

# ACCESS® 2021 / MICROSOFT 365 PROGRAMMING BY EXAMPLE



JULITTA KOROL

# **ACCESS® 2021/MICROSOFT 365**

## **PROGRAMMING BY EXAMPLE**

## **LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY**

By purchasing or using this book (the “Work”), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

MERCURY LEARNING AND INFORMATION (“MLI” or “the Publisher”) and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs (“the software”), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold “as is” without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book, and only at the discretion of the Publisher. The use of “implied warranty” and certain “exclusions” vary from state to state, and might not apply to the purchaser of this product.

*The companion files are also available for downloading by writing to the publisher at [info@merclearning.com](mailto:info@merclearning.com).*

# **ACCESS® 2021/MICROSOFT 365 PROGRAMMING BY EXAMPLE**

**Julitta Korol**



**MERCURY LEARNING AND INFORMATION**  
Dulles, Virginia  
Boston, Massachusetts  
New Delhi

Copyright ©2022 by MERCURY LEARNING AND INFORMATION LLC. All rights reserved.

*This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.*

Publisher: David Pallai  
MERCURY LEARNING AND INFORMATION  
22841 Quicksilver Drive  
Dulles, VA 20166  
[info@merclearning.com](mailto:info@merclearning.com)  
[www.merclearning.com](http://www.merclearning.com)  
(800) 232-0223

Julitta Korol. *Access 2021/Microsoft 365 Programming by Example.*  
ISBN: 978-1-68392-841-6

This book is printed on acid-free paper in the United States of America.

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2022940572

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc.  
For additional information, please contact the Customer Service Dept. at 800-232-0223(toll free).

All of our titles are available in digital format at [academiccourseware.com](http://academiccourseware.com) and other digital vendors.  
*Companion files for this title are also available by contacting [info@merclearning.com](mailto:info@merclearning.com).*

The sole obligation of MERCURY LEARNING AND INFORMATION to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

*To a new generation of Microsoft Access programmers*



# CONTENTS

<i>Acknowledgments</i> .....	<i>xxvii</i>
<i>Introduction</i> .....	<i>xxix</i>

---

## PART I ACCESS VBA PRIMER ..... 1

<b>Chapter 1 Getting Started with Access VBA.....</b>	<b>3</b>
Understanding VBA Modules and Procedure Types.....	3
Writing Procedures in a Standard Module.....	7
Executing Your Procedures .....	10
Understanding Class Modules .....	13
Events, Event Properties, and Event Procedures .....	15
Why Use Events?.....	17
Walking Through an Event Procedure .....	17
Compiling Your Procedures.....	22
Placing a Database in a Trusted Location.....	23
Summary .....	26
<b>Chapter 2 Getting to Know Visual Basic Editor (VBE).....</b>	<b>27</b>
Understanding the Project Explorer Window .....	28
Understanding the Properties Window.....	30
Understanding the Code Window.....	32
Other Windows in the VBE.....	34
Assigning a Name to the VBA Project .....	35
Renaming a Module.....	36

Syntax and Programming Assistance.....	36
List Properties/Methods .....	37
Parameter Info .....	38
List Constants.....	39
Quick Info.....	39
Complete Word.....	40
Indent/Outdent .....	41
Comment Block/Uncomment Block .....	42
Using the Object Browser .....	42
Using the VBA Object Library .....	45
Using the Immediate Window .....	46
Summary .....	49
<b>Chapter 3 Access VBA Fundamentals.....</b>	<b>51</b>
Introduction to Data Types .....	51
Understanding and Using Variables .....	54
Declaring Variables.....	55
Specifying the Data Type of a Variable .....	58
Using Type Declaration Characters.....	60
Assigning Values to Variables .....	61
Forcing Declaration of Variables .....	64
Understanding the Scope of Variables .....	66
Procedure-Level (Local) Variables .....	66
Module-Level Variables .....	67
Project-Level Variables.....	69
Understanding the Lifetime of Variables.....	71
Using Temporary Variables .....	71
Creating a Temporary Variable with a TempVars Collection Object.....	72
Retrieving Names and Values of TempVar Objects.....	72
Using Temporary Global Variables in Expressions .....	73
Removing a Temporary Variable from a TempVars Collection Object.....	74
Using Static Variables.....	74
Using Object Variables.....	76
Disposing of Object Variables .....	79

Finding a Variable Definition .....	79
Determining the Data Type of a Variable.....	79
Using Constants in VBA Procedures .....	81
Intrinsic Constants .....	82
Summary .....	83
<b>Chapter 4 Access VBA Built-In and Custom Functions.....</b>	<b>85</b>
Writing Function Procedures.....	85
Running Function Procedures .....	86
Data Types and Functions .....	88
Passing Arguments By Reference and By Value .....	90
Using Optional Arguments.....	92
Using the IsMissing Function .....	93
VBA Built-In Functions for User Interaction .....	94
Using the MsgBox Function.....	94
Returning Values from the MsgBox Function .....	103
Using the InputBox Function.....	104
Converting Data Types.....	107
Summary .....	109
<b>Chapter 5 Adding Decisions to Your Access VBA Programs.....</b>	<b>111</b>
Relational and Logical Operators.....	111
If...Then Statement .....	112
Multiline If...Then Statement .....	114
Decisions Based on More than One Condition .....	116
If...Then...Else Statement .....	118
If...Then...ElseIf Statement .....	121
Nested If...Then Statements.....	122
Select Case Statement .....	125
Using Is with the Case Clause .....	128
Specifying a Range of Values in a Case Clause .....	128
Specifying Multiple Expressions in a Case Clause .....	130
Summary .....	131
<b>Chapter 6 Adding Repeating Actions to Your Access VBA Programs .....</b>	<b>133</b>
Using the Do...While Statement .....	134

Another Approach to the Do...While Statement.....	136
Using the Do...Until Statement.....	137
Another Approach to the Do...Until Statement.....	138
Using the For...Next Statement.....	139
Using the For Each...Next Statement .....	142
Exiting Loops Early .....	143
Nested Loops.....	144
Summary .....	146

## Chapter 7 Keeping Track of Multiple Values Using Arrays .....147

Understanding Arrays .....	148
Declaring Arrays.....	150
Array Upper and Lower Bounds .....	151
Initializing and Filling an Array .....	152
Filling an Array Using Individual Assignment Statements.....	152
Filling an Array Using the Array Function .....	152
Filling an Array Using the For...Next Loop .....	153
Using a One-Dimensional Array .....	154
Arrays and Looping Statements .....	156
Using a Two-Dimensional Array.....	160
Static and Dynamic Arrays .....	161
Array Functions.....	163
The Array Function.....	164
The IsArray Function.....	164
The Erase Function .....	165
The LBound and UBound Functions .....	169
Errors in Arrays .....	170
Parameter Arrays.....	173
Passing Arrays to Function Procedures .....	174
Sorting an Array .....	175
Summary .....	177

## Chapter 8 Keeping Track of Multiple Values Using Collections .....179

Creating Your Own Collection .....	180
Adding Items to Your Collection.....	180

Determine the Number of Items in Your Collection .....	181
Accessing Items in a Collection .....	182
Removing Items from a Collection .....	182
Updating Items in a Collection.....	183
Returning a Collection from a Function .....	186
Collections vs. Arrays .....	188
Watching the Execution of Your VBA Procedures.....	189
Summary .....	196
<b>Chapter 9 Getting to Know Built-In Tools for Testing and Debugging .....</b>	<b>197</b>
Syntax, Runtime, and Logic Errors.....	197
Stopping a Procedure.....	199
Using Breakpoints .....	200
Removing Breakpoints.....	205
Using the Immediate Window in Break Mode .....	205
Using the Stop Statement.....	207
Using the Assert Statement.....	208
Using the Add Watch Window.....	209
Removing Watch Expressions.....	213
Using Quick Watch.....	213
Using the Locals Window .....	214
Using the Call Stack Dialog Box.....	215
Stepping Through VBA Procedures.....	216
Stepping Over a Procedure.....	217
Stepping Out of a Procedure .....	219
Running a Procedure to Cursor.....	219
Setting the Next Statement .....	219
Showing the Next Statement .....	219
Navigating with Bookmarks .....	220
Stopping and Resetting VBA Procedures .....	221
Trapping Errors.....	221
Using the Err Object.....	222
Procedure Testing .....	226
Setting Error-Trapping Options .....	228
Summary .....	229

