.Echo True

'-- Clean up
rstCheckRep.Close

Exit Sub

Error_ap_CheckReplicatedTables:
MsgBox Err.Description
Exit Sub

End Sub

```

Let's see what happens after the back-end name and path are stored to variables:

1. The `rstCheckRep` recordset is opened off the `qryCheckBackEndReplication` query. If any records exist, the routine then loops through each table to replicate:


```
Set cmdUpdateRep = catLocal. _
    Procedures("qryUpdateLastReplication").Command
```

```
rstCheckRep.Open "qryCheckBackEndReplication", _
    CurrentProject.Connection, adOpenStatic
```

```
'-- If a table has been updated, loop through
Do Until rstCheckRep.EOF
```

2. The current replicated table is deleted from the front end, and the copy is imported from the back end, refreshing the Tables collection after each command. Finally, the parameter is set in the query that updates the LastReplication field for the current replicated table entry:

```
'-- Delete the current local table, and import the backend table
On Error Resume Next
strTableName = rstCheckRep!TableName
```

```
catLocal.Tables.Delete strTableName
catLocal.Tables.Refresh
```

```
On Error GoTo Error_ap_CheckReplicatedTables
```

```
DoCmd.TransferDatabase acImport, "Microsoft Access", pstrBackEndPath & _
    pstrBackEndName, acTable, rstCheckRep!TableName, rstCheckRep!TableName
```

```
catLocal.Tables.Refresh
```

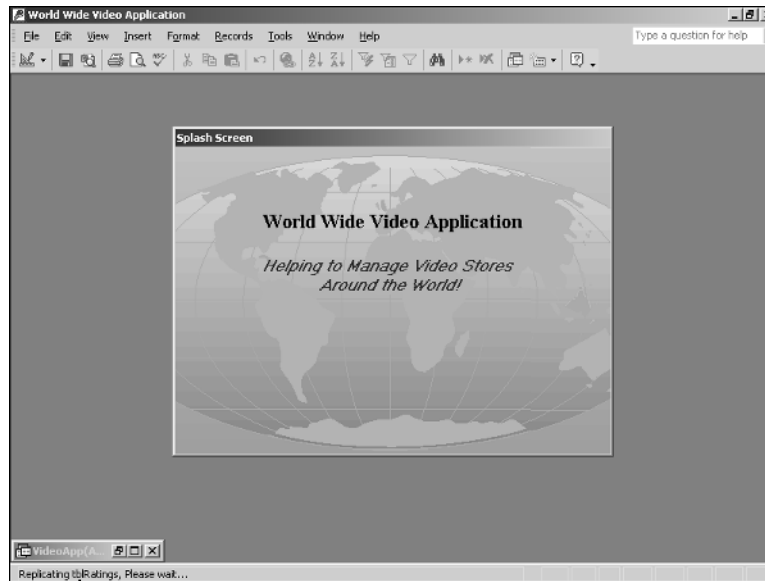
```
cmdUpdateRep.Parameters("CurrReplicatedTable") = strTableName
cmdUpdateRep.Execute
```

All that's left is to loop through the recordset and clean up. Again, the routine in Listing 26.9 tests whether a table that's flagged for replication has been updated on the back end since the last replication. Figure 26.17 shows the `ap_CheckReplicatedTables` routine in action.

The `ap_CheckReplicatedTables` routine is called from the `ap_AppInit` routine, which is called when the system is first started. (For more information about `ap_AppInit` and other startup system checking routines, see Chapter 25.) This segment of `ap_AppInit` calls `ap_CheckReplicatedTables`:

```
'-- Check for Replicated Tables
ap_CheckReplicatedTables
```

To check for updated tables more frequently, trigger the `ap_CheckReplicatedTables` routine off the `UserLogOutMonitor` form and have the system check every so often. However, it could be pretty slow to keep running that query every minute or so.



Message from ap_CheckReplicatedTables

FIGURE 26.17

When the system first starts, the routine should run.

Summary

Creating useful maintenance routines can be a boon to your applications. Handling situations that occur frequently by creating reusable routines and increasing performance make the system that much more enjoyable not only for users to use, but for you to work on. For more information about the topics brought up in this chapter, refer to the following chapters:

- Chapter 2, "Coding in Access 2002 with VBA," gives an overview of using VBA in Access.
- Chapter 4, "Working with Access Collections and Objects," discusses using Access collections.
- Chapter 25, "Startup Checking System Routines Using ADO," explains some of the other useful routines that can be used for making your applications as robust as possible.

INDEX

SYMBOLS

- ! (bang) character, 420
- !> operator, 804
- !< operator, 804
- != operator, 804
- \$ (dollar sign), 33
- % operator, 804
- & operator, 804
- * operator, 804
- > operator, 804
- >= operator, 804
- + operator, 804
- operator, 804
- ... character, 578
- / operator, 804
- 0-based collections, 58, 82
- 1-based collections, 82
- 16-bit API calls, 471
- 1stForms list box control, 529
- 32-bit API calls, 471
- < operator, 804
- <> operator, 804
- <= operator, 804
- = operator, 804
- ? CurrentUser(), 602
- ^ operator, 804
- _ (underscore), 33
- | operator, 804

A

Access

- add-ins, 512
- coding XML and VBA, 142
- compatibility, 744
- controls, morphing, 289
- cost per user, 744
- databases, upsizing, 764
- importing from XML, 142
- Jet, 110
- mdb files, 782
- multiuser functionality, 652.
 - See also* multiuser environments
 - Default Open Mode section*, 654
 - error handling*, 685-688
 - Number of Update Retries option*, 656
 - ODBC Refresh Interval setting*, 656
 - record locking*, 653
 - Refresh Interval setting*, 656
 - Update Retry Interval setting*, 656
- NorthwindCS Project, 795
- object replication, 706, 714
- ODBC front-end application, 744
- passing variables, 464
- replication identification fixup, 737
- restoring, 748
- security, 763
- syntax elements, 822
- Tab control, 282
- table design window, 798
- triggers, 813
- wizards, 512

Access 2000

- ADP, 780
- DAPs, 374

- synchronization conflicts, 719

Access 2002

- API Viewer, 468
- Application object, 871
- AutoLookup, 198
- Automation, 390
- clustered primary index, 230-231
- code libraries, 537
 - codeProject*, 541
 - CurrentProject*, 541
 - executing application routines*, 541
 - library database placement*, 538
 - pros/cons*, 538
 - setting references*, 539
 - viewing routines in the Object Browser*, 540

- DAO, 692

- DAPs, 357

- additional controls*, 378
- bound combo and list boxes*, 374
- Bound Span control*, 378
- buttons*, 362
- comparing to forms and reports*, 362-363
- creating*, 369
- Data Access Page Field List*, 369
- editing records*, 359-360
- expressions*, 374
- grouping*, 380-383
- hyperlinks*, 370-372
- Page wizard*, 365-366
- resources and information*, 384
- themes*, 368, 377
- wizards*, 364

- Data Outline window, 369
- error handling, 148

- Errors collection, 158

- exporting

- data to XML format*, 135, 139-141
- objects to HTML*, 587

- forms, 245

- background properties*, 248, 250
- Dirty event*, 246
- form Recordset property*, 245
- new features*, 243
- performing standard tasks*, 251

- hyperlinks, 574

- ... character*, 578
- editing properties*, 577-580
- Follow method*, 582
- FollowHyperlink method*, 584
- forms*, 575
- HyperlinkPart method*, 585
- IsHyperlink property*, 581
- options*, 586
- unbound controls*, 576

- installing add-ins, 518-520

- Internet features, 574

- linking to HTML files, 590-592

- multiuser environments, 652, 685-688

- Navigation control, 361

- Northwind databases, 73

- OLE DB providers, 783

- optimizing performance, 233
 - adding indexes*, 234-235
 - adjusting database structure*, 235
 - ambiguous field references*, 238

- pitfalls*, 236
- slow queries*, 237
- table relationships*, 233
- unconventional techniques*, 236
- PivotTable/PivotChart views, 243
- programming objects in
 - other applications, 390
- publishing to other Web file formats, 592
- queries, 188
 - action*, 199
 - advanced operations*, 201
 - naming conventions and documentation*, 191-192
 - new features*, 239
 - select*, 193
 - subqueries*, 205
 - user access*, 190
- replicas, 739
- Replication Manager, 728
- reports
 - formatting dynamically*, 347-352
 - snaking*, 323
- runtime error handling, 149
- stored procedures, 819
- wizards, 515
 - Bookmark Tracker Wizard*, 517
 - Combo Box Wizards*, 375
- XML. *See* XML
- Access Briefcase Reconciler**, 696
- Access Database Project**. *See* ADP
- Access object model**, 86
 - Application object, 86
 - adding custom properties*, 96
 - calling Windows APIs*, 92
 - CurrentProject properties*, 96
 - manipulating the active object*, 91
 - option values*, 89
 - quitting applications*, 90
 - screen painting*, 87
- Forms, Reports, and Data Access Pages collections, 101
- printer properties, 99
- References collection, 99
- Access Progress Bar**, 439
- Access Projects**. *See* Projects
- Access Replication submenu (Jet replication tool)**, 696-697
- Access Tab control**, 426
- accessing**
 - elements (collections), 56-57
 - Screen objects, 91
 - values (properties), 99
- AccessObject**, 93
- accounts**
 - Admin user, 601
 - default, 626-627
 - Jet database engine, 602
 - re-creating, 609-610
 - user, 602
- action queries**, 188-190, 199-201
- active objects**, 91
- ActiveX Common Dialog**
 - methods**, 486
- ActiveX controls**, 378, 424, 562
 - common, 424
 - drawbacks, 282
- ImageList control, 425, 428-429
 - adding images at run-time*, 431
 - adding images during design time*, 428
- ListView control, 433, 437
- emulating Windows Explorer, 434
 - properties*, 435
 - setting up manually*, 436
 - views*, 434
- ProgressBar control, 439-442
- references, 394
- Rich Textbox control, 445
 - code*, 448
 - loading files into control*, 449
 - properties*, 446
 - saving files from control*, 450
 - text alignment*, 449
- Slider control
 - properties*, 444
 - sizing text boxes at run-time*, 443
- StatusBar control, 450
 - properties*, 451
 - setting properties at run-time*, 452
- TabStrip control, 425-427
- ToolBar control, 453
 - button styles*, 455
 - creating*, 454
- TreeView control, 456
 - Nodes collection*, 459
 - populating*, 458
 - Style property*, 458
- ActiveX Data Objects**. *See* ADO
- ActiveX Data Objects 2.5 (ADODB)**, 110

ActualBM property, 495**ad hoc replica usage, 695****Add method, 81****Add-In Manager, 514, 520****add-ins, 512**

- creating, 515
- determining Access application type, 793
- extensions, 519
- installing, 518
 - USysRegInfo table, 519-520*
 - wizards with Add-In Manager, 520*
- .mda extension, 519
- storing, 521

AddBEToRegistry routine, 563**addhistory argument, 583****AddItem method, 100****AddMenu macro, 11****address book, 323****addresses (hyperlinks), 579****ade files, 794****adForwardOnly cursor type, 120****adLockPessimistic locking, 117****Admin user accounts, 601****Administer permission, 603-604****Administer property, 617****Administration Tools, Data Sources (ODBC) command (Start menu), 764****Admins group, 602, 632****ADO (ActiveX Data Objects), 54, 75, 110**

- ap_ADONewRecord() function, 674*
- ap_ADOOpenWithChoice() function, 676*
- converting from DAO, 76

Errors collection, 157

- ap_ErrorLog routine, 159*
- index base, 158*

examining recordset fields, 674**exporting data from Access to XML, 141****field data types and DAO equivalents, 129****libraries, 76, 114****limitations, 75****object models, 110**

- ADODB, 111*
- ADOX, 111*
- JRO, 113*

queries, 123

- creating parameterized, 124*
- deleting, 126*
- executing bulk, 125*
- modifying existing, 126*
- opening a recordset of parameterized queries, 124*

RecordCount property, 122**recordsets, 123**

- cloning, 123*
- creating, 116*
- locking, 118*
- navigating, 119*
- opening, 117*
- operations supported, 122*
- persistent, 120*
- properties used with Jet, 117*
- table-like, 118*

tables, 127

- creating new, 128*
- modifying existing, 130*
- OpenSchema method, 128*

Tables collection, 127

- temporary Command objects, 215
- unbound form example, 667

ADO Extensions 2.5 for DDL and Security (ADOX), 110**ADO.NET, 110****ADODB (ActiveX Data Objects 2.5), 110-111****adOpenStatic cursor type, 117****ADOX (ADO Extensions 2.5 for DDL and Security), 110-111****ADOX tables, 848****ADP (Access Database Project), 72, 780**

- compared to Access .mdb applications, 792-793
- files, 794
- queries, 788
- testing, 72

advanced SQL Server applications, 760**after reports, 328****AfterUpdate event, 527****AfterUpdate subroutine, 759****aggregation (records), 749**
algorithms (conflict resolution), 723-724**alias queries, 767****AliasName parameter, 464****All operator, 804****All Records locking mode, 663****AllDatabaseDiagrams object, 792****AllForms collection (CurrentProject object), 873****AllFunctions object, 792**

Allow Nulls property (SQL Server), 797
AllowAdditions property, 358
AllowDeletes property, 358
AllowEdits property, 358
AllQueries object, 792
AllStoredProcedures object, 792
AllViews object, 792
Alternate Color property, 358
Always Use Event Procedures check box, 25
Ambiguous field reference error, 238
Analyze, Documenter command (Tools menu), 766
Analyzer Wizards, 238
And operator, 804
anonymous replicas, 708
ANSI SQL Mode, 239
Any operator, 804
ap Applnit() function, 845
ap CheckLogOut() function, 836
ap LinkTables routine, 845, 847-848
apExitRoutine constant, 174
API (Application Programming Interface), 462
API calls, 462, 851-852
 API Viewer, 467-468
 ByVal keyword, 464
 cautions, 470
 converting 16-bit to 32-bit, 471
 creating declarations, 470
 declarations
 displaying folders within applications, 483-484
 FindExecutable() API routine, 472-475

GetUserNameA/GetComputerNameA, 482-483
 listing, 471
 routes, 463-466
 WNetAddConnectionA/WNetCancel-ConnectionA, 476-479
 WNetConnectionDialog/WNetDisconnection-Dialog, 479-481
 DLLs, 463
 examples, 471
 finding in Win32api.txt file, 468
 function-type, 464
 GPF, 462
 grouping in Win32api.txt file, 469
 modifying, 562
 Open File dialog, 486-487
 Registry, 555-556
 listing API Declarations and wrapper routines, 566
 registering back-end databases, 562
 sample application, 557
 working with code, 558
 returning string values, 475
 syntax, 463
 viewing possible, 466
 wrappers, 480
API routines, 849-850
API Viewer, 467-468
append queries, 200
Append query, 792
Application object (Access object model), 86, 658
 adding custom properties, 96
 calling Windows APIs, 92
 CompactRepair method, 871
 CurrentProject properties, 96

 FollowHyperlink method, 584
 ImportXML method, 143
 manipulating the active object, 91
 option values, 89
 Printer objects, 99
 quitting applications, 90
 screen painting, 87
Application Plus Web site, 580
Application Programming Interface (API), 462
Application.AccessError method, 161
applications
 Access
 databases, 780
 storing information, 545
 API calls, 467
 Automation, 390
 client/server, 795
 distributing, 772-776
 optimizing performance, 762
 planning, 750
 converting to SQL Server, 756-757
 creating, 72
 data sharing, 748
 DLLs, 463
 DSS, 747
 error handling
 cautions, 176
 custom error messages, 157
 environment changes, 176
 Err objects, 153-155
 Error objects, 153
 nesting error handlers, 181
 new options, 182

- On Error event, 179*
- rolling back transactions, 178*
- file-server versus
 - clients/servers, 742
- front-end (ODBC), 744
- Jet-based, 780-782
 - components, 789*
 - user/group security, 794*
- Office, 144
- OLTP, 747