---

**PART II ACCESS VBA PROGRAMMING WITH DAO AND ADO ..... 231****Chapter 10 Data Access Technologies in Microsoft Access..... 233**

Understanding Database Engines: Jet/ACE .....	233
Understanding Access Versions and File Formats.....	235
Understanding Library References.....	238
Overview of Object Libraries in Microsoft Access .....	241
The Visual Basic for Applications Object Library (VBA) .....	241
The Microsoft Access 16.0 Object Library.....	241
OLE Automation.....	241
The Microsoft Office 16.0 Access Database Engine Object Library....	241
The Microsoft DAO 3.6 Object Library.....	241
The Microsoft ActiveX Data Objects 6.1 Library (ADO) .....	243
Creating a Reference to the ADO Library .....	245
Understanding Connection Strings.....	246
Using ODBC Connection Strings.....	248
Creating and Using ODBC DSN Connections .....	248
Creating and Using DSN-Less ODBC Connections.....	254
Using OLE DB Connection Strings .....	255
Connection String via a Data Link File .....	256
Summary .....	260

**Chapter 11 Creating and Manipulating Databases with DAO ..... 261**

Understanding the DBEngine and Workspace Objects .....	261
The DAO Errors Collection.....	263
Creating a Database with DAO .....	265
Copying a Database.....	266
Opening Microsoft Access Databases .....	267
Opening a Microsoft Jet Database in Read/Write Mode.....	268
Opening a Microsoft Access Database in Read-Only Mode .....	270
Opening a Microsoft Jet Database Secured with a Password .....	270
Creating and Accessing	
Database Tables and Fields .....	274
Creating a Microsoft Access Table and	
Setting Field Properties .....	275

Creating Calculated Fields .....	282
Creating Multivalue Lookup Fields .....	285
Creating Attachment Fields .....	288
Creating Append Only Memo Fields .....	290
Creating Rich Text Memo Fields .....	296
Removing a Field from a Table .....	299
Retrieving Table Properties .....	300
Linking a dBASE Table.....	301
Creating Indexes.....	301
Adding a Multiple-Field Index to a Table .....	303
Finding and Reading Records .....	305
Introduction to DAO Recordsets.....	305
Opening Various Types of Recordsets .....	307
Opening a Snapshot and Counting Records.....	309
Retrieving the Contents of a Specific Field in a Table .....	311
Moving Between Records in a Table .....	312
Finding Records in a Table-Type Recordset .....	313
Finding Records in Dynasets or Snapshots.....	315
Finding the nth Record in a Snapshot .....	317
Working with Records.....	318
Adding a New Record .....	319
Adding Attachments .....	320
Adding Values to Multivalue Lookup Fields.....	323
Modifying a Record.....	326
Deleting a Record .....	329
Deleting Attachments .....	331
Copying Records to an Excel Worksheet.....	332
Filtering Records Using the SQL WHERE Clause .....	337
Filtering Records Using the Filter Property .....	338
Creating and Running Queries .....	339
Creating a Select Query Manually.....	340
Creating a Select Query with DAO .....	344
Creating and Running a Parameter Query .....	346
Creating and Running a Make-Table Query .....	348
Creating and Running an Update Query .....	350
Running an Append Query.....	352
Running a Delete Query .....	353
Creating and Running a Pass-Through Query .....	356

Performing Other Operations with Queries .....	358
Retrieving Query Properties with DAO .....	358
Listing All Queries in a Database with DAO .....	359
Deleting a Query from a Database .....	360
Determining If a Query Is Updatable .....	360
Transaction Processing .....	363
Creating a Transaction .....	364
Summary .....	368
<b>Chapter 12 Creating and Manipulating Databases with ADO ....</b>	<b>369</b>
Creating an Access Database with ADO .....	369
Copying a Database .....	371
Copying a Database with FileSystemObject.....	371
Database Errors .....	372
Opening a Microsoft Jet Database in Read/Write Mode.....	375
Connecting to the Current Access Database .....	378
Opening Other Databases, Spreadsheets, and Text Files from Access ....	380
Connecting to an SQL Server Database .....	380
Opening a Microsoft Excel Workbook .....	380
Opening a Text File .....	383
Creating a Microsoft Access Table and Setting Field Properties.....	385
Copying a Table.....	388
Deleting a Database Table.....	389
Adding New Fields to an Existing Table .....	391
Removing a Field from a Table .....	392
Retrieving Table Properties .....	393
Retrieving Field Properties .....	394
Linking a Microsoft Access Table .....	395
Linking a Microsoft Excel Worksheet.....	397
Listing Database Tables.....	399
Listing Tables and Fields .....	400
Listing Data Types.....	403
Changing the AutoNumber .....	404
Creating a Primary Key Index.....	406
Creating Indexes Using ADO.....	406
Creating a Single-Field Index .....	408
Listing Indexes in a Table.....	410

Deleting Table Indexes .....	411
Creating Table Relationships .....	412
Introduction to ADO Recordsets.....	414
Cursor Types .....	416
Lock Types.....	417
Cursor Location .....	418
The Options Parameter.....	419
Opening a Recordset.....	423
Opening a Recordset Based on a Table or Query.....	424
Opening a Recordset Based on an SQL Statement.....	429
Opening a Recordset Based on Criteria.....	430
Opening a Recordset Directly with ADO.....	431
Moving Around in a Recordset.....	432
Finding the Record Position.....	433
Reading Data from a Field.....	434
Returning a Recordset as a String.....	435
Finding Records Using the Find Method.....	438
Finding Records Using the Seek Method .....	439
Finding a Record Based on Multiple Conditions .....	441
Using Bookmarks .....	442
Using Bookmarks to Filter a Recordset .....	445
Using the GetRows Method to Fill the Recordset.....	446
Working with Records in ADO .....	447
Adding a New Record .....	448
Modifying a Record.....	449
Editing Multiple Records.....	451
Deleting a Record .....	452
Copying Records to a Word Document .....	453
Copying Records to a Text File .....	456
Filtering Records.....	458
Sorting Records.....	460
Creating and Running Queries with ADO .....	462
Creating a Select Query with ADO.....	462
Executing an Existing Select Query with ADO.....	465
Modifying an Existing Query .....	468
Creating and Running a Parameter Query .....	470
Executing an Update Query .....	473
Creating and Executing a Pass-Through Query .....	476

Listing Queries in a Database .....	480
Deleting a Query.....	480
Using Advanced ADO Features .....	482
Fabricating a Recordset.....	482
Disconnected Recordsets.....	486
Saving a Recordset to Disk .....	489
Part 1: Saving a Recordset to Disk .....	493
Part 2: Creating an Unbound Access Form to View and Modify Data.....	493
Part 3: Writing Procedures to Control the Form and Its Data.....	495
Part 4: Viewing and Editing Data Offline.....	501
Part 5: Connecting to a Database to Update the Original Data .....	502
Cloning a Recordset .....	506
Introduction to Data Shaping.....	512
Writing a Simple SHAPE Statement.....	513
Working with Data Shaping .....	515
Writing a Complex SHAPE Statement.....	520
Shaped Recordsets with Multiple Children.....	520
Shaped Recordsets with Grandchildren .....	524
Part 1: Creating a Form with a TreeView Control.....	524
Part 2: Writing an Event Procedure for the Form Load Event.....	526
Transaction Processing .....	535
Creating a Transaction .....	536
Examining the References Collection.....	538
Summary .....	540
<b>PART III ACCESS STRUCTURED QUERY LANGUAGE (SQL) .....</b>	<b>541</b>
<b>Chapter 13 Creating, Modifying, and Deleting Tables and Fields .....</b>	<b>543</b>
Introduction to Access SQL.....	544
Creating Tables.....	546
Deleting Tables .....	551
Modifying Tables with DDL.....	553
Adding New Fields to a Table .....	553
Changing the Data Type of a Table Column.....	554
Changing the Size of a Text Column.....	555

Deleting a Column from a Table .....	556
Adding a Primary Key to a Table.....	557
Adding a Multiple-Field Index to a Table.....	558
Deleting an Indexed Column.....	559
Deleting an Index .....	561
Setting a Default Value for a Table Column.....	562
Changing the Seed and Increment Values of AutoNumber Columns.....	563
Summary .....	565
<b>Chapter 14 Enforcing Data Integrity and Relationships between Tables .....</b>	<b>567</b>
Using CHECK Constraints.....	568
Establishing Relationships between Tables .....	574
Using the Data Definition Query Window .....	577
Summary .....	580
<b>Chapter 15 Defining Indexes and Primary Keys.....</b>	<b>581</b>
Creating Tables with Indexes .....	582
Adding an Index to an Existing Table.....	583
Creating a Table with a Primary Key.....	585
Creating Indexes with Restrictions.....	587
Deleting Indexes .....	591
Summary .....	592
<b>Chapter 16 Views and Stored Procedures.....</b>	<b>593</b>
Creating a View .....	593
Enumerating Views.....	597
Deleting a View.....	598
Creating a Stored Procedure.....	599
Creating a Parameterized Stored Procedure .....	600
Examining the Contents of a Stored Procedure.....	603
Executing a Parameterized Stored Procedure .....	604
Deleting a Stored Procedure .....	606
Changing Database Records with Stored Procedures .....	607
Summary .....	609

---

**PART IV IMPLEMENTING DATABASE SECURITY ..... 611****Chapter 17 Implementing Database Security with DDL ..... 613**

Two Types of Database Security .....	613
Setting the Database Password.....	614
Removing the Database Password .....	615
Creating a User Account.....	616
Changing a User Password .....	618
Creating a Group Account.....	619
Adding Users to Groups.....	620
Removing a User from a Group .....	621
Deleting a User Account .....	622
Granting Permissions for an Object.....	623
Revoking Security Permissions .....	625
Deleting a Group Account .....	626
Summary .....	627

**Chapter 18 Implementing User-Level and Share-Level Security ..... 629**

Share-Level Security.....	630
User-Level Security .....	630
Understanding Workgroup Information Files .....	631
Creating and Joining Workgroup Information Files.....	633
Opening a Secured MDB Database.....	640
Creating and Managing Group and User Accounts.....	643
Deleting User and Group Accounts .....	647
Listing User and Group Accounts .....	649
Listing Users in Groups .....	650
Setting and Retrieving User and Group Permissions.....	652
Determining the Object Owner.....	652
Setting User Permissions for an Object .....	655
Setting User Permissions for a Database .....	658
Setting User Permissions for Containers.....	660
Checking Permissions for Objects.....	663
Setting a Database Password Using the DBEngine.CompactDatabase Method .....	665

Setting a Database Password Using the NewPassword Method.....	667
Changing a User Password .....	669
Summary .....	671
<b>PART V VBA PROGRAMMING IN ACCESS FORMS AND REPORTS .....</b>	<b>673</b>
<b>Chapter 19 Enhancing Access Forms.....</b>	<b>675</b>
Creating Access Forms .....	676
Grouping Controls Using Layouts.....	680
Rich Text Support in Forms .....	681
Using Built-In Formatting Tools .....	682
Using Images in Access Forms.....	682
Using the Attachments Control .....	684
Summary .....	689
<b>Chapter 20 Using Form Events .....</b>	<b>691</b>
Data Events .....	692
Current.....	693
BeforeInsert .....	695
AfterInsert .....	696
BeforeUpdate.....	696
AfterUpdate.....	697
Dirty .....	700
OnUndo .....	700
Delete.....	701
BeforeDelConfirm.....	702
AfterDelConfirm .....	703
Focus Events .....	704
Activate .....	704
Deactivate .....	705
GotFocus.....	705
LostFocus.....	706
Mouse Events.....	706
Click.....	706
DblClick .....	707
MouseDown .....	707

MouseMove .....	709
MouseUp.....	709
MouseWheel.....	709
Keyboard Events.....	710
KeyDown .....	710
KeyPress.....	711
KeyUp.....	713
Error Events .....	715
Error .....	715
Filter Events .....	716
Filter .....	716
ApplyFilter.....	718
Timing Events.....	719
Timer.....	720
Events Recognized by Form Sections.....	721
DblClick (Form Section Event) .....	721
Understanding and Using the OpenArgs Property.....	722
Summary .....	728
<b>Chapter 21 Events Recognized by Form Controls.....</b>	<b>729</b>
Enter (Control) .....	730
BeforeUpdate (Control) .....	732
AfterUpdate (Control).....	733
NotInList (Control) .....	735
Click (Control) .....	737
DblClick (Control).....	742
Chapter Summary .....	745
<b>Chapter 22 Enhancing Access Reports and Using Report Events .....</b>	<b>747</b>
Creating Access Reports.....	748
Using Report Events .....	748
Open.....	748
Close.....	751
Activate .....	751
Deactivate .....	752
NoData.....	752

Page.....	753
Error .....	755
Events Recognized by Report Sections.....	756
Format (Report Section Event).....	756
Print (Report Section Event).....	759
Retreat (Report Section Event).....	762
Using the Report View .....	763
Sorting and Grouping Data .....	765
Saving Reports in .pdf or .xps File Format .....	766
Using the OpenArgs Property of the Report Object.....	767
Running Built-In Menu Commands from VBA.....	769
Creating a Report with VBA.....	771
Part I-Creating a Crosstab Query in the Query Design View .....	772
Part II-Creating a Query with VBA .....	775
Part III-Creating a Report with VBA .....	778
Part IV-Creating a Custom Form for the Query's Parameters.....	782
Part V-Running the Form and Report.....	785
Summary .....	786
<b>PART VI ENHANCING THE USER EXPERIENCE.....</b>	<b>787</b>
<b>Chapter 23 Customizing the Menu System in Access.....</b>	<b>789</b>
The Initial Access 2021 Window .....	789
Customizing the Navigation Pane .....	791
Using VBA to Customize the Navigation Pane .....	795
Locking the Navigation Pane .....	795
Controlling the Display of Database Objects.....	795
Setting Displayed Categories.....	797
Saving and Loading the Configuration of the Navigation Pane.....	798
A Quick Overview of the Access 2021 Ribbon Interface .....	800
Ribbon Programming with XML, VBA, and Macros .....	803
Creating the Ribbon Customization XML Markup .....	804
Loading Ribbon Customizations from an External XML Document ..	808
Part 1: Setting Access Options .....	809
Part 2: Setting Up the Programming Environment.....	810
Part 3: Writing VBA Code.....	810
Part 4: Calling the LoadRibbon Function from an Autoexec Macro.....	813