- profiles, 544
- projects, 771-772
- quitting, 90
- referencing libraries, 115
- Registry, 545
- replication, 713
 - back-end/front-end, 719*
 - partial replicas, 715-718*
 - replicable and non-replicable objects, 714*
- running Access with
 - Automation, 418-420
- running from Access with
 - Automation, 397
 - Excel, 402, 404*
 - Outlook, 407-417*
 - Project, 405-407*
 - Word, 398-400*
- runtime error handling, 148
 - creating a simple handler, 149*
 - On Error command, 149*
 - ON Error Goto 0 statement, 150*
 - ON Error Goto statement, 149*
 - On Error Resume Next statement, 150*
 - Resume LineLabel statements, 153*
 - Resume Next statements, 152*
 - Resume statements, 150-151*
 - screen repainting, 88
 - secured, 629-630
 - security, 598-599
 - SQL Pass-Through queries, 760
 - SQL Server, 760
 - SQL-linked, 750
 - upsized, 770
 - users, 840
- ap_ADONewRecord() function, 674**
- ap_ADOOpenWithChoice() function, 676**
- ap_Applnit routine, 886**
- ap_AssignPermissions() function, 641**
- ap_CheckReplicatedTables routine, 884-886**
- ap_CompactDatabase form, 871**
- ap_CreateOLMailItem routine, 409**
- ap_CreateUser() function, 632**
- ap_ErrorCreateStatus-Message function, 688**
- ap_ErrorHandler function, 686**
- ap_ErrorHandler() function, 173**
- ap_ErrorLog routine, 159, 165**
- ap_FormIsOpen() function, 494**
- ap_GetOpenFileName API routine, 562**
- ap_GetRecordSource() function, 670**
- ap_GlobalRoutines module, 876**
- ap_OpenFile function, 559**
- ap_ProcessKeys routine, 302**
- ap_Requery_SearchSub() function, 256**
- ap_ShowErrorCVer() function, 162**
- ap_StandardCodeMain form, 872**
- ap_SystemUtilities form, 862**
- ap_UpdateReplicated-Backend editor, 875**
- ap_UpdateReplicated-Backend form, 879**
- architecture (ODBC), 743**
- ArgList parameter, 464**
- arguments**
 - Follow method, 583
 - methods, 51
 - Optional versus ParamArrays, 37
 - passing, 37-38
 - procedures, 32-33
- arrays**
 - collections (comparison), 80
 - integers, 84
 - limitations, 85
 - listings, 84
 - parameterized, 36-37
 - Section, 318
- ASP, 592-593**
- assigning**
 - objects variables, 52
 - permissions to objects, 614-615
- asynchronous errors, 42-43**
- attaching. See linking**
- auto requery, 208**
- Auto Statement Builder, 51**
- AutoExec macro, 11**

AutoFit method, 404

AutoLookup, 198

AutomateToAccess routine, 418

automation, 11, 390

cleaning up when finished, 397

CreateObject() function, 395-396

declaring object variables in VBA, 394

running Access from other applications, 418, 420

running other applications from Access, 397

Excel, 402, 404

Outlook, 407-417

Project, 405, 407

Word, 398-400

sessions, 391

setting up references to application object libraries, 392-393

VBA, 390

Automation servers, 92

AutoPage: Column feature, 364

Avg() function, 202

B

back ends, 652

compacting, 868-870

corrupt, 852-857

deleting from Registry, 571

error logging, 168

locating if links are broken, 486

Projects, 789

searching, 849-852

tables, linking/unlinking, 845-848

updating with errors found, 168

back-end design (replication applications), 719

background properties (forms), 248, 250

backups

before converting to replicable databases, 700

live backups, 747

recovering Design Masters, 707

replicas, 738

replication, 694

server databases, 747

Bad Code button, 216

bang (!) character, 420

base keys, 547

base pages (grouped DAPs), 380

before reports, 325-327

BeforeUpdate event, 527

Between operator, 804

bigint data type, 800

binary data type, 800

binding subforms to Source Object properties during runtime, 261

bit data type, 800

bitmaps, 428

blocks, 50

BMDescription property, 495

Bookmark Tracker, 492, 515

basic objects, 494

class modules, 494

clsBookmarkItems, 495

clsBookmark-

Management, 496-497, 500

features, 492

overview, 515

Bookmark Tracking Wizard, 517

installing, 521

programming, 522

cmdFinished button, 532-534

command buttons, 524

Design view, 522

fields and event procedures on the second page of the form, 529

fields and event procedures on the third page of the form, 531

initializing the wizard form, 523

switching pages of the Tab control, 526

validating fields on the first page of form, 527

BookmarkAction method, 499

adding bookmarks, 501

deleting bookmarks, 503

moving to bookmarks, 505

bookmarks, 123

adding, 501

deleting, 503

moving to, 505

storing, 499

storing information about, 495

trackers, 492

boolTrackingLevel variable, 701

bound hyperlinks, 371, 581

Bound Span control, 378

Briefcase (Jet replication tool), 695-696

builders, 512

built-in collections, 82

buttons

- captions, 843
- DAPs, 362
- hiding or showing, 642
- Modules page, 45
- Object Browser, 842
- unnecessary (DAPs), 361

ByRef, 37-38**ByVal, 37-38****C****Cachesize property, 120****calculated values (DAPs), 374****calculating percentages, 218****calendar entries, creating from Access, 416****calling**

- API calls, 851-852
- centralized error handlers, 173
- methods, 50
- modules, 67
- Open File dialog box, 559
- reports with different layouts, 314-317
- RequerySubform() function, 214
- routines located in one form from other forms, 258
- Windows APIs, 92

Cancel button**(ap_CompactDatabase form), 871****CancelUpdate method, 120****caption Section property, 358****captions (hyperlinks), 579****Cartesian products, 219****cascading deletions, 809****case sensitivity**

- logons, 611
- passwords, 600
- PID (personal identifier), 609
- upsizing tables, 752

Catalog object, 111**catalog variables, 847-848****Categories field, 224****CBF (code behind forms), 242****cboCategories combo box, 277****cboFileToFindExe_After-Update routine, 475****cboSections combo box, 529****cboTablesToEdit combo box, 876****cboTablesToEdit_Before-Update event procedure, 879****centralized error-handling routines, 171-172****Chap20.mdb file**

- secure databases, 647-648
- Web site, 633

char data type, 801**Chart Office Web Component, 378****check boxes, 25****check constraints (SQL Server), 804-806****CheckBMTAlreadyExists function, 528****CheckBMTNameDupe() function, 532****CheckFEVersion() function, 857****CheckUpdateTime routine, 878, 884****ChoiceDisplay text box, 258****Choose Builder dialog box, 25****class modules, 66-69**

- Bookmark Tracker, 494
- clsBookmarkItems*, 495
- clsBookmarkManagement*, 496-497, 500
- coding, 63-64
- creating, 65-66
- custom, 67

clauses (SQL), 749**Clear method, 154****client/server, 652**

- applications
 - building, 795
 - distributing, 772-776
 - optimizing performance, 762
 - planning, 750
- combo boxes, 754
- data, limiting, 753-754
- development (ODBC), 743
- forms, 758-759
- functions, 755
- joins, 755
- migration, 745-747
- network traffic, 748-749
- OLE objects, 755-756
- queries, 757-758
- static information, 756

cloning recordsets, 123**clsBookmarkItems, 495, 502****clsBookmarkManagement, 496-497**

- BookmarkAction method, 500
- Declarations area, 497
- InitBookmarks method, 498
- methods, 496

CLT (ColumnLevelTracking property), 699**clustered primary indexes, 230-231****cmdBrowse button, 559****cmdLink_Click routine, 569**

cmdPreview button, 314

cmdTestRaiseError_Click()
function, 155

cmdTextErrorCollection
command, 164

code, 11

- combining elements of
 - applications, 404
- comments, 42
- control-flow, 42
- converting macros to VBA, 15-18
- databases
 - compacting, 645-646*
 - decrypting, 645-646*
 - encrypting, 645-646*
 - setting passwords, 634-635*
- decision-making code, 39
- form reusability, 251
- groups
 - creating, 635-636*
 - deleting, 636-637*
- libraries, 537
 - CodeProject, 541*
 - CurrentProject, 541*
 - executing application routines, 541*
 - library database placement, 538*
 - pros/cons, 538*
 - setting references, 539*
 - viewing routines in the Object Browser, 540*
- macro-to-code changes, 12
 - code equivalents of macro commands, 14*
 - DoCmd object, 12*
- navigating, 45
- object owners, changing, 639-640

permissions

- checking, 641-642*
 - setting, 640-641*
- protecting, 599
- repeating, 39-42
- security, 630
- database passwords, 634-635*
 - databases, compacting, 645-646*
 - Documents collections, 631*
 - groups, 635-637*
 - object owners, changing, 639-640*
 - object permissions, setting, 640-641*
 - permissions, 631, 641-642*
 - programming with DAO (Data Access Objects), 630-631*
 - Shift bypass key, disabling, 646-647*
 - users (whom you're logged on as), 643*
 - Users and Groups collections, 631*
 - users, creating, 632-638, 643-645*
- Shift bypass key, 646-647
- troubleshooting, 42-43
- users
- adding/removing from groups, 637-638*
 - creating, 632-633*
 - database creation, 643-644*
 - deleting, 633-634*
 - query object creation, 644-645*
 - table object creation, 644-645*

code behind forms (CBF), 242

code blocks, 50

Code command (View menu), 59

code modules, 23

- behind forms, 23-26
- functions, 23
- standard modules, 27
- subroutines, 27

code tables, 871-873

CodeData object, 96

CodeProject object, 96

coding

- code modules, 23
 - behind forms, 23-26*
 - standard modules, 27*
- modules, 63-64
- MoveCurrentField() function, 344
- VBA in Access 2002, 142
- XML in Access 2002, 142

collections, 54-55, 80

- 0-based, 58
- Add method, 81
- arrays (comparison), 80
- built-in, 82
- creating custom, 80
- defining new, 80
- Documents
- security, programming with
 - DAO (Data Access Objects), 631
- elements
 - accessing, 56-57*
 - counting, 55-56*
- ImageList control, 432
- integers, 83
- listings, 83-84
- Printers, 100
- removing items from, 82
- user-defined, 55

- Users and Groups
 - DAO (Data Access Objects) hierarchy, 630
 - security, programming with DAO (Data Access Objects), 631
- column data types (SQL Server), 800**
- Column Headings property (crosstabs), 227**
- Column Name property (SQL Server), 797**
- ColumnLevelTracking property (CLT), 699**
- columns**
 - default constraints, 807
 - foreign key constraints, 808
 - primary key constraints, 807
 - replication, 699
 - specifying headings in crosstabs, 227
 - SQL Server, 797-798, 804
 - Unicode-enabled, 802
 - unique constraints, 809-810
- COM components, 789**
- Combo Box Wizard, 268-270, 375**
- combo boxes, 268**
 - client/server, 754
 - creating
 - InitBookmarks method*, 499
 - the row source*, 874
 - DAPs, 374
 - filling in (listing), 473
 - lookups, 269
 - NotInList event, 281
 - programming, 270-271
 - adding new items based on user input*, 280-282
 - displaying columns outside the control*, 277
 - example*, 273
 - requeruing all choices in a subform*, 276
 - union queries*, 272-273
 - UNION SQL statements*, 274
 - properties, 376
 - RowSource property
 - combo boxes, 754
 - unbound forms, 269
- command buttons**
 - Bookmark Tracking Wizard, 524
 - Command Button Wizard, 314
- Command objects**
 - ADODB, 111
 - creating temporary, 215
- commandBeforeExecute event, 244**
- CommandChecked event, 244**
- CommandEnabled event, 244**
- CommandExecute event, 244**
- commands**
 - File menu, 767
 - Insert menu, 760
 - macros, code equivalents, 14
 - Query menu
 - SQL Specific, Data Definition*, 762
 - SQL Specific, Pass-Through*, 760
 - Registry, 550-551
 - DeleteSetting*, 554
 - limitations*, 555
 - SaveSetting*, 551-553
 - runtime error handling, 149
 - Start menu, 764
 - Tools menu
 - Analyze, Documenter*, 766
 - Database Scripting*, 776
 - Database Utilities, Repair Database*, 831
 - Database Utilities, Upsizing Wizard*, 768
 - References*, 46
 - Security*, 600-601
 - Security and Workgroup Administrator command*, 607
 - View menu
 - Code*, 59
 - Object*, 26
 - Object Browser*, 45, 842
 - Properties*, 24, 629
 - Toolbox*, 24
- comments (code), 42**
- Common Dialog control, 486**
- Compact and Repair Database dialog box, 831**
- Compact Database method, 852**
- Compact utility, 738**
- compacting**
 - back ends, 868-870
 - databases, 645-646, 853-855
- CompactRepair method, 871**
- components (ODBC), 744**
- compression utilities, 606**
- concatenating strings, 37**
- conditional formatting in reports, 349-352**
- Configure Replication Manager Wizard, 729**
- configuring applications, 770**
- Conflict Viewer, 720**
 - alternative conflict-resolution algorithms, 723
 - functionality, 721
 - selecting correct record, 722

conflicts (replication), 719

- alternative conflict-resolution algorithms, 723
- Conflict Viewer, 720-722
- dialog warnings, 722
- identifying, 724
- Last-Update-Wins algorithm, 724
- list of, 726-727
- locking, 727
- referential integrity, 727
- simultaneous update, 726
- table-level validation, 727
- unique key, 727
- update-delete, 726

Connect event, 244**connecting**

- alternate databases, 116
- current database, 115

Connection object**(ADODB), 111**

- assigning, 115
- Errors collection, 157-158
- Errors
 - collectionap_ErrorLog routine, 159

Connection option (File menu), 782**connections**

- databases, 115
- network drives, 476

constants, 171**constraints (SQL Server), 803**

- check constraints, 804-807
- default constraints, 807
- foreign key constraints, 808
- primary key constraints, 807
- unique constraints, 809-810

contacts (Outlook), 412, 415**Control (generic object type), 53****control wizards, 513****control-flow code, 42****control-of-flow statements, 205****controllers, 390****controlling**

- parameters with enumeration, 33-35
- program flow, 39

controls, 266

- ActiveX, 424, 562
 - common, 424
 - ImageList, 425, 428-431
 - ListView, 433-437
 - ProgressBAR, 439-442
 - Rich Textbox, 445-450
 - Slider, 444
 - StatusBar, 450-452
 - TabStrip, 425-427
 - ToolBar, 453-455
 - TreeView, 456-459

- adding to forms, 24

- combo boxes, 268, 375

- adding new items based on user input, 280-282
 - displaying columns outside the control, 277
 - example, 273
 - lookups, 269
 - programming, 270-276

- copying from one form to another, 260

- creating, 534

- cursor movement, 300-301
 - ap_ProcessKeys routine, 302
 - handling the keyboard with VBA, 303

- DAPs, 378

- focus, 91

- iterating over, 57-58

- ListBox, 292

- getting/setting selected items from/to a table, 295-298
 - manipulating selected items with VBA, 294
 - multiselect mode, 293

- locking, 60

- manipulating with VBA, 304, 307

- listings, 308

- Option Group menu form, 305

- morphing, 289

- combo box/option button into a list box/check box, 291
 - design time, 289
 - examples, 290
 - runtime with VBA, 290

- referencing, 54, 57

- sizing, 443

- subform, 299

- Tab, 283

- type constants, 290

ControlSource property, 377**converting**

- applications to SQL Server, 756-757

- databases to a replicable format, 699

- backing up, 700

- creating additional replicas, 701

- input form (listing), 700

- listing, 700

- read-only replicas, 702

- macros to VBA code, 15-18

- queries, 771

copying

- Access customers into

- Outlook contacts, 413

- controls between forms, 260

forms, 104-105
 objects from the add-in database to the current application database, 535
 Project table column properties to the Clipboard, 799
 tables form back end to front end, 874
 listing, 877, 880
 replicated table editors, 875
 replication check routine, 883, 886
 updating front end tables, 882
 variables, 52
core tables (wizard-like interfaces), 336
corrupt backend databases, 852-857
corrupt front ends, 831
cost per user (Access), 744
Count() function, 202, 219
counter fields, 735
CREATE DEFAULT permissions, 764
Create New Data Source dialog box, 765
Create Relationship dialog box, 809
CREATE TABLE permissions, 764
Create Trigger statement, 812
CREATE VIEW statement, 761
CreateComboAfterUpdate-Event routine, 537
CreateControl() function, 534
CreateFormLoadevent method, 537
CreateItem method, 408
CreateNewReqKey wrapper routine, 567
CreateObject() function, 395-396
CreateProjectTable routine, 420
CreateReplica method, 701
Creating Secure Data Access Pages, 385
criteria
 PID (personal identifier), 609-610
 queries, 209
 user names, 609
cross-database joins, 755
crosstab queries, 752
 combining with a union query, 226
 creating a total row for, 223
 handling the data's main detail, 224
 specifying column headings, 227
Ctrl+clicking keyboard shortcut, 615
CurrentData object, 92-93
CurrentProject object, 92, 115
 AllForms collection, 873
 object collections, 93
 properties, 96
CurrentUser() function, 643
cursors, 300-303
custom class modules, 67
custom collections
 creating, 80-81
 defining new, 80
 integers, 83
 removing items from, 82
custom error logs, 163
 ap_ErrorLog routine, 165
 example, 164
 logging errors, 166-168

custom error messages, 173
custom messages, 66
custom methods, 63
custom properties, 47
 adding to Access objects, 96
 listing, 98
 tracking, 545
 writing, 59-62
CustomerSearchSubform subform, 257
CustomersSearchSubform form, 251
CustomerTabbedFormLateLoadExample form, 261
customizing
 forms, 58
 Registry entries, 232
 Tab control, 284
CVErr() function, 161

D

DAO (Data Access Objects), 54, 75, 110, 598, 630
 Access 2002, 692
 converting to ADO, 76
 data types and their ADO equivalents, 129
 libraries, 115
 security
 Documents collections, 631
 permissions, 631
 programming, 630-631
 Users and Groups collections, 631
 Users and Groups collection, 630
DAPs (Data Access Pages), 73, 134, 356
 Access version differences, 374
 additional controls, 378

- bound combo and list boxes, 374
 - Bound Span control, 378
 - buttons, 362
 - comparing to forms and reports, 362-363
 - creating, 369
 - Data Access Page Field List, 369
 - designing, 357
 - editing records, 359-360
 - example, 357
 - expressions, 374
 - grouping
 - adding a caption section*, 383
 - banded DAPs*, 383
 - base pages*, 380
 - creating group levels*, 382
 - Group Filter controls*, 383
 - relationships*, 381
 - hyperlinks, 370-372
 - bound*, 371
 - unbound*, 370
 - Navigation control, 360
 - properties, 359
 - resources and information, 384
 - structure, 356
 - themes, 368, 377
 - wizards, 364
 - Combo Box Wizard*, 375
 - Page wizard*, 365-366
- data**
- application data, 742
 - client/server
 - limiting*, 753-754
 - migration*, 745-747
 - exporting
 - from Access to XML*, 139-141
 - from XML to Access*, 142
 - loading into SQL Server, 776
 - migrating to clients/servers, 742
 - non-English, 802
 - refresh interval setting, 656
 - sensitive, 598
 - shared, 657
 - storing, 742
- Data Access Objects. See DAO**
- Data Access Page Field List, 369**
- data access page wizards, 514**
- Data Access Pages. See DAPs**
- data consumer (OLE DB), 781**
- data definition language (DDL), 762**
- Data Link Properties dialog box, 782, 790**
- data links**
- creating, 783-785
 - encapsulating connection information, 785
 - udl file extension, 783
- Data Outline window, 369**
- Data Page Size property, 359**
- data sharing across applications, 748**
- data sources**
- ODBC, 764-765, 772-775
 - OLE DB, 781
- Data Type property (SQL Server), 797**
- data types**
- variables, 29-30
 - Variant, 36
- data-conflict rule, 720**
- database diagrams, 788**
- Database Documentor Wizard, 239**
- database encryption, 605-607**
- Database object, 624**
- database passwords, 634-635**
- database permissions, 616-617**
- Database Scripting command (Tools menu), 776**
- database servers (ODBC), 744**
- Database Splitter, 660**
- Database Utilities**
- Repair Database command (Tools menu), 831
 - Upsizing Wizard command (Tools menu), 768
- Database windows (Access Projects), 787**
- databases**
- Access, upsizing, 764
 - application development, 557
 - back end, 852-857
 - changing for a Project, 782
 - Chap20.mdb file, 633
 - compacting, 645-646, 853-855
 - compacting the back end, 868
 - deleting the original database and renaming the temporary database*, 870
 - listing*, 868
 - compression utilities, 606
 - connecting to current, 115
 - connections, 116

- converting to a replicable format, 699
 - backing up*, 700
 - creating additional replicas*, 701
 - input form (listing)*, 700
 - listing*, 700
 - read-only replicas*, 702
- creating
 - code to deny*, 643-644
 - objects in*, 205
- decrypting, 645-646
- deleting duplicate records, 220
- design, 747
- encrypting, 619
 - code*, 645-646
 - performance*, 606
- Exclusive mode, 700
- Jet, 110
- linking
 - new databases to Projects*, 791
 - Projects to*, 790
- manually securing, 620-624
- multiuser environments, 657, 747
- All Records locking mode, 663
 - alternate locking schemes*, 665-666
 - built-in locking modes*, 661
 - Edited Records locking mode*, 663
 - locking modes in VBA*, 665
 - No Locks locking mode*, 664
 - splitting*, 658-659
- Northwind, 73
- OLE DB, 781
- opening connections, 115
- opening exclusively, 655
- passwords, 600-601
- queries
 - crosstab*, 223-224
 - distinguishing between new and old records*, 223
 - naming conventions and documentation*, 191-192
 - nesting groups*, 221
 - performance*, 227
 - resolution*, 228
 - Rushmore technology*, 229
- referencing, 46
- referencing TypeLibs, 393
- registering, 562
- reordering records, 232
- repairing, 831
- replicas
 - Design Masters*, 706-707
 - physical changes*, 708
 - replica set topologies*, 710
 - replica sets*, 710
 - singly connect list topology*, 710
 - star and hub topology*, 711-712
 - synchronizing*, 704-705
 - visibility*, 707
- replication, 692
 - alternative conflict-resolution algorithms*, 723
 - applications of*, 694
 - back-end/front-end applications*, 719
 - backups*, 738
 - Compact utility*, 738
 - Conflict Viewer*, 720-722
 - conflicts*, 719
 - counter fields*, 735
 - design goals*, 693
 - direct/indirect synchronizations*, 734
 - distributing replicable applications*, 713-714
 - identifying conflicts*, 724
 - Jet*, 694-699
 - last synchronization partners*, 737
 - Last-Update-Wins algorithm*, 724
 - MDE files*, 739
 - partial replicas*, 715-718
 - potential conflicts*, 726-727
 - read-only attributes*, 736
 - replication identification fixup*, 737
 - scheduled and on-demand synchronizations*, 734
 - security*, 739
 - specifying Access objects*, 714
 - synchronization phases*, 733
 - Synchronizer*, 728-732
 - synchronizing over the Internet*, 734
- security, 598, 647-648, 748
- server databases, backups/recovery, 747
- SQL Server
 - column data types*, 800
 - creating*, 776
 - non-English data*, 802
 - optimizing data access*, 813-815
- structure, 745
- tracking bookmarks, 492
- unsecuring, 624
- updating, 193
- VideoApp(ADO).mdb, 862

DataChange event, 244
DataSetChange event, 244
Datasheet view, 23
datetime data type, 801
DDE (Dynamic Data Exchange), 390
DDL (data definition language), 762
DDL queries, 205
Debug window, 602
debugging
 Access's default error handler, 148
 Echo method, 88
 forms, 23
decimal data type, 801
decision support system (DSS), 747
decision-making code, 39
Declarations area (clsBookmarkManagment), 497
Declarations sections (modules), 465
declaring
 API calls
 listing, 471
 wrapper functions, 480
 API routines
 creating declarations from scratch, 470
 displaying folders within applications, 483-484
 example, 466
 FindExecutable(), 472-475
 finding declarations, 466
 GetUserNameA/GetComputerNameA, 482-483
 listing, 465
 viewing possible calls, 466

WNetAddConnectionA/WNetCancel-ConnectionA, 476-479
 WNetConnectionDialog/WNetDisconnection-Dialog, 479-481
 enumerators, 34
 object variables in VBA, 394
 procedures, 31-32
 variables, 27-29, 396
 catalog variables, 847-848
 forcing, 29-31
 public, 47
decreasing network traffic, 748-749
decrypting databases, 645-646
default accounts, 626-627
default constraints, 807
Default Value property (SQL Server), 797
defaults values, 49-50
defining
 methods, 50-51
 new collections, 80
 queries, 228
 unique constraints, 810
Delete Data permission, 603
Delete Query, 792
DeleteAllReqEntriesForKey wrapper routine, 567
DeleteSetting command (Registry), 550, 554
deleting
 bookmarks, 503
 contacts in Outlook from Access, 415
 duplicate records, 220
 groups, 632, 636-637
 links (tables), 848

 queries, 126
 Registry entries, 558
 users, 632-634
 variables, 28
delimited identifiers, 799
Deploying Data Access Pages on the Internet, 385
design goals (replication), 693
Design Masters, 706
 Conflict Viewer, 721
 distributing replicable applications, 713
 making Access projects replicable, 714
 record manipulation, 706
 recovering, 707
design time
 loading images, 428
 morphing Access controls, 289
Design view
 creating union queries, 226
 triggers, 812
detaching data files, 791
detail information (reports), 319
detail queries, 221
detail reports, 314
dialog boxes
 Choose Builder, 25
 Compact and Repair Database, 831
 Create New Data Source, 765
 Generate SQL Scripts, 776
 Locate Backend Database, 850
 New Form, 24
 New Query, 760-762
 New User/Group, 611
 ODBC Microsoft SQL Server Setup, 765

- ODBC SQL Server Setup, 765
 - Print Table Definition, 766
 - Select Data Source, 767
 - Select Workgroup
 - Information File, 620
 - Set Database Password, 600
 - User and Group Accounts, 601
 - user input, 505
 - Dim versus Private/Public, 28**
 - Dir() function, 842-843**
 - direct exchange, 705**
 - direct synchronization, 733-734**
 - Dirty event, 246**
 - disabling Shift bypass key, 646-647**
 - Disconnect event, 244**
 - disconnecting network drives, 476**
 - displaying messages, 66**
 - Distinct keyword, 816**
 - Distributed Management Objects (DMO), 776**
 - distributing applications**
 - client/server, 772-776
 - replicable, 713
 - partial replicas, 715-718*
 - replicable and non-replicable objects, 714*
 - secured, 629-630
 - DLLs (Dynamic link libraries, 462-463)**
 - API calls, 464
 - commonly used, 463
 - registering, 550
 - routines, 463
 - DLookup() function, 867**
 - DMO (Distributed Management Objects), 776**
 - Do Until...Loop, 119**
 - Do...Loop, 40**
 - DoCmd object, 12, 92, 215, 252**
 - methods, 254
 - RunSQL method, 215
 - documentation (queries), 191-192**
 - documents**
 - mail merge functions, 398
 - XML
 - data display, 138*
 - exporting data from Access to XML, 140*
 - file types, 135*
 - listing, 136*
 - Documents collections, 631**
 - dollar sign (\$), 33**
 - downloading code library (example), 539**
 - drivers (ODBC), 743-744**
 - drop-down lists (DAPs), 375**
 - DSS (decision support system), 747**
 - Dynamic Data Exchange (DDE), 390**
 - dynamic grouping (reports), 320-321**
 - dynamic link libraries. See DLLs**
 - dynasets, 754**
- ## E
- Echo method, 88**
 - Edit Hyperlink dialog box, 578**
 - Edited Records locking mode, 663**
 - editing**
 - hyperlink properties, 577-581
 - records, 120, 359-360
 - Registry, 548
 - triggers, 812
 - editors**
 - VBE (Visual Basic Editor), 604
 - Windows Registry Editor, 608
 - elements**
 - collections, 56-57
 - counting, 55-56
 - embedded spaces (field/table names), 751**
 - enabling tracing (ODBC), 752-753**
 - encapsulating forms, 44**
 - encrypting databases, 605-607, 619**
 - code, 645-646
 - performance, 606
 - RSA RC4 technology, 605
 - enumeration, 33-35**
 - enumerators, declaring, 34**
 - environments**
 - multiuser, 627
 - replication, 629
 - Err objects, 153**
 - Clear method, 154
 - properties, 153-154
 - Raise method, 154-155
 - error handling**
 - exiting error handler, 150
 - testing links, 844-845
 - error messages, 798**
 - Error objects, 153**
 - ADODB, 111
 - properties, 157

errors

- asynchronous, 42-43
- custom error logs, 163
 - ap_ErrorLog routine*, 165
 - example*, 164
 - logging errors*, 166
 - logging to the back end first*, 168
 - updating the back end with errors found*, 168
- custom messages, 173
- Errors collection (ADO), 157
 - ap_ErrorLog routine*, 159
 - index base*, 158
- handling, 43. *See also* troubleshooting
- cautions*, 176
- centralized error-handling routines*, 171-172
- creating a simple handler*, 149
- custom messages*, 157
- environment changes*, 176
- Err objects*, 153-155
- Error objects*, 153
- multiuser environments*, 685-688
- nesting error handlers*, 181
- new options*, 182
- On Error event*, 179
- On Error Resume Next statement*, 150
- rolling back transactions*, 178
- runtime*, 148-153
- VBA*, 11
- Missing Reference, 393

- multiuser*, 174
- parameterized queries*, 216
- rolling back transactions*, 179
- synchronous*, 42-43
- user-defined*, 161-162
- values*, 171-172
- write conflicts*, 664

Errors collection (ADO), 157

- ap_ErrorLog routine*, 159
- index base*, 158