Part 5: Applying the Customized Ribbon .....	814
Embedding Ribbon XML Markup in a VBA Procedure .....	815
Storing Ribbon Customization XML Markup in a Table .....	816
Assigning Ribbon Customizations to Forms and Reports .....	822
Part 1: Creating Ribbon Customization for a	
Report Using a Local System Table .....	822
Part 2: Making Access Aware of the New Customization .....	824
Part 3: Assigning a Ribbon Customization to a Report.....	825
Using Images in Ribbon Customizations .....	826
Requesting Images via the loadImage Callback .....	827
Part 1: Creating Ribbon Customization	
for Loading Custom Images.....	828
Part 2: Setting Up the Programming Environment.....	829
Part 3: Writing the VBA Callback Procedures.....	830
Part 4: Making Access Aware of the New Customization .....	830
Requesting Images via the getImage Callback.....	831
Understanding Attributes and Callbacks .....	836
Using Various Controls in Ribbon Customizations .....	838
Creating Toggle Buttons .....	838
Creating Split Buttons, Menus, and Submenus.....	839
Creating Checkboxes.....	841
Creating Edit Boxes .....	843
Creating Combo Boxes and Drop Downs.....	844
Creating a Dialog Box Launcher .....	846
Disabling a Control .....	847
Repurposing a Built-in Control.....	848
Refreshing the Ribbon .....	849
The CommandBars Object and the Ribbon .....	852
Tab Activation and Group Auto-Scaling.....	854
Customizing the Backstage View.....	855
Customizing the Quick Access Toolbar (QAT).....	859
Summary .....	861
<b>PART VII ADVANCED CONCEPTS IN ACCESS VBA .....</b>	<b>863</b>
<b>Chapter 24 Creating Classes in VBA .....</b>	<b>865</b>
Important Terminology .....	866
Creating Custom Objects in Class Modules.....	867

Creating a Class.....	868
Variable Declarations .....	869
Defining the Properties for the Class.....	870
Creating the Property Get Procedures.....	872
Creating the Property Let Procedures.....	873
Creating the Class Methods .....	874
Creating an Instance of a Class .....	875
Event Procedures in Class Modules .....	877
Creating the User Interface.....	878
Running the Custom Application.....	889
Watching the Execution of Your Custom Object.....	890
Creating and Working with Collection Classes .....	893
The Collection Object.....	893
The Collection Class .....	894
Summary .....	898
<b>Chapter 25 Advanced Event Programming.....</b>	<b>899</b>
Sinking Events in Standalone Class Modules.....	900
Part 1: Database File Preparation .....	901
Part 2: Creating the cRecordLogger Class .....	902
Part 3: Creating an Instance of the Custom Class in the Form's Class Module .....	905
Part 4: Testing the cRecordLogger Custom Class.....	907
Part 5: Using the cRecordLogger Custom Class with another Form .....	908
Writing Event Procedure Code in Two Places.....	910
Responding to Control Events in a Class.....	910
Declaring and Raising Events.....	914
Summary .....	920
<b>PART VIII VBA AND MACROS.....</b>	<b>921</b>
<b>Chapter 26 Macros and Templates.....</b>	<b>923</b>
Macros or VBA? .....	924
Access 2021 Macro Security.....	924
Using the AutoExec Macro.....	928
Understanding Macro Actions, Arguments, and Program Flow .....	929

Creating and Using Macros in Access 2021 .....	932
Creating Standalone Macros .....	933
Running Standalone Macros .....	938
Creating and Using Submacros.....	940
Creating and Using Embedded Macros.....	941
Copying Embedded Macros.....	942
Examining Shadow Properties.....	946
Using Data Macros .....	948
Creating a Data Macro .....	950
Creating a Named Data Macro .....	957
Editing an Existing Named Macro .....	960
Calling a Named Macro from Another Macro .....	960
Using ReturnVars in Data Macros.....	960
Tracing Data Macro Execution Errors .....	962
Error Handling in Macros .....	965
Using Temporary Variables in Macros.....	968
Converting Macros to VBA Code .....	969
Converting a Standalone Macro to VBA .....	969
Converting Embedded Macros to VBA .....	971
Access Templates .....	973
Creating a Custom Blank Database Template.....	973
Understanding the .accdt File Format .....	973
Summary .....	978
<b>PART IX WORKING TOGETHER: VBA, XML, AND RESTAPI.....</b>	<b>979</b>

<b>Chapter 27 XML Features in Access 2021 .....</b>	<b>981</b>
XML and Access .....	982
What Is a Well-Formed XML Document? .....	983
Exporting XML Data .....	985
Understanding the XML Data File .....	987
Understanding the XML Schema File.....	990
Understanding the XSL Transformation Files .....	992
Viewing XML Documents Formatted with Stylesheets.....	996
Advanced XML Export Options.....	999
Data Export Options .....	1000
Schema Export Options .....	1001

Presentation Export Options.....	1002
Applying XSLT Transforms to Exported Data .....	1009
Import XML Data .....	1016
Importing a Schema File.....	1016
Importing an XML File.....	1017
Part 1: Creating a Custom Transformation File to be Used After the XML Data Import.....	1019
Part 2: Exporting the Customers and Related Orders Tables to an XML File.....	1020
Part 3: Importing to an Access Database Only Two Columns from the Customers Table and Five Columns from the Orders Table.....	1021
Programmatically Exporting to and Importing from XML .....	1022
Exporting to XML Using the ExportXML Method .....	1022
Transforming XML Data with the TransformXML Method .....	1028
Part 1: Creating a Custom Stylesheet for Transforming an XML Source File into Another XML Data File .....	1029
Part 2: Writing a VBA Procedure to Export and Transform Data.....	1031
Part 3: Importing the Transformed XML Data File to Access .....	1033
Part 4: Creating another transformation .....	1033
Importing to XML Using the ImportXML Method.....	1036
Manipulating XML Documents Programmatically .....	1037
Loading and Retrieving the Contents of an XML File.....	1038
Working with XML Document Nodes .....	1040
Retrieving Information from Element Nodes.....	1042
Retrieving Specific Information from Element Nodes.....	1044
Retrieving the First Matching Node.....	1045
Using ActiveX Data Objects with XML .....	1046
Saving an ADO Recordset as XML to Disk.....	1046
Attribute-Centric and Element-Centric XML.....	1049
Changing the Type of an XML File .....	1051
Applying an XSL Stylesheet.....	1052
Transforming Attribute-Centric XML Data into an HTML Table .....	1054
Loading an XML Document in Excel .....	1058
Summary .....	1061

<b>Chapter 28 Access and REST API .....</b>	<b>1063</b>
Introduction to a VBA Dictionary Object .....	1063
Accessing the VBA Dictionary .....	1064
Adding a Reference to the Microsoft Scripting Runtime Library .....	1064
Working with the Dictionary Object's Properties and Methods .....	1066
Dictionary versus Collection .....	1069
Action Item 28.1 .....	1069
Introduction to Regular Expressions .....	1070
Character Matching in RegExp Patterns .....	1071
Quantifiers in RegExp Patterns .....	1072
Using the RegExp Object in VBA.....	1073
The RegExp Object Declaration.....	1074
RegExp Properties .....	1074
RegEx Methods .....	1075
Writing VBA Programs Using the RegExp Object .....	1075
Introduction to REST API .....	1078
Accessing REST APIs with VBA.....	1080
Methods and Properties of the XMLHttpRequest Object .....	1081
Making a Basic GET Request.....	1085
Action Item 28.2 .....	1088
Overview of JSON .....	1088
Loading JSON Data into Access .....	1091
Parsing JSON with Third-Party Libraries .....	1097
Summary .....	1097
<b>Appendix: Installing Internet Information Services (IIS) .....</b>	<b>1099</b>
Creating a Virtual Directory.....	1102
Setting ASP Configuration Properties.....	1105
Turning Off Friendly HTTP Error Messages.....	1106
<b>Index .....</b>	<b>1109</b>

## *ACKNOWLEDGMENTS*

**F**irst, I'd like to express my gratitude to everyone at Mercury Learning and Information. A sincere thank-you to my publisher, David Pallai, for offering me the opportunity to update this book to the new 2021 version and tirelessly keeping things on track during this long project.