Errors collection (ADODB), 111**Eval() function, 400****event handlers, 25****events**

- PivotTable/PivotChart form
- views*, 244
- timer events*, 841

Excel

- running from Access with
 - Automation, 402, 404
 - XML Spreadsheet Schema (XML SS), 144

Exchange parameter, 705**exclusive databases, 652****Exclusive mode (databases), 700****executing bulk queries, 125****execution plans (queries), 228****Exists operator, 805****Exit Do statements, 40****Exit SubFunction command, 150, 154****exiting loops, 41-42****Expand control, 378****explicit permissions, 604****Export Registry File option (Registry menu), 549****Export XML dialog box, 139****exporting**

- Access objects to HTML, 587
- from Access to XML, 139, 141
- tables, 765-766, 862
 - ap_SystemUtilities form*, 863
 - DLookup() function*, 867
 - ExportTypeDefault field (ztblDatabaseProperties table)*, 865
 - file types*, 864
 - listing*, 865
- XML to Access, 142-143

Expression() function, 202**expressions (DAPs), 374****F****FetchDefaults property, 243****fields**

- displaying in list/combo boxes on DAPs, 377
- giving percentage of other fields, 218
- lookup property, 266-267
- naming, 751
- timestamp, 757

File menu

- commands, 767
- Connection option, 782
- options, 793

file-server applications versus clients/servers, 742**files**

- Chap20.mdb
 - secure databases*, 647-648
 - Web site*, 633

- flag files
 - creating*, 841-842
 - setting*, 841-843
- .mda extension, 607
- .mde, 630
- .mdw, 607
- Olddb.mdb, 646
- registering/unregistering, 549
- System.mdw, 607-608
- workgroup information, 607-608
 - creating*, 619-620
 - joining*, 620
 - users and groups, printing reports of*, 625
- workgroup information file key, 608
- Find dialog box (Registry Editor)**, 549
- FindExecutable() API routine**, 472, 474-475
- First() function**, 202
- flag files**, 841-843
- float data type**, 801
- FMS**, 515
- focus (controls)**, 91
- Follow method**, 582-583
- Followed Hyperlink Color option**, 587
- FollowHyperlink method**, 584
- For...Each statements**, 873
- forcing declarations**, 29-31
- foreign key constraints**, 808
- foreign key referential integrity conflicts**, 727
- Form Timer procedure**, 838
- Form view**, 23
- formatting**
 - reports
 - conditional formatting*, 349-352
 - dynamically*, 347
- Rich Textbox control, 446
- text boxes, 447
- forms**
 - accessing with hyperlinks, 575
 - ap_CompactDatabase, 868
 - ap_StandardCodeMain, 872
 - ap_SystemUtilities, 862
 - bookmarks
 - adding*, 501
 - deleting*, 503
 - moving to*, 505
 - bound hyperlinks, 581
 - calling reports with different layouts, 314-317
 - CBF (code behind forms), 242
 - client/server, 758-759
 - coding behind, 23-26
 - combo boxes, 272
 - compacting the back end, 868
 - comparing to DAPs, 362-363
 - controls
 - adding*, 24
 - controls between forms*, 260
 - morphing*, 291
 - creating, 24-26
 - controls*, 534
 - Load events*, 536
 - cursor movement, 300-301
 - ap_ProcessKeys routine*, 302
 - handling the keyboard with VBA*, 303
 - customizing, 58
 - DAPs. *See* DAPs
 - Datasheet view, 23
 - debugging, 23
 - docking toolbars, 453
 - encapsulating, 44
 - examining the code, 106
 - features, 245
 - background properties*, 248-250
 - Dirty event*, 246
 - form Recordset property*, 245
 - flash screens, 247
 - Form view, 23
 - functionality, exposing, 44
 - generic code table editors, 871
 - ImageList controls, 429
 - increasing performance, 242
 - lightweight, 104
 - loading, 103
 - locking modes, 661
 - lookup property, 266-267
 - managing multiple copies, 507
 - Me keyword, 102
 - multiple, 506
 - multiple instance support, 104
 - MultiSelect list box, 330, 333-334
 - opening
 - copies of the same form*, 506
 - multiple instances*, 106
 - performance, 249
 - performing standard tasks, 251
 - PivotTable/PivotChart views, 243
 - programming multiple copies of, 104-105
 - properties, 59-62
 - queries, 217
 - Record Source property, 189
 - referencing, 47, 102-103
 - reusing, 251-253, 257-259
 - OpenArgs property, 254-255

Section property, 318
 setting properties at runtime, 255
 tabbed, 282-283
 adding controls to pages, 286
 pages, 286
 performance, 260-261
 tracking multiple back ends, 558
 unbound, 269, 666
 adding records, 671
 clearing to create new records, 674
 example, 668-669
 getting the Key field from the form's Tag property, 671
 getting the record source, 670
 locating and loading records with, 675-678
 routines, 667
 saving records, 680, 683-685
 variables, 27-29
 WizExReports, 342
Forms collection, 102
Forms object, 624
Forms, Reports, and Data Access Pages collections, 101
Formula property (SQL Server), 798
Form_Open routine, 331, 342
FoxPro 2, 229
friendlyName property, 98
frmAutomationDemoCalls form, 397
frmBookmarkTracker-Example form, 494-496

frmBookmarkTracking-Wizard, 522
 fields and event procedures on the second page, 529
 fields and event procedures on the third page, 531
 Open event, 523
 txtBMTName text box, 531
 validating fields on first page, 527
frmFindExecutableExample, 472
frmMaintainBERegistry-Entries, 557
frmWindowscommon-ControlsRichText form, 446
frmWizExReports form, 336
 intCurrOrder variable, 340
 listing, 339
fromChild variable, 107
fromSectionsDisplay-Example form, 314
fromWizExReports form, 338
front end, 652
 applications (ODBC), 744
 corrupt, 831
 design (replication applications), 719
 Projects, 789
 updating, 857-859
function-type API calls, 464
functions, 31. See also methods
 Access-specific, 755
 ap_AppInit(), 845
 ap_CheckLogOut(), 836
 ap_AssignPermissions(), 641
 ap_CreateUser(), 632
 CheckFEVersion(), 857
 comparing VBA to T-SQL, 823-824

creating, 32
 CurrentUser(), 643
 Dir(), 842-843
 in-line, 763. *See also* user-defined functions
 IsMissing(), 36
 organizing, 23
 Reverse(), 41
 SysCmd(), 644
 T-SQL, 823
 TypeName(), 54
 user-defined, 763, 788
 client/server, 755
 SQL Server, 757

G

General Protection Fault (GPF), 462
Generate SQL Scripts dialog box, 776
generic code table editors, 871-873
generic object types, 53
Get External Data, Link Tables command (File menu), 767
GetAllSettings command (Registry), 550
GetComputerNameA API routine, 482-483
GetDefaultSetting() function, 555
GetObject() function, 395
GetOption method, 89
GetRegistrySubKeys wrapper routine, 567
GetSetting command (Registry), 550
GetString method, 117

GetSystemDirectory command, 468

GetSystemDirectoryA API routine, 483-484

GetTempPathA API routine, 483-484

GetUserNameA API routine, 482-483

GetWindowsDirectoryA API routine, 483-484

global replicas, 707

Globally Unique Identifiers (GUIDs), 708

GoToPage method, 260

GPF (General Protection Fault), 462

graphics

ImageList control, 428

reports, 329

GROUP BY clause (SQL), 749

Group By function, 202

Group Filter controls, 383

Group Interval property, 326

Group object (ADOX), 112

grouping

DAPs

adding a caption section, 383

banded, 383

base pages, 380

creating group levels, 382

Group Filter controls, 383

relationships, 381

queries to get percentages, 218

reports

MultiSelect list box,

328-330, 333-334

snaking reports feature, 326

wizard-like interfaces, 335-347

reports dynamically, 321

groups

Admins, 602, 632

creating, 611, 635-636

creating or deleting, 632, 636-637

objects, 618

removing, 612

reports of, 625

security, 601, 603

users

adding, 612-613

user accounts, 602

Groups collection (ADOX), 112

GUIDs (Globally Unique Identifiers), 708

H

handling errors, 42-43. *See also* troubleshooting
headerinfo string argument, 583

heterogeneous joins, 755

hidden objects, 519

hierarchies (Access object model), 86

HKEY_CLASSES_ROOT, 547

HKEY_CURRENT_CONFIG, 547

HKEY_CURRENT_USER, 547, 549, 564

HKEY_DYN_DATA, 547

HKEY_LOCAL_MACHINE, 547

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\Office\10.0\Access\Wizards, 513

HKEY_USERS, 547

HTML (Hypertext Markup Language), 575

exporting Access objects, 587

files, 356

linked tables, 592

linking files to Access, 590, 592

Hyperlink Address property, 576-578

Hyperlink Base property, 578

Hyperlink Color option, 587

Hyperlink controls, 378

Hyperlink data type, 579

Hyperlink SubAddress property, 576-578

HyperlinkPart method, 585
hyperlinks, 574

... character, 578

adding to DAPs, 370-372

adding to tables, 580

captions, 579

editing properties, 577-580

Follow method, 582

FollowHyperlink method, 584

forms, 575

HyperlinkPart method, 585

IsHyperlink property, 581

options, 586

type field, 581

unbound controls, 576

I

icons, 428

differences between modules, 494

ListView control, 434

IDC, 592

identifiers, 799

Identity Increment property (SQL Server), 798

Identity property (SQL Server), 798

Identity Seed property (SQL Server), 798

IDs

PID (personal identifier), 608-610
 SID (security identifier), 608
 WID (workgroup identifier), 609

If...Then statements, 39

Ilf() function, 321

Image control, 378

Image Count property, 431

image data type, 801

ImageList control, 424, 428

adding images at runtime, 431
 adding images during design time, 428
 Image Count property, 431
 Index property, 431
 loading with VBA (listing), 432
 properties, 429
 storing bitmaps, 456
 with TabStrip control, 425

images (ListView control), 433

implicit permissions, 604

importing

HTML files to Access, 590, 592
 XML into Access, 143

ImportXML method, 143

In operator, 805

in-line functions, 763. See *also* user-defined functions

in-place activation, 390

IncludeCategories() function, 211

IncludePurchaseDate() function, 211

indenting code, 42

Index property (ImageList control), 431

indexes

adding to existing tables, 130
 clustered primary, 230-231
 optimizing performance, 234-235
 rebuilding, 766
 Rushmore technology, 229

indirect synchronization, 729, 734

Indirect synchronization modes, 704

individual user profiles (Registry), 551

infinite loops, 40

INI files, 544

InitBookmarks method, 496

listing, 498
 parameters, 497

inner joins, 195

input forms, 700

input parameters (stored procedures), 820

insert buffers, 812

Insert Data permission, 603

Insert Hyperlink dialog box, 370

Insert menu, 760

insert triggers, 811

installing

add-ins, 518-520
 Bookmark Tracker Wizard, 521
 MSDE, 787
 USysRegInfo table, 519-520

int data type, 801

intCurrOrder variable, 340

integers

arrays, 84
 collections, 83

interfaces, 581, 608. See *also* user interfaces

Internet, 574

hyperlinks, 574
forms, 575
options, 586
 synchronizations, 734

Internet synchronization mode, 704

interoperability, 390

intRepEdited variable, 876

Is RowGuid property (SQL Server), 798

IsError() function, 161

IsHyperlink property, 581

IsMissing() function, 36

ItemData property (ListBox control), 293

ItemsSelected property (ListBox control), 293

iterating over controls, 57-58

J

Jet, 110

database connections, 116
 locking modes, 661
 OLE DB, 781
 queries
optimizing, 229-232
resolution, 228

- recordsets, 117
- replication tools, 694
 - Access Replication submenu*, 696-697
 - briefcase*, 695-696
 - JRO*, 699
 - Replication Manager*, 698
- splitting databases, 658
- system architecture, 657
- threads, 232

Jet and Replication Objects 2.5 (JRO), 110

Jet-based applications, 780, 782

- comparing data types with SQL Server, 803
- components, 789
- default value attributes, 807
- user/group security, 794

Jet database engine, 602

Jet-linked table SQL Server applications, 77

JetEngine object (JRO), 113 jets, 72

joining

- tables, 195-196
- workgroup information file, 620

joins, 755

JRO (Jet and Replication Objects 2.5), 110, 699

- contents, 113
- Retention Period, 732

JRO library (Jet and Replication objects, 692

K

Key property (ImageList control), 430

keyboard shortcuts, 615

KeyCode system parameter, 301

keys, 546

- Access applications, 549
- adding, 549
- clearing references to, 565
- predefined, 547
- Registry, 565
- Registry Editor, 548
- Shift bypass, 646-647
- subkey lists, 568
- subkeys, 547
- wizards, 513

keywords

- ParamArray, 37
- Private, 47
- Public, 47, 59
- While versus Until, 41

L

Label Wizard, 323

LANs, 653

last synchronization partners, 737

Last() function, 202

Last-Update-Wins algorithm, 724-725

Layout Wizard, 381

lboTablesToExport list box, 863

left joins, 195

Length property (SQL Server), 797

LibName parameter, 464

libraries

- ADO, 76
- code, 537
 - CodeProject*, 541
 - CurrentProject*, 541
 - executing application routines*, 541

- library database placement*, 538

- pros/cons*, 538

- setting references*, 539

- viewing routines in the Object Browser*, 540

- type, 113

- TypeLibs, 392-393

Library.ClassName syntax, 76

lightweight forms, 104

Like operator, 805

limiting data, 753-754

Link HTML Wizard, 591

LinkChildFields property, 277

linked servers, 755

linked tables, 719

- loop properties, 267
- securing, 627

linking

- Projects
 - to existing databases*, 790
 - to new databases*, 791
- servers to tables, 780
- tables, 568
 - back end*, 845-848
 - server tables*, 767
- views (SQL Server), 761-762

LinkMasterFields property, 277

links

- adding to DAPs, 370-372
 - bound*, 371
 - unbound*, 370
- table links
- testing at startup, 843-845
- tables, 848
- testing, 844-845

list boxes

- DAPs, 374
- frmWizExReports form, 338

listBoundField property,
377

ListBox controls, 292-298

ListDisplayField property,
377

ListFiles() function, 474

listings

- Access Progress Bar, 439
- Access Tab control, 426
- ActiveX, TabStrip control, 427
- API calls, 851-852
 - calling Network Connect dialog box, 481*
 - calling Network Disconnect dialog box, 481*
 - calling Open File dialog box, 559*
- Common Dialog ActiveX methods, 486
- connecting network drives programmatically, 478
- declarations, 472
- disconnecting a network drive, 479
- displaying associated files, 474
- displaying folders within applications, 484
- filling in a combo box, 473
- opening requested files, 475
- retrieving computer names, 483
- retrieving filename through Open File dialog, 487
- retrieving user names, 482

API declarations, 465

ap_ShowErrorCVer() function, 162

ap_StandardCodeMain form, OnLoad event, 872

Automation

- creating Access tables from Project, 419*

- custom letters in Word, 401*

- Excel, 403*

- Outlook, 409-416*

- Project, 406*

- Word, 398*

Bookmark Tracking Wizard, 523-524

- cmdFinished_Click event procedure, 532*

- command buttons, 524-525*

- required fields, 525*

- Tab control, 526*

BookmarkAction method, 500

bookmarks

- Bookmark Tracker, 498*

- deleting, 504*

- evoking*

- clsBookmarkManagement methods, 496*

- managing multiple on a given form, 516*

- re-creating combo box*

- from the colBookmarks collection, 502*

- storing information*

- about, 495*

buttons, modifying captions, 843

calling custom class modules, 67

cboSections combo box, building the row source, 529

CheckBMTAlreadyExists function, 528

CheckBMTNameDupe() function, 532

collections

- comparing to arrays, 84*
- example, 83*

combo box controls, 281

Common Dialog ActiveX methods, 486

controls

- arranging command buttons, 309*

- cursor movement, 303*

- morphing a combo box/option button into a list box/check box, 291*

copying objects from the add-in database to the current application database, 535

CurrentProject object, 94

custom error logs, 163

custom messages, displaying, 66

database creation permissions, 643-644

databases

- canceling compaction, 871*

- code to compact, 645-646*

- code to decrypt, 645-646*
- compacting, 868*

- compacting/repairing, 853-855*

- setting passwords, 634-635*

declaring API routines, 566

- error handling
 - calling the centralized error handler, 173*
 - calling the error log routine, 164*
 - centralized error routines, 174*
 - checking local error log, 169*
 - controlling returned errors, 161*
 - creating a simple handler, 149*
 - custom error logs, 163*
 - custom error messages, 156*
 - forms, 180*
 - logging errors, 159, 166*
 - looking for multiple errors, 159*
 - opening forms, 102*
 - passing back errors, 162*
 - raising an error on purpose, 155*
 - resetting environment settings, 176*
 - Resume LineLabel statement, 153*
 - Resume statements, 151*
 - rolling back transactions, 179*
- errors
 - displaying error messages, 688*
 - locking records, 687*
 - trapping multiuser errors, 686*
- flag files, 841-842
- forms
 - closing directly, 509*
 - controlling keystrokes, 301*
 - creating target form's Load event, 536*
 - loading a subform control at runtime, 262*
 - locating the record for the choice made, 258*
 - managing multiple copies, 507*
 - opening in Print Preview, 315*
 - opening multiple instances, 106*
 - requering the StandardSearch subform, 256*
 - selecting and deselecting all elements, 345*
 - setting properties at runtime, 255*
- frmWizExReports form, 339
- groups
 - code to create, 635-636*
 - code to delete, 636-637*
- hyperlinks
 - Follow method, 583*
 - FollowHyperlink method, 584*
 - HyperlinkPart method, 586*
- ImageList control, loading with VBA, 432
- InitBookmarks method, 498
- links, testing, 843-844
- ListBox control
 - displaying selected table names, 298*
 - noting previously chosen tables, 296*
 - saving currently selected items, 298*
 - ShowNamesChosen() function, 294*
- ListView controls, 437
- loading available printers into list boxes, 100
- logging out users, 840
- LogTimes procedure, 746
- MacrosToConvert macro (VBA conversion), 17
- modifying option settings, 89
- MoveCurrentField() function, 343
- MultiSelect list box, 331
 - deselecting the categories and clearing the Temp table, 334*
 - selecting all categories and adding them to the Temp table, 333*
 - using the selected collection of the lboGetCategory list box, 332*
- objects
 - adding custom properties, 98*
 - showing friendly names, 97*
- ODBC data sources, 773-775
- Open Group menu form, 309
- OpenSchema method, 128
- Option Group menu form, 307
- owners of objects, 639-640
- parameterized queries, 124
- permissions of objects
 - code to check, 641-642*
 - code to set, 640-641*
- PreviousControl property, 91
- printers, displaying properties, 100
- ProgressBar control, 442

- queries
 - code to deny creating objects, 644-645*
 - combining user choices and updating the subform's RecordSource property, 209*
 - creating and opening, 207*
 - runtime, 746*
 - Raise method (Err object), 155
 - recordsets
 - feature example, 119*
 - opening of a parameterized query, 125*
 - persistent, 121*
 - References collection, 99
 - Registry
 - adding new settings, 554*
 - creating default settings, 551*
 - creating new keys with passed arguments, 565*
 - creating/removing settings for current applications from tables, 552*
 - deleting back end from, 571*
 - parsing filename and path to add entries to, 563*
 - replication
 - conflicts, 724-725*
 - converting databases to replicas, 700*
 - creating additional replicas, 703*
 - direct exchanges, 705*
 - input form for making a database replicable, 700*
 - partial replicas, 717*
 - synchronizing star and hub replicas, 712*
 - reports
 - adding a format condition in code, 352*
 - alternating colors in a report section, 348*
 - multiple layouts, 316*
 - RequerySubform() function, 214
 - Resume LineLabel statement, 153
 - Resume statements, 151
 - reusable Search dialog, 253
 - Rich Textbox control, 449
 - changing text formatting, 448*
 - saving files from control, 450*
 - text alignment, 449*
 - rptDynamicGroupingExample report, 321
 - searching back ends, 849-850
 - Shift bypass key, 646-647
 - Slider control, 444
 - SQL criteria string, creating, 211
 - startup system checking routine, 831-835
 - StatusBar control, 453
 - stored procedures, 821
 - subroutines, 759
 - Tab control, 288
 - table objects, 644-645
 - Table type objects, 127
 - tables
 - checking for updated tables, 884*
 - cleaning up after replication, 879*
 - editing, 875*
 - exporting, 866*
 - linking in back end, 846-847*
 - replication, 877, 880*
 - with fields and keys, 128*
 - ToolBar control, swapping source objects based on a button, 455
 - TreeView control, populating, 458
 - triggers, incrementing a quantity value in a related table, 812
 - unbound forms
 - clearing to create a new record, 672*
 - getting the form's record source, 670*
 - getting the Key field from the form's Tag property, 671*
 - loading existing information, 679*
 - lookup fields, 676*
 - opening and executing a query, 677*
 - saving records, 681*
 - UseBeep properties, 68
 - users
 - code to create, 632-633*
 - code to delete, 633-634*
 - groups, adding removing, 637-638*
 - logging out, 838*
 - VideoApp(ADO).mdb, 831-834
 - Windows Open File dialog box, 559
 - XML
 - example document, 136*
 - Web display, 138*
- ListRowSource property, 377**

ListView control, 424-426, 433

- creating and filling using VBA, 437
- emulating Windows Explorer, 434
- properties, 435
- setting up manually, 436
- views, 434

live backups, 747**load balancing (replication), 694****Load events, 536****loading**

- data into SQL Server, 776
- Excel spread sheets from Access, 403
- existing information (unbound forms), 679
- forms, 103
- records with unbound forms, 675

local replicas, 707**local tables, 756****Locate Backend Database dialog box, 850****locked files (.ade), 794****LockForm property, 60-62****locking**

- alternative schemes, 665
 - roll-your-own scheme*, 665
 - unbound forms scheme*, 666
- conflict, 727
- forms, 60
- pessimistic, 663
- recordsets, 117-118, 123

locking records

- alternate locking schemes, 665
 - roll-your-own scheme*, 665
 - unbound forms scheme*, 666

built-in locking modes, 661

All Records, 663

Edited Records, 663

No Locks, 664

row-level versus page-level, 661

VBA, 665

logging errors, 166**logging out users**

- applications, 836-843
- checking necessity, 838-839
- warning, 839-840

logon names, 601, 611**logs**

- error, 163
- transaction logs, 769

LogTimes procedure, 746**lookup property, 266**

- combo boxes, 269
- setting at table field level, 267

loops, 39-42

- Do...Loop, 40
- exiting, 41-42
- infinite, 40
- reversing, 41
- While...Wend, 41

lstBackends list box control, 558

M

macros, 10

- AutoExec, 11
- code, 11
- controlling screen repainting, 88
- converting existing macros to VBA code, 15-18
- macro-to-code changes, 12-14
- when to use, 10

Macros object, 624**MacroToConvert macro, 15****Mail Application****Programming Interface (MAPI), 411****mail merge documents, 398****mailing labels, 323****make table queries, 199****MakeReplicable method, 699-700****managing**

- copies of forms, 507
- client/server queries, 757-758
- security through code, 630
 - databases*, 634-635, 645-646
 - Documents collections*, 631
 - groups*, 635-637
 - object owners*, 639-640
 - permissions*, 631, 640-642
 - programming with DAO*, 630-631
 - Shift bypass key, disabling*, 646-647
 - users*, 632-634, 637-638, 643-645
 - Users and Groups collections*, 631
- stored procedures, 763

MAPI (Mail Application Programming Interface), 411**Max() function, 202****.mda extension, 519 607****MDB (Microsoft Database), 72-73****.mdb files, 780, 794****MDE files, 739****.mde extension, 519, 630****.mdw file extension, 607**

Me keyword, 102

menus, 11

messages, custom, 66

methods

arguments, 51

calling, 50

clsBookmarkManagement,
496

Compact Database, 852

custom, 63

defining, 50-51

DoCmd object, 13, 254

objects, 44

referencing, 50

Microsoft Active Server

**Pages Output Options
dialog box, 593**

Microsoft Data Engine. See

MSDE

Microsoft Database. See

MDB

Microsoft Office Developer.

See MOD

Microsoft SQL Server

**Database Wizard dialog
box, 791**

Microsoft SQL Server

Desktop Engine. See
MSDE

migration

client/server, 745-747

data, 742

Min() function, 202

**Missing Reference error,
393**

MOD (Microsoft Office

Developer), 424

ActiveX Common controls,
424

API Viewer, 467

replication option, 728

modAutomationDemos

module, 394

**modBERegistryRoutines,
558, 563**

**modFormUtilities module,
494**

**modGlobalRoutines mod-
ule, 884**

**Modify Design permission,
603**

modifying

API calls, 562

captions (buttons), 843

existing queries, 126

Printer object properties, 99

Registry, 548

Registry settings, 233

Tab control, 284

tables, 797

**modOpenFileAPIRoutines
module, 558**

modOpenFileRoutine

ap_OpenFile function, 559

**modRegistryAPIRoutines,
558, 564**

module level variables, 27
modules

class modules, 66-67, 69

coding, 63-64

creating, 65-66

Declarations section, 465

standard modules. *See stan-*
dard modules

VBE, 604, 615

Modules object 624

Modules page, 45

**modUtilityRoutines mod-
ule, 558**

money data type, 801

morphing controls, 289-291

Moveable property, 243

MoveCurrentField() func-
tion, 342-344

Movie control, 378

**MovieTitlesSearchSubform
form, 251**

moving to bookmarks, 505

MSDE (Microsoft SQL

**Server Desktop Engine),
72, 786-787**

MsgBox() function, 14

MSysReplicas table, 713

**multiple-form instances,
105**

**MultiSelect list box,
328-330, 333-334**

**multiselect mode (ListBox
controls), 293**

**MultiSelect property
(ListBox controls), 293**

**multiuser environments,
653**

back end, 652

built-in locking modes, 661

All Records, 663

alternate locking

schemes, 665-666

Edited Records, 663

No Locks, 664

row-level versus page-
level locking, 661

VBA, 665

client/server, 652

databases, 657-659

Default Open Mode section,
654

error handling, 685-688

exclusive, 652

front end, 652

LANs, 653

linked tables, 627

Number of Update Retries

option, 656

record locking, 653

settings, 656

shared, 652

WANs, 653

N

named parameters, 51-52

names

- logon, 601
- usernames, 601
- users, 609

Namespace Object, 411

naming

- fields, 751
- objects, 46
- SQL Server Projects, 799
- tables, 751
- variables, 29

navigating

- code, 45
- recordsets, 119

Navigation control, 360

nchar data type, 801

nested queries, 204

networks

- ODBC, 744
- routines, 469
- traffic, 748-749

New button (Modules page), 45

New Form dialog box, 24

New keyword, 81, 396

New Query dialog box, 760, 762, 816, 819

New User/Group dialog box, 611

newwindow argument, 583

No Locks locking mode, 664

No value given for one or more required parameters error, 216

Nodes collection, 459

Northwind databases, 73

NorthwindCS Project, 796, 795, 805

Not operator, 805

NotInList event, 281

ntext data type, 801

numeric data type, 801

nvarchar data type, 801

O

Object (generic object type), 53

Object Browser (VBA), 44-46, 393, 540

Object Browser button, 842

Object Browser command (View menu), 45, 842

Object command (View menu), 26

object models

- ADO, 110-113
- referencing libraries, 113

Object type variable, 390

Object variables, 395

object-oriented features (VBA), 44

object-valued properties, 63

objects, 44

- Access object model, 86
- active, 91
- ADO unbound form example, 667
- Bookmark Tracker, 494
- collections. *See* collections
- copying, 535
- creating, 626-627
- DAO (Data Access Objects), 598, 630-631
- Database, 624
- declaring variables, 394-396
- Forms, 624
- Forms, Reports, and Data Access Pages collections, 101
- friendly names, 97

groups, 618

hidden, 519

in-place activation, 390

libraries, 115

locally kept, 658

Macros, 624

MDB, 73

methods, 44

Modules, 624

naming, 46

OLE, 755-756

owners

changing, 617-619

code to change, 639-640

passwords, 614-615

permissions, 603

assigning, 614-615

code to set, 640-641

programming with, 47

properties, 44

Queries, 623, 644-645

references, 46

replicable, 706

Replicable parameter, 714

Reports, 624

searching, 46

SQL Server, 788, 799-800

support objects, 64-65

table, 644-645

Tables, 623

types, 53-54

variables

assigning, 52

setting to Nothing, 397

viewing, 45

ODBC (Open Database Connectivity), 742-744, 780

ODBC data source, 764-765, 772-775

ODBC Driver Manager, 744

ODBC Refresh Interval setting, 656

ODBC SQL Server Setup
 dialog box, 765
ODBC tracing, 752-753
Office Web Components,
 574
.olb file extension, 46
Olddb.mdb file, 646
OLE DB, 780
 data links, 783-785
 providers, 781, 784
 terminology, 781
OLE objects, 755-756
OLTP (online transaction
processing), 747
on delete referential
integrity conflicts, 727
On Error command, 149
On Error event, 179
On Error Goto 0 state-
ments, 150
On Error Goto statements,
 149
On PivotTable Change
event, 244
On Selection Change event,