A whole bunch of thanks go to the editorial team for working so hard to bring this book to print. In particular, I would like to thank Jennifer Blaney, for her production expertise. To the composito, Swaradha Typesetters, for all the composition efforts that gave this book the right look and feel.

Special thanks to my husband, Paul, for his patience during this long project.

Finally, I'd like to acknowledge readers like you who cared enough to post reviews of the previous editions of this book online. Your invaluable feedback has helped me raise the quality of this work by including the material that matters to you most. Please continue to inspire me with your ideas and suggestions.

**Julitta Korol**  
**July 2022**



# *INTRODUCTION*

Since its creation, Microsoft Access has allowed users to design and develop Windows-based database applications and has grown into the world's most popular database. This book is for people who have already mastered the use of Microsoft Access databases and now are ready for the next step—programming. *Access 2021/Microsoft 365 Programming by Example* takes nonprogrammers through detailed steps of creating Access databases from scratch and shows them how to retrieve and manage their data programmatically using various programming languages and techniques. With this book in hand, users can quickly build the toolset required for developing their own database solutions. With this book's approach, programming an Access database from scratch and controlling it via programming code is as easy as designing and maintaining databases with the built-in tools of Access. This book gives a practical overview of many programming languages and techniques necessary in programming, maintaining, and retrieving data from today's Access databases.

## **PREREQUISITES**

---

You don't need any programming experience to use *Access 2021/Microsoft 365 Programming by Example*. The only prerequisite is that you already know how to manually design an Access database and perform database tasks by creating and running various types of queries. This book also assumes that you know how to create more complex forms with embedded subforms, combo boxes, and other built-in controls. If you don't have these skills, there are countless books on the market that can teach you step by step how to build simple databases. If you do meet these criteria, this book will take you to the Access programming level by example. You will gain working knowledge immediately by performing concrete tasks and without having to read long descriptions of concepts. True learning by

example begins with the first step, followed by the next step, and the next one, and so on. By the time you complete all the steps in a hands-on exercise or a custom project, you should be able to effectively apply the same technique again and again in your own database projects.

## HOW THIS BOOK IS ORGANIZED

---

This book is divided into nine parts (a total of 28 chapters) that progressively introduce you to programming Access databases.

### PART I—ACCESS VBA PRIMER

---

Here you are introduced to Visual Basic for Applications (VBA)—the programming language for Microsoft Access. In this part of the book, you acquire the fundamentals of VBA that you will use repeatedly in building real-life Access database applications. Part I chapters are also the subject of a standalone book, *Access 2021 Programming Pocket Primer*, available from Mercury Learning and Information (ISBN: 9781683928898). If you already worked through the pocket primer book, you can skip chapters 1–9 and begin from Chapter 10.

*Part I consists of the following nine chapters:*

*Chapter 1—Getting Started with Access VBA*

In this chapter you learn about the types of Access procedures you can write and learn how and where they are written.

*Chapter 2—Getting to Know Visual Basic Editor (VBE)*

In this chapter you learn almost everything you need to know about working with the Visual Basic Editor window, commonly referred to as VBE. Some of the programming tools that are not covered here are discussed and used in Chapter 9.

*Chapter 3—Access VBA Fundamentals*

This chapter introduces basic VBA concepts that allow you to store various pieces of information for later use.

*Chapter 4—Access VBA Built-In and Custom Functions*

In this chapter you find out how to provide additional information to your procedures and functions before they are run.

*Chapter 5—Adding Decisions to Your Access VBA Programs*

In this chapter you learn how to control your program flow with several different decision-making statements.

*Chapter 6—Adding Repeating Actions to Your Access VBA Programs*

In this chapter you learn how to repeat the same actions in your code by using looping structures.

*Chapter 7—Keeping Track of Multiple Values Using Arrays*

In this chapter you learn about static and dynamic arrays and how to use them for holding various values.

*Chapter 8—Keeping Track of Multiple Values Using Collections*

In this chapter you learn how you can maintain your items of data while your program is running by using a special type of object – the collection.

*Chapter 9—Getting to Know Built-In Tools for Testing and Debugging*

In this chapter you begin using built-in debugging tools to test your programming code. You also learn how to add effective error-handling code to your procedures.

The above nine chapters will give you the fundamental techniques and concepts you will need to continue your Access VBA learning path. The skills obtained in Access VBA Primer are portable. They can be utilized in programming other Microsoft Office applications that also use VBA as their native programming language such as Excel, Word, PowerPoint, Outlook, and so on.

---

## PART II—ACCESS VBA PROGRAMMING WITH DAO AND ADO

---

Here you are introduced to two sets of programming objects known as Data Access Objects (DAO) and ActiveX Data Objects (ADO) that enable Microsoft Access and other client applications to access and manipulate data. You learn how to use DAO and ADO objects in your VBA code to connect to a data source, as well as create, modify, and manipulate database objects.

*Part II consists of the following three chapters:*

*Chapter 10—Data Access Technologies in Microsoft Access*

In this chapter you get acquainted with two database engines (Jet/ACE) that Access uses, as well as several object libraries that provide objects, properties, and methods for your VBA procedures.

*Chapter 11—Creating and Manipulating Databases with DAO*

This chapter demonstrates how to create, copy, link, and delete database tables programmatically by using objects from the DAO object library. Here you learn how to write code to add and delete fields as well as create listings of existing tables in a database and fields in a table. You add primary keys and indexes to your database tables and create relationships between your tables. Next you practice various methods of using programming code to open a set of database records, commonly referred to as a recordset. You learn how to move around in a recordset and find, filter, and sort the required records, as well as read their contents. Finally, you learn how to perform essential database operations such as adding, updating, and deleting records. You also learn how to render your database records in three popular file formats like Excel, Word, and a text file. Creating and running various types of database queries using VBA instead of the Query Design view is also covered in this chapter.

*Chapter 12—Creating and Manipulating Database with ADO.*

This chapter demonstrates how to create, copy, link, and delete database tables programmatically by using objects from the ADO object library. Here you learn how to write code to add and delete fields as well as create listings of existing tables in a database and fields in a table. You add primary keys and indexes to your database tables. You also learn how to use objects from the ADOX library to create relationships between your tables. Next you practice various methods of using programming code to open a set of database records, commonly referred to as a recordset. You learn how to move around in a recordset and find, filter, and sort the required records, as well as read their contents. Finally, you learn how to perform essential database operations such as adding, updating, and deleting records. You also learn how to render your database records in three popular file formats like Excel, Word, and a text file. Creating and running various types of database queries using VBA instead of the Query Design view is also covered in this chapter. This chapter explains several advanced ADO features such as how to disconnect a recordset from a database, save it in a disk file, clone it, and shape it. You also learn about database transactions.

You will find the skills obtained in Part II of this book essential for accessing and manipulating Access databases.

---

**PART III—ACCESS STRUCTURED QUERY LANGUAGE (SQL)**

---

Here you are introduced to the Data Definition Language (DDL), an important component of the Structured Query Language (SQL). Like ADO and DAO, which

were introduced in Part II, DDL is used for defining database objects (tables, views, stored procedures, primary keys, indexes, and constraints) and managing database security. In this part of the book, you learn how to use DDL statements with Access databases.

*Part III consists of the following four chapters:*

*Chapter 13—Creating, Modifying, and Deleting Tables and Fields*

In this chapter you learn special Data Definition Language commands for creating a new Access database, as well as creating, modifying, and deleting tables. You also learn commands for adding, modifying, and deleting fields and indexes.