 244
on update referential
integrity conflicts, 727
On View Change event,
 244
online transaction process-
ing (OLTP), 747
Open Database
 Connectivity. *See* ODBC
Open Exclusive permission,
 604
Open Exclusive property,
 617
Open File dialog box,
 486-487, 559
Open Group menu form,
 309

Open/Run permission,
 603-604
Open/Run property, 617
OpenArgs property, 252,
 255
opening
 forms, 106
 recordsets, 117
OpenSchema method, 128
operators, 53-54
optDisplay option group,
 317
optGroupBy option group,
 320
optimistic locking, 664
optimizing
 Access 2002 performance,
 233
 adding indexes, 234-235
 adjusting database
 structure, 235
 ambiguous field refer-
 ences, 238
 pitfalls, 236
 slow queries, 237
 table relationships, 233
 unconventional tech-
 niques, 236
 data access (SQL Server),
 813-815
 performance, 762
 queries, 227-232, 758
Option Explicit (variable
declaration), 29-31
Option Group menu form,
 305-307
Optional arguments, 37
optional parameters, 35-36
Or operator, 805
organizing functions, 23

Outlook

creating a mail item, 409
 running from Access with
 Automation, 407, 411, 414
 adding contacts, 412
 calendar entries,
 416-417
 deleting contacts, 415
 mail items, 409
 task items, 411
 task items, 411
Outlook.olItemType enum,
 408
output parameters, 820
owner's permissions,
 627-629
owners
 objects, 617-619, 639-640
 queries, 628
ownership, 605

P

Page Break control, 260
Page objects, 715
Page wizard (DAPs),
 365-366, 379
pages-level locking, 661
ParamArray keyword, 37
ParamArrays, 37
parameter queries, 216
parameterized arrays,
 36-37
parameterized queries, 124
Parameterized query, 792
parameters
 ap_AssignPermissions()
 function, 641
 behavior, 38
 controlling, 33-35
 function-type API calls, 464

methods

CreateReplica, 701*InitBookmarks*, 497*Synchronize*, 704

optional, 35-36

specifying named, 51-52

stored procedures, 820

partial replicas, 715-718**passing**

arguments, 37-38

variables, 37

passwords, 601

adding to user accounts, 613

case sensitivity, 600

databases, 600-601, 634-635

PID, 610

protecting, 600

removing, 613-614

user interfaces, 610

percentages, 218**performance**

Access 2002, 233

adding indexes, 234-235*adjusting database structure*, 235*ambiguous filed references*, 238*pitfalls*, 236*slow queries*, 237*table relationships*, 233*unconventional techniques*, 236

client/server applications, 762

compression utilities, 606

encrypted databases, 606

forms

background pictures, 249*increasing*, 242

queries, 227

reports, 341

SQL Server Projects, 813

tabbed forms, 260-261

table replication, 874

Performance Analyzer**Wizard, 239****permissions**

Administer, 603-604

checking code, 641-642

CREATE DEFAULT, 764

CREATE TABLE, 764

databases, 616-617

Delete Data, 603

explicit, 604

implicit, 604

Insert Data, 603

Modify Design, 603

Modules object, 624

objects, 603

assigning to, 614-615*code to set*, 640-641*Database object*, 624*Forms object*, 624*Macros object*, 624*Queries object*, 623*Reports object*, 624*setting on*, 614-615*Tables object*, 623

Open Exclusive, 604

Open/Run, 603-604

owner's, 627-629

Read Data, 603

Read Design, 603

security, 603-605, 631

setting, 623-624

SQL Server, 763

Update Data, 603

persistent recordsets,**120-121****pessimistic locking, 663****PID (personal identifier),****608-610****PIMS (Personal Information****Management Systems),****578****Pivot Table Office Web****Component, 378****pivot tables, 378****PivotChart View, 239, 243****PivotTable View, 239, 243****PivotTableChange event,****244****planning**

client/server applications, 750

security, 626

populating

combo boxes, 474

tables, 756

TreeView control, 458

Precision property (SQL Server), 797**predefined keys, 547****PreviousControl property, 91****primary indexes, 230-231****primary key constraints, 807****Print Table Definition dialog box, 766****Printer objects, 99-100****Printer property, 243****Printers collection, 100****printing**

multiple topics, 328

user and group reports, 625

Priority argument, 701**Private keyword, 28, 47****private procedures, 32****private variables, 28****procedure level variables, 27****Procedure object, 112****procedures**

arguments, 32-33

declaring, 31-32

Form Timer, 838

functions. *See* functions

LogTimes, 746

private, 32

- scope, 32
- stored procedures, 762-763
- subroutines. *See* subroutines
- Procedures collection, 112**
- processing queries, 752-753**
- profiles, 544**
- program flow, 39**
- programming**
 - Bookmark Tracking Wizard, 522
 - cmdFinished* button, 532-534
 - command* buttons, 524
 - Design* view, 522
 - fields and event procedures*, 529-531
 - initializing the wizard form*, 523
 - switching pages of the Tab control*, 526
 - validating fields*, 527
 - combo boxes, 270-271
 - adding new items*, 280, 282
 - displaying columns*, 277
 - example*, 273
 - requering choices*, 276
 - union queries*, 272-273
 - UNION SQL statements*, 274
 - controls, 304-308
 - forms, 104-105
 - ListBox controls, 292
 - getting/setting selected items from/to a table*, 295-298
 - manipulating selected items with VBA*, 294
 - multiselect mode*, 293
 - objects, 47, 390
 - partial replicas, 717
 - relationship filters, 718
 - security, 630-631
 - stored procedures, 818
 - Tab control, 287
- Programming Data Access Pages, 385**
- Programming Jet Security white papers, 626**
- programming languages, 762**
- Progress control, 424**
- ProgressBar control, 439-442**
- Project, 405-407, 419**
- Project Properties dialog box, 794**
- Projects (Access), 750**
 - Access, 789, 792-793
 - ADPs, 780, 792-793
 - architecture, 795
 - components, 789
 - connecting to existing SQL Servers, 787
 - copying table column properties to the Clipboard, 799
 - creating, 789-791
 - from applications*, 771-772
 - overwriting existing files*, 792
 - workstation names*, 790
 - Data Link Properties dialog box, 782
 - data links, 783-785
 - Database window, 787
 - modifying tables, 796
 - linking, 781-784, 790-791
 - MSDE, 786
 - NorthwindCS, 795
 - objects, 799
 - primary key constraints, 807
 - properties, 792
 - retrieving data from servers, 795
 - security, 794
 - stored procedures, 818
 - Access and SQL Server syntax comparison*, 822
 - attributes*, 818
 - creating*, 818-819
 - parameters*, 820
 - returning values*, 820
 - testing parameters*, 821
 - table triggers, 813
 - .udl file extension, 783
 - views, 815
- Promoting fields, 382**
- properties**
 - Administer, 617
 - ADO recordset properties
 - used with Jet, 117
 - columns (SQL Server), 797-798
 - combo boxes, 376
 - CurrentProject objects, 96
 - custom, 47
 - adding to Access objects*, 96
 - writing*, 59-62
 - Err objects, 153-154
 - Error objects, 157
 - ImageList control, 429
 - ListView control, 435
 - LockForm, 60-62
 - Navigation control, 360
 - object-valued, 63
 - objects, 44
 - Open Exclusive, 617
 - Open/Run, 617
 - Printer objects, 100
 - Projects, 792
 - read-only, 62
 - record sources, 358
 - Rich Textbox control, 446
 - RowSource (combo boxes), 754
 - Run Permissions, 628-629

- setting, 50
- Slider control, 444
- StatusBar control, 451
- Tab control, 285
- Timer Interval, 837
- tracking system properties, 834
- UseBeep, 68
- user-definable, 254
- values, 48-50
- Properties command (View menu), 24, 629**
- Property Get subroutine, 59**
- Property Let subroutine, 59**
- property sheets, 48**
- property wizards, 514**
- protecting**
 - code, 599
 - modules in VBE, 604
 - passwords, 600
 - sensitive data, 598
- Protection page (Project Properties dialog box), 794**
- provider (OLE DB), 781**
- providers, 781, 784**
- pstrBackEndPath variable, 836**
- Public keyword, 28, 47, 59**
- public variables, 47**

Q

- QBF (query by form), 206-207, 213, 320**
- qryCheckBackEndReplication query, 883**
- qryMultiSelectCategory-Example query, 329**
- qrySectionsDisplayExample query, 315**
- qryUpdateReleaseDate query, 125**
- queries, 188. See also views**
 - Access 2002, 188
 - action, 199-201
 - ADP, 788
 - advanced operations, 201
 - alias queries, 767
 - auto requery, 208
 - AutoLookup feature, 198
 - binding subforms to, 245
 - bulk query operations, 653
 - Cartesian products, 219
 - client/server, 757-758
 - combo boxes, 273
 - compiling SQL statements, 228
 - converting, 771
 - creating
 - ahead of time, 217*
 - parameterized, 124*
 - SQL Server, 758*
 - criteria, 209
 - crosstab, 223-224, 752
 - DDL, 205
 - defining, 228
 - deleting, 126
 - distinguishing between new and old records, 223
 - documentation, 191-192
 - driving reports and forms, 217
 - duplicate records, 220
 - executing bulk, 125
 - execution plans, 228
 - exposed flag, 193
 - grouping to get percentages, 218
 - joining tables, 197
 - locking modes, 661
 - manipulating in VBA using ADO, 123

- modifying existing, 126
- naming conversions, 191-192
- nested, 204
- nesting groups, 221
- opening recordsets of parameterized queries, 124
- optimizing, 229, 758
- owners, 628
- parameter, 216
- performance, 227
- qryCheckBackEnd-Replication, 883
- query by form, 206-207, 213
- query design grid, 193
- Record Source properties, 189
- resolution, 228
- Run Permissions property, 628-629
- runtime, 746
- select, 193
- speed, 205, 237
- SQL Pass-Through queries, 760-761
- SQL Server, 788
- subqueries, 205
- summary, 201
 - functions, 202*
 - results, 203*
 - totaling rating values, 225*
- temporary Command objects, 215
- testing, 757
- Top Values property, 204
- troubleshooting, 217
- union, 194, 203, 226, 272
- user access, 190
- VBA, 205
- versus views, 761
- zstblQuery table, 190
- Queries object, 623**

query by form. See **QBF**
Query command (Insert menu), 760
query design grid
 Datasheet view, 199
 generating SQL statements, 194
 joining tables, 195-196
 Totals command (View menu), 201
Query Design view, 193
query designer, 216
Query menu commands, 760-762
query objects, 644-645
query processing, 752-753
query wizards, 514
QueryReqValue wrapper routine, 567
QueryValueEx wrapper routine, 568
Quit method, 90

R

Raise method (Err objects), 154-155
Rating field, 382
Read Data permission, 603
Read Design permission, 603
read-only attributes, 736
read-only properties, 62
read-only replicas, 702, 706
real data type, 801
record aggregation, 749
record Navigation control, 378
Record Navigation Section, 360
Record Source properties (queries), 189

record-returning stored procedures, 762
RecordCount property, 122
records
 adding to unbound forms, 671
 deleting duplicate, 220
 distinguishing between new and old, 223
 editing, 120, 359-360
 locating and loading with an unbound form, 675-676, 678
 locking, 653
 manipulating in Design Master, 706
 reordering, 232
 replication conflicts, 719
 saving on an unbound form, 680, 683, 685
 tracking multiple, 492
Recordset object
 ADODB, 111
 CancelUpdate method, 120
Recordset property (forms), 245
RecordsetLabel property, 362
recordsets
 ADO properties used with Jet, 117
 auto requery, 208
 bookmarks, 123
 Cachesize property, 120
 cloning, 123
 combining with union queries, 226
 creating, 116
 persistent, 120
 table-type, 118
 Do Until...Loop, 119
 GetString method, 118

 listings, 119-121
 locking, 118, 123
 navigating, 119
 opening, 117, 124
 operations supported, 122
 RecordCount property, 122
 records using DAO, 122

RecordSource property, 209

recovery (server databases), 747

recreating SQL databases with Server scripts, 775-776

References collection, 99

References command (Tools menu), 46

References dialog box, 46

referencing

 controls, 54, 57
 databases, 46
 forms, 47, 102
 default instances, 103
 multiple instances, 107
 methods, 50
 Missing Reference errors, 393
 NameSpace object, 411
 object model libraries, 113
 TypeLibs, 392

referential integrity conflict, 727

Refresh Interval setting, 656

Refresh method (Errors collection), 157

registering

 add-ins, 520
 DLLs, 550
 files, 549

Registry (Windows), 544

 Access wizard objects, 513
 adding entries, 563

- API calls, 555-556
 - listing API declarations and wrapper routines, 566*
 - modifying, 562*
 - registering back-end databases, 562*
 - sample application, 557*
 - working with code, 558*
- builders, 513
- commands, 550-551
 - DeleteSetting, 554*
 - limitations, 555*
 - SaveSetting, 551-553*
- creating entries, 559
- default settings, 551
- deleting a back end, 571
- deleting entries, 558
- format and contents, 546
- history of, 544
- keys, 546
 - clearing reference to, 565*
 - creating with passed arguments, 565*
 - individual user profiles, 551*
 - predefined, 547*
 - subkey lists, 568*
 - subkeys, 547*
- manipulating, 544
- read-ahead settings, 232
- Registry Editor, 548
- Regsvr32, 549
- restoring, 548
- tracking custom properties, 545
- values, 546-547
- wizards, 513
- Registry Editor, 548-549**
- Registry menu, Export**
- Registry File option, 549**
- Regsvr32, 549-550**
- regular identifiers, 799**
- REG_SZ constant, 564**
- relating tables, 196**
- relationship filters, 718**
- Relationship Wizard dialog box, 381**
- relationships, 792**
 - database performance, 814
 - foreign key constraints, 808
 - grouped DAPs, 381
 - rebuilding, 766
- Remove method, 82**
- removing**
 - groups, 612
 - items from collections, 82
 - passwords from user accounts, 613-614
 - users, 612, 638
- repeating code, 39-42**
- Replica object (JRO), 113**
- replica set topologies, 710**
 - singly connected list, 710
 - star and hub, 711-712
- replicable objects, 714**
- ReplicaIDs, 737**
- replicas**
 - anonymous, 708
 - application distribution, 713
 - backups, 738
 - Compact utility, 738
 - conflicts
 - Last-Update-Wins algorithm, 724*
 - list of, 726-727*
 - counter fields, 735
 - creating additional (listing), 703
 - direct/indirect synchronization, 734
 - global, 707
 - identifying conflicts, 724
 - last synchronization partners, 737
 - local, 707
 - MDE files, 739
 - partial, 715
 - creating in code, 717*
 - creating reports, 717*
 - listing, 717*
 - Partial Replica wizard, 716*
 - relationship filters, 718*
 - physical changes, 708
 - read-only, 702, 706
 - read-only attributes, 736
 - replica sets, 710
 - replicable objects, 706
 - replication identification
 - fixup, 737