*Chapter 14—Enforcing Data Integrity and Relationships between Tables*

Here you learn how to define rules regarding the values allowed in table fields to enforce data integrity and relationships between tables.

*Chapter 15—Defining Indexes and Primary Keys*

Here you learn DDL commands for creating indexes and primary keys.

*Chapter 16—Views and Stored Procedures*

This chapter shows you how to work with two powerful database objects known as views and stored procedures. You learn how views are like SELECT queries, and how stored procedures can perform various actions like Access Action queries and Select queries with parameters.

The skills you learn in Part III of this book will allow you to create and manipulate your Access databases using SQL DDL statements. Numerous Access SQL DDL statements and concepts introduced here are important in laying the groundwork for moving into the client/server environment (porting your Microsoft Access database to SQL Server).

---

## PART IV—IMPLEMENTING ACCESS DATABASE SECURITY

---

Here we focus on Securing Access databases in the .ACCDB and .MDB file formats.

*Part IV consists of the following two chapters:*

*Chapter 17—Implementing Database Security with DDL*

In this chapter you learn how to use DDL commands and ADO objects to manage share-level security in the Microsoft Access database. You learn how to quickly create, modify, and remove a database password, and how to manage user-level accounts.

*Chapter 18—Implementing User-Level and Share-Level Security*

In this chapter you learn about two types of security in Microsoft Access databases: share-level security that applies to both older (MDB) and new (ACCDB) Access databases, and user-level security that can only be used with .mdb files.

The skills learned in Part IV will allow you to build more secure multiuser Access database applications that preserve and protect both data and the application itself. With the understanding of elements of security available in both file formats, you can create robust database solutions that will be less vulnerable to the malicious attacks on your computers and entire networks. Be sure to work through both security chapters before deciding which method of security is best for your database application.

---

**PART V—PROGRAMMING IN ACCESS FORMS AND REPORTS**

---

Here you learn how to respond to events that occur in Access forms and reports. The behavior of Microsoft Access objects such as forms, reports, and controls can be modified by writing programming code known as an event procedure or an event handler. In this part of the book, you learn how you can make your forms, reports, and controls perform useful actions by writing event procedures in form and report class modules.

*Part V consists of the following four chapters:*

*Chapter 19—Enhancing Access Forms*

This chapter presents a quick overview of types of forms you can create with Access 2021 and types of formatting you can apply to make your forms more attractive. You learn how you can group form controls using the layouts, implement rich formatting in form controls, professionally format your forms using built-in themes, and enhance forms with images.

*Chapter 20—Using Form Events*

In this chapter you learn the types of events that can occur on a Microsoft Access form and write event procedures to handle various form events.

*Chapter 21—Events Recognized by Form Controls*

In this chapter you learn the types of events that can occur on a Microsoft Access form, and you learn how to write event procedures to handle various form events.

*Chapter 22—Enhancing Access Reports and Using Report Events*

In this chapter you learn about many events that are triggered when an Access report is run. You write your own event procedures to specify what happens when the report is opened, activated/deactivated, or closed.

The skills acquired in Part V will help you create forms and reports that provide the desired functionality to your users thanks to the implementation of various events.

## PART VI – ENHANCING THE USER EXPERIENCE

---

Since its 2007 release, Access like other Microsoft 365 applications, uses the Ribbon interface for its menu system. Knowing how the Ribbon works and how you can modify it to customize your Access databases will enhance the experience of your database users. We will cover this topic in one big chapter with numerous illustrated hands-on examples and programming code written in VBA and XML.

### *Chapter 23 – Customizing the Menu System in Access*

This chapter provides an overview of the programming elements available in the Ribbon and shows how you can customize the user interface (UI) in your Access database applications. You learn how to create XML Ribbon customization markup and load it in your database. You also learn how Ribbon customizations can be assigned to forms or reports.

The skills acquired in Part VI of this book will allow you to enhance and alter the way users interact with your database application.

## PART VII—ADVANCED CONCEPTS IN ACCESS VBA

---

Microsoft Access offers numerous built-in objects that you can access from your VBA procedures to automate many aspects of your databases. You are not limited to using these built-in objects, however. VBA allows you to create your own objects and collections of objects, complete with their own methods and properties. In this part of the book, you learn how thinking in terms of objects can help you write reusable code that's easy to maintain. In the next two chapters you'll be working in a new type of module, known as a class module, creating, and using classes and responding to class events.

### *Chapter 24—Creating Classes*

In this chapter you will work with advanced VBA concepts: VBA classes, class objects, and collection classes.

### *Chapter 25—Advanced Event Programming*

This chapter teaches advanced concepts in event programming. You learn how to respond to events in standalone class modules to make your code more manageable and portable to other objects. You also learn how to create and raise your own events.

The skills acquired in Part VII of this book will allow you write VBA code that is more efficient, much easier to read and maintain, and can be reused in many places.

## **PART VIII—MACROS AND TEMPLATES**

---

Here you are introduced to three types of macros that you can create in Access 2021. In addition, you learn how to convert macros to VBA and get started with built-in templates that extensively use macros. Writing VBA code is not the only way to provide rich functionality to your Access database users. Macros have long been used to enhance the user experience without users having to write any VBA code. Access 2021 Macro Builder allows you to include complex logic, business rules, and error handling in your macros.

*Part VIII contains the following chapter:*

### *Chapter 26—Macros and Templates*

This chapter introduces you to using macros. We take a detailed look at macro security, work with three types of macros (standalone, embedded, and data macros), see examples of using variables in macros, and examine error-handling actions in macros. We also discuss working with the template format in Access 2021.

The skills acquired in Part VIII will allow you to correctly utilize many of the macros available in Microsoft provided database templates in your own custom Access applications.

## **PART IX—WORKING TOGETHER: VBA, XML AND REST API**

---

Extensible Markup Language (XML) has long been the standard format for sharing data without regard for the originating application or the operating system. In this part of the book, you learn how XML is used in Access to bring external data to your database as well as provide your data to other applications. You also learn about the Rest APIs, the newest and the most flexible method of integrating applications. You will use your Access VBA and XML skills to make HTTP requests to a Webserver to retrieve data and integrate it with Access. In this process you are introduced to using JSON (JavaScript Object Notation), the most popular file format for storing and transporting data.

*Part IX consists of the following two chapters:*

*Chapter 27—XML Features in Access 2021*

In this chapter you learn how to use the Extensible Markup Language (XML) with Access. You learn how to export Access data manually and programmatically to XML files, as well as import an XML file to Access and display its data in a table. You also learn how to use stylesheets and transformations to present Access data to users in a desired format.

*Chapter 28—Access and REST API*

This chapter focuses on expanding your VBA skillset by covering topics such as working with a VBA Dictionary Object, using regular expressions, and calling a new type of a web service, known as REST API.

The skills acquired in Part IX will make your Access applications ready to integrate with any operating system or web-based platform.

---

**APPENDIX—INSTALLING INTERNET INFORMATION SERVICES (IIS)**

---

This Appendix walks you through the installation of a Web server application that is used in Chapter 27 Hands-On projects.

---

**HOW TO WORK WITH THIS BOOK**

---

This book has been designed as a tutorial and should be followed chapter by chapter.

As you read each chapter, perform the tasks that are described. Be an active learner by getting involved in the book’s hands-on exercises and custom projects. When you are completely involved, you learn things by doing rather than studying, and you learn faster. Do not move on to new information until you’ve fully grasped the current topic. Allow your brain to sort things out and put them in proper perspective before you move on. Take frequent breaks between your learning sessions, as some chapters in this book cover lots of material. Do not try to do everything in one sitting. It’s always better to divide the material into smaller units than attempt to master all there is to learn at once. However, never stop in the middle of a hands-on exercise; finish it before taking a break. After learning a particular technique or command, try to think of ways to apply it to your own work. As you work with this book, create small sample procedures for yourself

based on what you've learned up to a particular point. These procedures will come in handy when you need to review the subject in the future or simply need to steal some ready-made code.