 - scheduled and on-demand synchronizations, 734
 - security, 739
 - synchronization phases, 733
 - Synchronizer, 728-730
 - configuring options, 730*
 - scheduled synchronizations, 732*
 - synchronizing, 704
 - Design Masters, 706-707*
 - direct exchanges, 705*
 - over the Internet, 734*
 - upgrading sets, 739
 - visibility, 707
- Replicas collection (JRO), 113**
- replicated table editors, 875**
- replication, 692-694**
 - back-end/front-end applications, 719
 - columns, 699
 - Compact utility, 738
 - conflicts, 719
 - alternative conflict-resolution algorithms, 723*
 - Conflict Viewer, 720-722*

- dialog warnings*, 722
- identifying*, 724
- Last-Update-Wins algorithm*, 724
- list of*, 726-727
- converting databases to replicable formats, 699-700
 - backing up*, 700
 - creating additional replicas*, 701
 - input form (listing)*, 700
- counter fields, 735
- design goals, 693
- direct/indirect synchronization, 734
- distributing replicable applications, 713
 - partial replicas*, 715-718
 - replicable and non-replicable objects*, 714
- Jet, 694
 - Access Replication submenu*, 696-697
 - Briefcase*, 695-696
 - JRO*, 699
 - Replication Manager*, 698
- JRO library, 692
- MDE files, 739
- read-only attributes, 736
- read-only replicas, 702
- remote location users, 698
- replica set topologies, 710
 - singly connected list*, 710
 - star and hub*, 711-712
- replica sets, 710
- scheduled and on-demand synchronizations, 734
- security, 739
- semi-static data, 883
- SQL Server, 788
- synchronization phases, 733
- Synchronizer, 728-730
 - configuring options*, 730
 - scheduled synchronization*, 732
- synchronizing over the Internet, 734
- synchronizing replicas, 704
 - Design Masters*, 706-707
 - direct exchanges*, 705
 - physical changes*, 708
 - visibility*, 707
- tables from the back end to the front end, 874
 - listing*, 877, 880
 - replicated table editors*, 875
 - replication check routine*, 883, 886
 - updating front end tables*, 882
- replication check routine, 883, 886**
- replication environments, 629**
- replication identification fixup, 737**
- Replication Manager (Jet replication tool), 698**
 - additional features, 698
 - Configure Replication Manager Wizard, 729
 - configuring, 731
 - schedules and on-demand synchronizations, 734
 - Synchronizer, 728
- report properties (Upsizing Wizard), 771**
- report wizards, 514**
- reports**
 - adding with wizard-like interface, 345-347
 - alternating colors in a report section, 348
 - coding, 314
 - comparing to DAPs, 362-363
 - creating summary, detail and summary/detail reports from the same report, 314
 - DAPs. *See* DAPs
 - detail information, 319
 - dynamic groupings, 320-321
 - formatting dynamically, 347
 - conditional formatting*, 349-352
 - rptDynamicFormatting-Example report*, 347
 - graphics, 329
 - grouping
 - wizard-like interfaces, 335-347
- locking modes, 661
- MultiSelect list box, 328-330, 333-334
- partial replicas, 717
- performance, testing, 341
- queries, 217
- Record Source property, 189
- Section property, 318
- snaking report feature, 323
 - after reports*, 328
 - before reports*, 325
 - Columns Page settings*, 327
- Upsizing Report, 771
 - of users and groups, 625
- Reports object, 624**
- ReqCloseKey routine, 565**
- ReqCreateKeyEX() API function, 565**
- RequerySubform() function, 210**
- reserved words (SQL Server), 751**
- resolution algorithms, 719**

resolving replication conflicts, 719

- alternative conflict-resolution algorithms, 723
- Conflict Viewer, 720-722
- dialog warnings, 722

restoring

- Access, 748
- Registry, 548

Resume *LineLabel* statements, 153**Resume Next statements, 152****Resume statements, 150-151****Retention Period, 732****return values (arguments), 33****reusing forms, 251-259****Reverse() function, 41****reversing loops, 41****Rich Textbox control, 424, 445**

- code, 448
- loading files into control, 449
- properties, 446
- saving files from control, 450
- text alignment, 449

right joins, 195**roll-your-own locking scheme, 665****routines, 462**

- ap_CompactDatabase form, 868
- ap_LinkTables, 845-848
- DLLs, 463
- error handling, 171
- unbound forms, 667
- wrappers, 480

Row Source property, 271**row-level locking, 661****RowLevelTracking property, 699****RowSource property (combo boxes), 754****rptConditionalFormatting-Example report, 349****rptConditionalFormattingInCodeExample report, 352****rptDynamicFormatting-Example report, 347****rptDynamicGroupings-Example report, 321****rptMultiSelectCategory-Example report, 329****RSA RC4 encryption technology, 605****Run Permissions property, 628-629****Run With Owner's Permission (RWOP), 761****RunSQL method (DoCmd object), 215****runtime**

- adding images to ImageList control, 431
- error handling, 148
 - creating a simple handler, 149*
 - On Error command, 149*
 - On Error Goto 0 statement, 150*
 - On Error Goto statement, 149*
 - On Error Resume Next statement, 150*
 - Resume LineLabel statement, 153*
 - Resume Next statement, 152*
 - Resume statement, 150-151*
- morphing Access controls, 290
- queries, 746

Rushmore technology, 229**RWOP (Run With Owner's Permission), 761****S****Sams Publishing Web site, 633****Save As Data Access Page dialog box, 364****SaveSetting command (Registry), 550-553****saving**

- objects as HTML, 587
- records on an unbound form, 680, 683-685

Scale property (SQL Server), 797**scheduled synchronization, 733****s_ColLineage field, 709****scope**

- procedures, 32
- variables, 27-28, 107

Screen objects, accessing, 91**screen painting, 87****scripts (SQL), 776****Scrolling Text control, 378****searching**

- back ends, 849-852
- objects, 46

SearchSubform subform control, 255**Section properties, 318****secured applications, distributing, 629-630****security, 599**

- ? CurrentUser(), 602
- Access, 763
- Admin user accounts, 601

- applications, 598
 - code (developer's perspective)*, 599
 - sensitive data (client's perspective)*, 598
- Chap20.mdb file, 647-648
- code, 599
- databases, 598, 748
 - passwords*, 600-601
- linked tables
 - securing*, 627
- managing through code, 630
 - adding users to groups*, 637-638
 - databases*, 634-635, 645-646
 - Documents collections*, 631
 - groups*, 635-637
 - object owners*, 639-640
 - object permissions*, 640-641
 - permissions*, 631, 641-642
 - programming with DAO (Data Access Objects)*, 630-631
 - removing users from groups*, 638
 - Shift bypass key, disabling*, 646-647
 - users*, 632-633, 643-645
 - Users and Groups collections*, 631
 - users (whom you're logged on as)*, 643
- objects
 - creating with default accounts*, 626-627
 - permissions*, 603
- owner's permissions, 627-629
- passwords
 - for databases*, 600-601
 - protecting*, 600
- pitfalls, 626
- planning, 626
- programming, 630-631
- Programming Jet Security white papers, 626
- Projects, 794
- replicated databases, 739
- replication environments, 629
- resources, 626
- RSA RC4 encryption technology, 605
- secured applications, 629-630
- Security white papers, 626
- Security Wizard, 625
- sensitive data (client's perspective), 598
- share-level, 599-601
- SQL Server, 748, 763, 789
- tools, 625
- user access to queries, 190
- user interface
 - accounts*, 609-610
 - database permissions*, 616-617
 - databases*, 619-624
 - groups, adding users*, 611-613
 - modules*, 615
 - objects*, 617, 619
 - passwords*, 610-614
 - permissions*, 614-615, 623-624
 - PID (personal identifier)*, 608-610
 - SID (security identifier)*, 608
 - user accounts*, 609-614
 - WIDs (workgroup identifiers)*, 609
 - workgroup information files, creating*, 619-620
- user-level, 599-601
 - database encryption*, 605-607
 - groups*, 601-603
 - logon names*, 601
 - ownership*, 605
 - passwords*, 601
 - permissions*, 603-605
 - System.mdw file*, 607-608
 - usernames, finding*, 601
 - users*, 601, 603
- User-Level Security Wizard, 625
- views, 815
- workgroup information file, 625
- Security and Workgroup Administrator command (Tools menu)**, 607
- Security command (Tools menu)**, 600-601
- security identifier (SID)**, 608
- security models (creating Projects)**, 790
- Security white papers**, 626
- Security Wizard**, 598, 625
- Select Data Source dialog box**, 767
- select queries**, 188-190, 193
- Select query**, 792
- SELECT statements**, 206
- Select Workgroup Information File dialog box**, 620
- Select...Case statement**, 39
- Selected property (ListBox control)**, 293
- Selected() array**, 297
- SelectionChange event**, 244

- self joins, 196
- semi-static data, replicating, 883
- semi-static objects, 658
- sensitive data, 598
- server databases,
 - backups/recovery, 747
- Server scripts (SQL databases), 775-776
- server tables, linking, 767
- servers
 - Automation, 394
 - database, 744
 - linked, 755
 - linking tables to, 780
- Service Manager dialog box, 787
- sessions (Automation), 391
- Set Database Password dialog box, 600
- Set statements, 52, 91
- SetOption method, 89
- SetReqKeyValue wrapper routine, 568
- setting
 - database passwords, 634-635
 - flag files, 841-843
 - form properties at runtime, 255
 - permissions, 623-624
 - properties, 50
 - values, 48
- settings (system), 835
- SetValueEx wrapper routine, 568
- s_Generation field, 709
- s_GUID field, 709
- share-level security, 599-601
- shared databases, 652
- sharing data, 657, 748
- Shift bypass key, 646-647
- Shift+clicking keyboard shortcut, 615
- ShowCollectionFeatures() function, 83
- ShowXMLImportExport routine, 143
- SID (security identifier), 608
- simultaneous update conflict, 726
- singly connected list topology, 710
- sizing text boxes at runtime, 443
- Slider control, 424, 443-444
- s_Lineage field, 709
- smalldatetime data type, 801
- smallint data type, 801
- smallmoney, 801
- snaking reports, 323
 - after reports, 328
 - before reports, 325-327
 - functionality, 323
- snapshots versus dynasets, 754
- Some operator, 805
- source code, stripping out, 519
- SourceObject property, 261
- spaces, embedded (field/table names), 751
- speed (queries, 205), 237
- splash screens, 247
- splitting databases
 - advantages, 658
 - Database Splitter wizard, 660
 - drawbacks, 659
 - manually, 659
- Spreadsheet Office Web Component, 378
- spreadsheets
 - loading from Access, 403
 - XML, 144
- SQL
 - clauses, 749
 - criteria strings, 211
 - databases, 775-776
 - scripts, 776
- SQL Pass-Through queries, 760-761
- SQL Server
 - applications
 - advanced, 760
 - converting to, 756-757
 - cascading deletions, 809
 - column attributes, 797-798
 - column data types, 800
 - constraints, 803
 - check, 804-807
 - default, 807
 - foreign key, 808
 - primary key, 807
 - unique, 809-810
 - creating new databases, 791
 - data, loading, 776
 - data types, comparing with
 - Jet-based applications, 803
 - database diagrams, 788
 - detaching data files, 791
 - functions, user-defined, 757
 - identifiers, 799
 - Jet-linked table applications, 77
 - modifying tables on a linked database, 796
 - non-English data, 802
 - objects, 788
 - name rules, 800
 - naming conventions, 799
 - optimizing data access, 813
 - balancing index use, 814
 - relationships, 814

timestamp fields, 815
triggers and stored procedures, 814

permissions, 763
 queries, 758
 replication, 788
 reserved words, 751
 security, 748, 763, 789
 stored procedures, 762-763, 788, 818
 attributes of, 818
 creating, 818-819
 managing, 763
 parameters, 820
 returning values, 820
 testing parameters, 821
 storing data outside the host table, 802
 syntax elements, 822
 timestamp fields, 757
 Transact-SQL, 762
 triggers, 788
 updating tables, 812
 upgrading from MSDE to SQL Server 2000, 786
 views, 761-762, 788

SQL Server databases

creating, 776
 OLE DB, 781

SQL Specific, Data Definition (Query menu), 762

SQL Specific, Pass-Through command (Query menu), 760

SQL statements

compiling, 228
 executing against the current database, 215
 generating in the query design grid, 194
 moving to VBA, 195
 UNION, 272-275

SQL view, 194

SQL-linked applications, 750

standard modules, 27

StandardSearch form, 251

star and hub topology, 711-713

StandardSearch subform, 256

Start menu, 764

Startup properties, 792

startup system checks, 830-835

statement

statements

CREATE VIEW, 761
 Exit Do, 40
 If...Then, 39
 Select...Case, 39
 Set, 52
 SQL. *See* SQL statements

static information

(client/server), 756

static objects, 658

static variables, 28, 32

StatusBar control, 424, 450

properties, 451
 setting properties at runtime, 452

StatusBar Panels collection, 451

StDev() function, 202

Stored Procedure design window, 818

stored procedures, 74, 762-763, 818

Access and SQL Server syntax comparison, 822
 attributes of, 818
 creating, 818-819
 managing, 763
 parameters, 820

returning recordsets, 819

returning values, 820

SQL Server, 788

testing parameters, 821

triggers, 788

creating, 811

editing, 812

testing, 813

storing

add-ins, 521
 application data
 clients/servers, 742
 back-end database information, 563
 bookmarks, 123, 499
 data, 742
 data outside the host table (SQL Server), 802
 information
 in Access applications, 545
 about bookmarks, 495
 queries, 193
 values, 49, 865

Stream object (ADODB), 111

strings

appending, 36
 concatenating, 37
 maximum fixed lengths, 475

subform controls, 255

Subform/Subreport Wizard, 299

subforms

binding to Source Object
 properties during runtime, 261
 binding to tables or queries, 245
 RequerySubform() function, 210
 Subform/Subreport Wizard, 299

subkeys, 547

- lists for Registry keys, 568
- Registry Editor, 548

subqueries, 205**subroutines, 31, 47. See****also methods**

- AfterUpdate, 759
- in code modules, 27
- creating, 32
- Property Get, 59
- Property Let, 59

Sum() function, 202**summary queries, 201**

- functions, 202
- results, 203
- totaling rating values, 225

summary reports, 314**support objects**

- creating, 64-65

Supports method, 122**switchboards, 642****synchronization (Internet), 728****Synchronize method, 704****Synchronizer**

- configuring options, 730
- direct synchronization, 733
- scheduled synchronizations, 732

synchronizing replicas, 704

- conflicts, 720
- Design Masters, 706-707
- direct exchanges, 705