## **THE COMPANION FILES**

---

The example files for all the hands-on activities in this book are available in the companion files included with this book and may also be downloaded by contacting the publisher at [info@merclearning.com](mailto:info@merclearning.com). Digital versions of this title are available at [academiccourseware.com](http://academiccourseware.com) and other digital vendors.

# *ACCESS VBA PRIMER*

## Part

The Access VBA Primer is divided into nine chapters that progressively introduce you to programming Microsoft Access using the 2021 version of the product. These chapters present the fundamental techniques and concepts that you need to master before you can take further steps in Access programming.

- Chapter 1 Getting Started with Access VBA
- Chapter 2 Getting to Know Visual Basic Editor (VBE)
- Chapter 3 Access VBA Fundamentals
- Chapter 4 Access VBA Built-In and Custom Functions
- Chapter 5 Adding Decisions to Your Access VBA Programs
- Chapter 6 Adding Repeating Actions to Your Access VBA Programs
- Chapter 7 Keeping Track of Multiple Values Using Arrays
- Chapter 8 Keeping Track of Multiple Values Using Collections
- Chapter 9 Getting to Know Built-in Tools for Testing and Debugging



# Chapter 1

# GETTING STARTED WITH ACCESS VBA

Visual Basic for Applications (VBA) is the programming language built into all Microsoft® 365 applications, including Access®. In this chapter, you acquire the fundamentals of VBA that you will use over and over again in building real-life Access database applications.

## **UNDERSTANDING VBA MODULES AND PROCEDURE TYPES**

Your job as a programmer boils down to writing various procedures that address specific problems. A *procedure* is a group of instructions that allows you to accomplish certain tasks when your program runs. When you place instructions (programming code) in a procedure, you can call this procedure whenever you need to perform that task. Although many tasks can be automated in Access by using macro actions, such as opening forms and reports, finding records, and executing queries, you will need VBA skills to perform advanced customizations in your Access databases.

In VBA you can write four types of procedures: *subroutine procedures*, *function procedures*, *event procedures*, and *property procedures*. Procedures are created and stored in modules. A module resembles a blank document in Microsoft Word. Each procedure in the same module must have a unique name; however, procedures in different modules can have the same name. Let's learn a bit about

each procedure type so that you can quickly recognize them when you see them in books, magazine articles, or online.

### 1. Subroutine procedures (also called subroutines or subprocedures)

Subroutine procedures perform useful tasks but never return values. They begin with the keyword `Sub` and end with the keywords `End Sub`. *Keywords* are words that carry a special meaning in VBA. Let's look at the simple subroutine `ShowMessage` that displays a message to the user:

```
Sub ShowMessage()  
    MsgBox "This is a message box in VBA."  
End Sub
```

Notice a pair of empty parentheses after the procedure name. The instruction that the procedure needs to execute is placed on a separate line between the `Sub` and `End Sub` keywords. You may place one or more instructions and even complex control structures within a subroutine procedure. Instructions are also called *statements*. The `ShowMessage` procedure will always display the same message when executed. `MsgBox` is a built-in VBA function often used for programming user interactions (see Chapter 4, “Access VBA Built-In and Custom Functions,” for more information on this function).

If you'd like to write a more universal procedure that can display a different message each time the procedure is executed, you will need to write a subroutine that takes arguments. *Arguments* are values that are needed for a procedure to do something. Arguments are placed within the parentheses after the procedure name. Let's look at the following procedure that also displays a message to the user; however, this time we can pass any text string to display:

```
Sub ShowMessage2(strMessage As String)  
    MsgBox strMessage  
End Sub
```

This subprocedure requires one text value before it can be run; `strMessage` is the arbitrary argument name. It can represent any text you want. Therefore, if you pass it the text “Today is Monday,” that is the text the user will see when the procedure is executed. If you don't pass the value to this procedure, VBA will display an error. Notice that following the argument name, we can also specify the data type for the argument. Data types are discussed in Chapter 3. If the data type is not specified, Access will use its default data type (Variant) for this argument. The built-in `MsgBox` function requires that we pass to it a text string, so we can use String data type for the `strMessage` argument. All this information is covered in detail as you progress through this book.

If your subprocedure requires more than one argument, list the arguments within the parentheses and separate them with commas. For example, let's improve the preceding procedure by also passing it a text string containing a user name:

```
Sub ShowMessage3(strMessage As String, strUserName as String)
    MsgBox strUserName & ", your message is: " & strMessage
End Sub
```

The ampersand (&) operator is used for concatenating text strings inside the VBA procedure. If we pass to the above subroutine the text “Keep on learning.” As the `strMessage` argument and “John” as the `strUserName` argument, the procedure will display the following text in a message box:

John, your message is: Keep on learning.

## 2. Function procedures (functions)

Functions perform specific tasks and can return values. They begin with the keyword `Function` and end with the keywords `End Function`. Let's look at a simple function that adds two numbers:

```
Function addTwoNumbers()
    Dim num1 As Integer
    Dim num2 As Integer

    num1 = 3
    num2 = 2
    addTwoNumbers = num1 + num2
End Function
```

The preceding function procedure always returns the same result, which is the value 5. The `Dim` statements inside this function procedure are used to declare variables that the function will use. A *variable* is a name that is used to refer to an item of data. Because we want the function to perform a calculation, we specify that the variables will hold integer values. Variables and data types are covered in detail in Chapter 3, “Access VBA Fundamentals.”

The variable definitions (the lines with the `Dim` statements) are followed by the variable assignment statements in which we assign specific numbers to the variables `num1` and `num2`. Finally, the calculation is performed by adding together the values held in both variables: `num1 + num2`. To return the result of our calculation, we set the function name to the value or the expression we want to return:

```
addTwoNumbers = num1 + num2
```

Although this function example returns a value, not all functions have to return values. Functions, like subroutines, can perform actions without returning any values.

Like procedures, functions can accept arguments. For example, to make our addTwoNumbers function more versatile, we can rewrite it as follows:

```
Function addTwoNumbers2(num1 As Integer, num2 As Integer)
    addTwoNumbers2 = num1 + num2
End Function
```

Now we can pass any two numbers to the preceding function to add them together. For example, we can write the following statement to display the result of the function in a message box:

```
Sub DisplayResult()
    MsgBox("Total=" & addTwoNumbers2(34, 80))
End Sub
```

Notice that the procedure DisplayResult calls two functions: the built-in MsgBox function and your custom addTwoNumbers2 function. The result of the addTwoNumbers2 function is concatenated to the “Total=” string and then displayed to the user.

### 3. Event procedures

Event procedures are automatically executed in response to an event initiated by the user or program code or triggered by the system. Access provides numerous events that you can respond to, like the Click, Double-Click, Open, Load, and Focus events. Events, event properties, and event procedures are introduced later in this chapter. They are also covered in Chapter 9, “Getting to Know Built-In Tools for Testing and Debugging”, Chapter 20, “Using Form Events”, Chapter 22, “Enhancing Access Reports and Using Report Events”, and Chapter 25, “Advanced Event Programming”.

### 4. Property procedures

Property procedures are used to get or set the values of custom properties for forms, reports, and class modules. The three types of property procedures (*Property Get*, *Property Let*, and *Property Set*) begin with the `Property` keyword followed by the property type (`Get`, `Let`, or `Set`), the property name, and a pair of empty parentheses, and end with the `End Property` keywords. Here’s an example of a property procedure that retrieves the value of an author’s royalty:

```
Property Get Royalty()
    Royalty = (Sales * Percent) - Advance
End Property
```

Property procedures are a more advanced feature in Access programming and are covered in detail in Chapter 24, “Creating Classes in VBA”.