- direct/indirect synchronization, 734
- Indirect and Internet synchronization modes, 704
- last synchronization partners, 737
- local/global, 707
- over the Internet, 734
- phases, 733
- physical changes, 708

replica sets, 710

- scheduled and on-demand synchronizations, 734
- star and hub replica topology, 712
- Synchronizer, 728-732
- visibility, 707

synchronous errors, 42-43**syntax. See also listings**

- API calls, 463
- comparing Access and SQL Server, 822
- Dir() function, 842-843

SysCmd function, 413**SysCmd() function, 439-440, 644****system checks (startup), 830-835****system properties, 834****system settings, 835****System.mdw file, 607-608****T****Tab control, 282-283**

- Bookmark Tracking Wizard, 526
- creating and editing, 284
- default values, 284
- moving pages, 286, 288
- printing the Caption property for each page, 287
- versus the ActiveX TabStrip control, 426

tabbed forms, 282-283

- pages, 286
- performance, 260-261
- Tab control
 - adding controls, 286
 - creating and editing, 284
 - default values, 284
 - moving pages, 286-288
 - programming, 287

Table Analyzer Wizard, 238**Table Design view, 579****table design window, 798****table links, 843-845****Table objects**

- ADOX, 111
- creating, 644-645

table wizards, 514**table-level validation conflict, 727****table-type recordsets, 118****tables**

- adding hyperlinks, 580
- ADO, 127
 - creating new, 128
 - modifying existing, 130
 - OpenSchema method, 128
- ADOX, 848
- binding subforms to, 245
- check constraints, 806
- ColumnLevelTracking property, 699
- controls, 268
- creating a Project from, 406
- elements, toggling to
 - include/disclude, 337
- exporting, 765-766, 862
 - ap_SystemUtilities form, 863
 - DLookup() function, 867
 - ExportTypeDefault field (ztblDaatabaseProperties table), 865
 - file types, 864
 - listing, 865
- insert buffer, 812
- joining, 195-196
- linked, 568, 719
 - linking/unlinking, 845-848
 - securing, 627
 - to servers, 780

- local, 756
- lookup property, 266-267
- modifying, 796
- naming, 751
- Properties dialog box, 806
- relationships, 233
- replicating from back end to front end, 874
 - listing*, 877, 880
 - replicated table editors*, 875
 - replication check routine*, 883, 886
 - updating front end tables*, 882
- Rushmore technology, 229
- saving as HTML, 587
- server tables, linking, 767
- shared data, 657
- SQL Server, 788
- storing database values, 865
- triggers, 812
- upsizing, 750
 - case sensitivity*, 752
 - field/table names*, 751
 - query processing*, 752-753
 - reserved words*, 751
- views, 816
- Tables collection**
 - ADOX, 111
 - compared to TableDefs collection (DAO), 127
- Tables object, 623**
- tblLinkedTables, 557**
- TabStrip control, 424-425**
 - images, 430
 - listing, 427
 - versus Access Tab control, 426
 - with ImageList control, 425
- Tag property, 252-254, 257**
- task items (Outlook), 411**
- tblRatings, 871**
- tblReplicatedTables, 882**
- tblSharedTables table, 863**
- temporary objects, 658**
- testing**
 - ADP, 72
 - back ends, 868
 - databases, corrupt back end, 852-857
 - links, error handling, 844-845
 - queries, 757
 - reports, 341
 - table links at startup, 843-845
 - triggers, 813
- text boxes, sizing at run-time, 443**
- text data type, 802**
- themes (DAPs), 368, 377**
- threads (Jet), 232**
- timer events, 841**
- Timer Interval property, 837**
- timestamp data type, 802**
- timestamp fields, 757, 815**
- tinyint data type, 802**
- tlb file extension, 46**
- tooggling views, 26**
- ToolBar control, 424, 453**
 - button styles, 455
 - creating, 454
- Toolbox command (View menu), 24**
- Tools menu**
 - commands
 - Analyze, Documenter*, 766
 - Database Scripting*, 776
 - Database Utilities*,
 - Repair Database*, 831
 - Database Utilities*,
 - Upsizing Wizard*, 768
- References*, 46
- Security*, 600-601
- Security and Workgroup Administrator*, 607
- options, 793
- Top Values property, 204**
- topologies (replica sets), 710**
- tracking**
 - Bookmark Tracker
 - basic objects*, 494
 - class modules*, 494-497, 500
 - features*, 492
 - custom properties, 545
 - errors
 - custom error logs*, 163-166
 - logging to back end first*, 168
 - updating back end with errors found*, 168
 - properties, 834
 - records, 492
- traffic (network), 748-749**
- Transact-SQL (T-SQL), 762, 818**
- transaction logs, 769**
- transactions**
 - processing, 11
 - rolling back, 178
- trapping**
 - errors, 151
 - synchronous, 43
- TreeView control, 424, 456**
 - Nodes collection, 459
 - populating, 458
 - Style property, 458
- triggers (stored procedure), 762, 788**
 - creating, 811
 - Design view, 812
 - editing, 812

- listings, 812
- optimizing SQL Server
 - Project performance, 814
- testing, 813
- troubleshooting**
 - code, 42-43
 - controls, 57
 - error handling issues, 176
 - environment changes*, 176
 - nesting error handlers*, 181
 - On Error event*, 179
 - rolling back transactions*, 178
 - forms, debugging, 23
 - objects, naming, 46
 - queries, 217
 - timer events, 841
 - transaction logs, 769
- T-SQL (Transact-SQL), 818, 823**
- twips, 51**
- txtBMTName text box, 531**
- type libraries, 113-114**
- TypeLibs, 392-393**
- TypeName() function, 54**
- TypeOf operators, 53-54**
- types (objects), 53-54**

U

- unbound forms, 269**
 - example, 668-669
 - getting the Key field from the form's Tag property, 671
 - getting the record source, 670
 - records
 - adding*, 671
 - creating new*, 674
 - locating and loading with*, 675-678
 - saving*, 680, 683-685
 - routines, 667
- unbound forms locking scheme, 666**
- unbound hyperlink controls, 576**
- unbound hyperlinks, 370**
- UNC (Unified Naming Convention), 737**
- UNC path, 577**
- Underline Hyperlinks option, 587**
- underscore symbol (_), 33**
- Unicode-enabled, 802**
- unidirectional exchanges, 705**
- Unified Naming Convention (UNC), 737**
- union queries, 194, 203, 272**
 - combining two crosstabs, 226
 - combo boxes, 273
 - creating, 226
- UNION SQL statements, requering all in a sub-form, 275**
- unique constraints, 809-810**
- unique key conflict, 727**
- uniqueidentifier data type, 802**
- unlinking tables, 845-848**
- unsecuring databases, 624**
- Until keyword versus While keyword, 41**
- Update Data permission, 603**
- update-delete conflict, 726**
- update queries, 201, 792**
- UpdateStatusBarControl routine, 456**
- updating**
 - back ends with errors found, 168
 - front end after replicating back end tables, 882
 - HTML files, 592
 - records, 120, 720
 - SQL Server tables, 812
 - tables, 884
 - versions (front end), 857-859
- upsized applications, 770**
- upsizing**
 - databases (Access), 73, 764
 - tables, 750
 - case sensitivity*, 752
 - field/table names*, 751
 - query processing*, 752-753
 - reserved words*, 751
- Upsizing Report, 771**
- Upsizing Wizard, 750, 768-771**
 - form/report properties, 771
 - starting, 768
- URLs (uniform resource locators), 575**
- UseBeep properties, 68**
- user accounts**
 - passwords
 - adding*, 613
 - removing*, 613-614
 - Users groups, 602
- User and Group Accounts dialog box, 601**
- user interfaces**
 - accounts, re-creating, 609-610
 - databases
 - encrypting*, 619
 - manually securing*, 620-624

setting permissions,
616-617

unsecuring, 624

groups

adding users, 612-613

creating, 611

removing, 612

modules, securing in VBE
(Visual Basic Editor), 615

objects, changing owners,
617-619

passwords, 610

adding to user accounts,
613

removing from user
accounts, 613-614

permissions, 614-615,
623-624

PID (personal identifier),
608-610

SID (security identifier), 608

user accounts

adding passwords, 613

removing passwords,
613-614

user name criteria, 609

users

adding to groups,
612-613

creating, 610-611

removing, 612

WIDs (workgroup identi-
fiers), 609

workgroup information files,
619-620

User object (ADOX), 112

user-definable properties, 254

user-defined collections, 55

user-defined errors, 155, 161-162

user-defined functions, 763, 788. See also in-line functions

client/server, 755

SQL Server, 757

user-level security, 599-601

database encryption,
605-607

groups, 601-603

logon names, 601

ownership, 605

passwords, 601

permissions, 603-605

System.mdw file, 607-608

usernames, 601

users, 601-603

User-Level Security Wizard, 625

user/group security, 794

usernames, 601

users

? CurrentUser(), 602

adding to groups, 612-613

Admin user accounts, 601

code, 637-638

creating, 610-611, 632-633

creating or deleting, 632

database creation, 643-644

deleting, 633-634

logging out

applications, 836-843

checking necessity,
838-839

countdowns, 840-841

warning, 839-840

multiple, 747

query object creation,
644-645

removing, 612, 638

reports of, 625

security, 601, 603

table object creation,
644-645

Users and Groups collection

DAO (Data Access Objects)

hierarchy, 630

security, 631

Users collection (ADOX), 112

Users groups, 602

USysRegInfo table, 519-520

utility forms

(*ap_SystemUtilities*), 862

V

validation

SQL Server constraints, 803

triggers, 811

values

errors, 171-172

properties

accessing, 49

defaults, 49-50

setting, 48

storing, 49

Registry, 546-548

Var() function, 202

varbinary data type, 802

varchar data type, 802

variables

catalog variables, 847-848

copying, 52

data types, 29-30

declaring, 27-31

module level, 27

naming (Leszynski naming
conventions), 29

objects, 52

passing, 37

private, 28

procedure level, 27

pstrBackEndPath, 836

public, 47

scope, 27-28, 107

static, 28, 32

Variant data types, 36**VBA (Visual Basic for****Applications), 10, 27-29**

- arguments, 33
- Automation functionality, 390
- Bad Code button, 216
- coding, 142
- commands (Bookmark Tracking Wizard), 522
- conflict-resolution algorithms, 723
- control-of-flow statements, 205
- converting macros to code, 15-18
- creating Access tables from Project, 420
- creating and filling ListView controls, 437
- decision-making code, 39
- declaring object variables, 394
- features (object-oriented), 44
- functions (compared to T-SQL), 823-824
- Jet objects and references, 551
- ListBox controls, manipulating in multiselect mode, 294
- locking modes, 665
- manipulating controls, 304, 307
 - listings, 308*
 - Option Group menu form, 305*
- morphing controls at runtime, 290
- New keyword, 81
- Object Browser, 44, 393
- objects, 44
 - methods, 44*
 - programming with, 47*

- properties, 44*

- viewing, 45*

- program flow, 39

- queries, 205

- manipulating with ADO, 123-124*

- parameter, 216*

- referencing libraries, 114

- Registry commands, 550

- DeleteSetting, 554*

- limitations, 555*

- SaveSetting, 551-553*

- SELECT statements, 206

- setting cursor movement, 303

- SQL statements, 194

- strings, concatenating, 37

- tasks requiring, 11

- temporary Command

- objects, 215

- unbound forms, 666

- variables, 29

- wrapper routines, 462, 555

VBE (Visual Basic Editor), 604

- Access Project security, 794

- modules, 604, 615

versions (front end), 857-859**VideoApp(ADO).mdb, 831-834****VideoPaa(ADO).mdb database, 862****View menu commands**

- Code, 59

- Object, 26

- Object Browser, 45, 842

- Properties, 24, 629

- Toolbox, 24

View object (ADOX), 112**ViewChange event, 244****viewing**

- hidden objects, 519

- Object Browser, 45-46

- objects, 45

- property sheets, 48

- Run Permissions property, 629

- subroutines in code modules, 27

views. See also queries

- creating, 816

- Datasheet. *See* Datasheet view

- Distinct keyword, 816

- Form. *See* Form view

- Project uses, 815

- SQL Server, 761-762, 788

- toggleing, 26

- versus queries, 761

Views collection (ADOX), 112**visibility, 27-28, 707****Visibility argument, 701****Visual Basic Editor. See VBE****Visual Basic for****Applications. See VBA****W****W3C (World Wide Web Consortium), 134****WANs, 653, 730****Web sites**

- Application Plus, 580

- Chap20.mdb database, 633

- DAPs, 372

- SAMS Publishing, 115, 633

Where() function, 202**While keyword versus Until keyword, 41****While...Wend loop, 41****white papers, 626****WIDs (workgroup identifiers), 609**

wildcard characters (SQL Server check constraint), 805

Win32api.txt file, finding API calls, 468

Windows APIs, 92

Windows Registry. *See* Registry

Windows Registry Editor, 608

With statements, 50

wizard-like interfaces, 335, 344

adding reports, 345-347

core tables, 336-342

wizards, 512

Analyzer, 238

Bookmark Tracker Wizard, 517

builders, 512

Combo Box Wizard, 268, 375

Configure Replication Manager, 729

DAPs, 364-366

Database Splitter, 660

installing with Add-In Manager, 520

Partial Replicas, 715

Registry entries of Access wizard objects, 513

Security Wizard, 598, 625

specifying Registry settings, 519

Subform/Subreport Wizard, 299

Upsizing, 750, 768-771
form/report properties,
771

starting, 768

User-Level Security Wizard, 625

WizExElements table, 336-337, 341

adding reports, 346-347
maintaining element order, 338

WizExReports table, 336

adding entries, 347
command buttons, 339
MoveCurrentField() function, 342
rstReportToUse variable, 339
selecting and deselecting all elements, 345

WNetAddConnectionA declaration, 476-479

WNetCancelConnectionA declaration, 476-479

WNetConnectionDialog declaration, 479-481

WNetDisconnectDialog declaration, 479-481

Word (Microsoft), running from Access with Automation, 398-400

workgroup identifiers (WIDs), 609

workgroup information file, 607-608

creating, 619-620
joining, 620
key (Windows Registry Editor), 608
users and groups, printing reports of, 625

World Wide Web

Consortium (W3C), 134

wrapper routines, 462, 480

list of, 567

Registry, 566

VBA, 555

write conflicts, 664

writing

code, 23
event handlers, 25
methods, 63
properties
custom, 59-62
object-valued, 63

X – Y – Z

XML (Extensible Markup Language), 75, 77, 134, 574, 592

coding, 142

documents

data display, 138

file types, 135

listing, 136

exporting data to Access, 142

history of, 134

importing from Access, 135, 139-141

Office application support, 144

XML Spreadsheet Schema (XML SS), 144

XSL, 139

zsfrmSQLVBA form, 195

zsqupdQueryListNew query, 193

zstblQuery table, 190

ztblDatabaseProperties table, 865