## WRITING PROCEDURES IN A STANDARD MODULE

As mentioned earlier, procedures are created and stored in modules. Access has two types of modules: *standard module* and *class module*. Standard modules are used to hold subprocedures and function procedures that can be run from anywhere in the application because they are not associated with any form or report.

Because we already have a couple of procedures to try out, let’s do a quick hands-on exercise to learn how to open standard modules, write procedures, and execute them.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



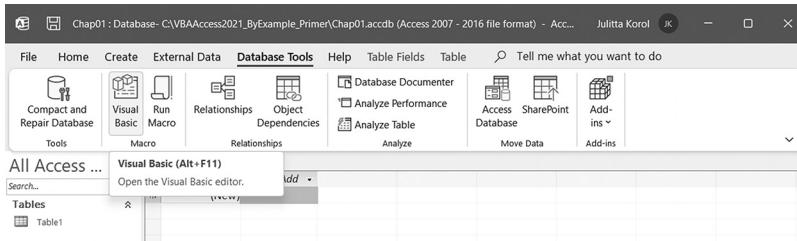
### Hands-On 1.1 Working in a Standard Module

1. Create a folder on your hard drive named **C:\VBAAccess2021\_ByExample\_Primer**.
2. Open Access and click **Blank database**. Type **Chap01** in the File Name box and click the folder button to set the location for the database to the **C:\VBAAccess2021\_ByExample\_Primer** folder. Finally, click the **Create** button to create the specified database (see Figure 1.1). Access will create the database in its default .ACCDB format.



FIGURE 1.1. Creating a blank desktop Access database.

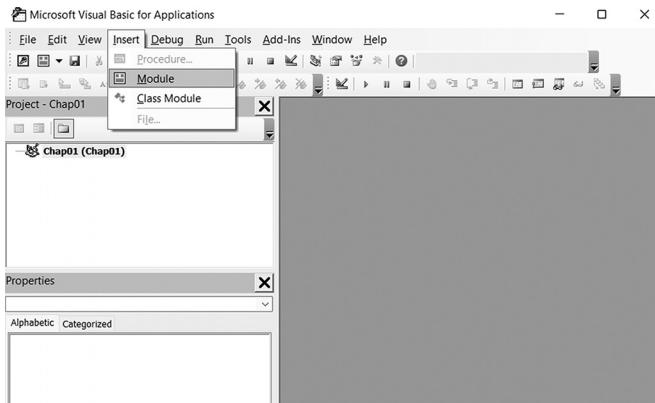
- To launch the programming environment, select the **Database Tools** tab and click **Visual Basic** (see Figure 1.2). You can also press **Alt+F11** to get to this screen.



**FIGURE 1.2.** Activating a Visual Basic development environment.

The screen that opens is your Visual Basic Environment, often referred to as VBE. All your coding will be performed in this screen. Before you can do your work here, you need to determine which module you need to work with. As mentioned earlier, we use a standard module for most general programming tasks. Initially nothing is open, so let's add the first module.

- Insert a standard module by choosing **Module** from the **Insert** menu (see Figure 1.3).



**FIGURE 1.3.** Inserting a standard module.

Each module begins with a declaration section that lists various settings and declarations that apply to every procedure in the module. Figure 1.4 shows the default declaration. `Option Compare Database` specifies how string comparisons are evaluated in the module—whether the comparison is case-

sensitive or insensitive. This is a case-insensitive comparison that respects the sort order of the database. This means that “a” is the same as “A.” If you delete the `Option Compare Database` statement, the default string comparison setting for the module is `Option Compare Binary` (used for case-sensitive comparisons where “a” is not the same as “A”).

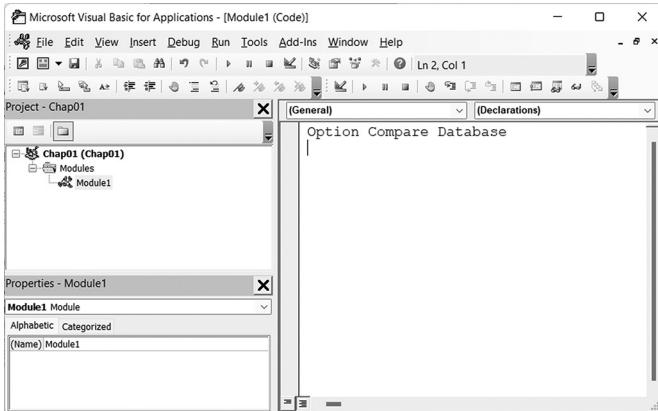


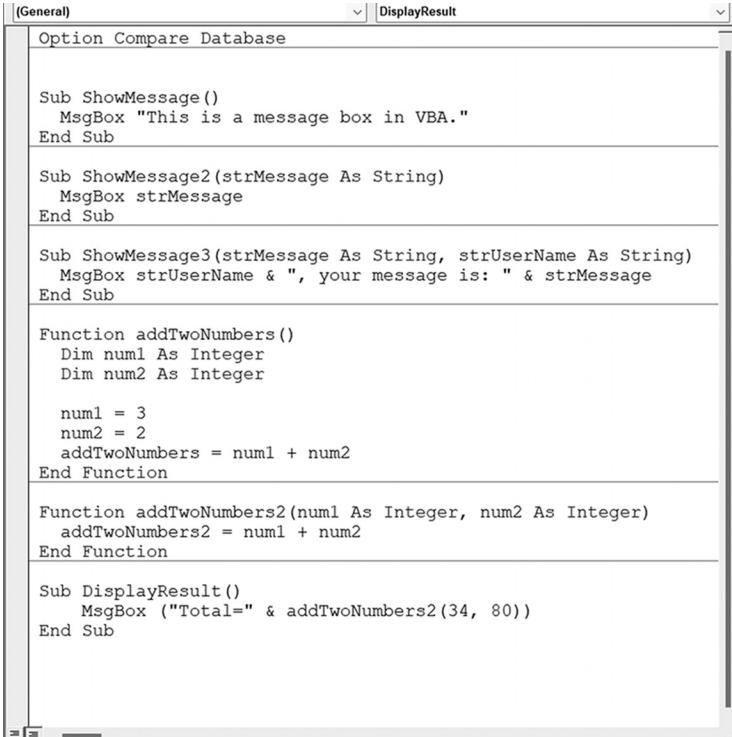
FIGURE 1.4. Standard module.

Another declaration (not shown here) called the `Option Explicit` statement is often used to ensure that all variables used within this module are formally declared. You will learn about this statement and variables in Chapter 4.

Following the declaration section is the procedure section, which holds the module's procedures. You can begin writing your procedures at the cursor position within the `Module1 (Code)` window.

5. In the `Module1 (Code)` window, enter the code of subroutines and function procedures as shown in Figure 1.5. These are the same sub procedures and functions that we've discussed so far in this chapter. You can write procedures and functions in the same module, or you can keep them separate by adding another standard module to your VBA project.

Notice that Access inserts a horizontal line after each `End Sub` or `End Function` keyword to make it easier to identify each procedure. The Procedure drop-down box at the top-right corner of the `Module1 (Code)` window displays the name of the procedure in which the insertion point is currently located.



The screenshot shows a Microsoft Access VBA code editor window. The title bar says '(General)' and 'DisplayResult'. The code in the module is as follows:

```
Option Compare Database

Sub ShowMessage()
    MsgBox "This is a message box in VBA."
End Sub

Sub ShowMessage2(strMessage As String)
    MsgBox strMessage
End Sub

Sub ShowMessage3(strMessage As String, strUserName As String)
    MsgBox strUserName & ", your message is: " & strMessage
End Sub

Function addTwoNumbers()
    Dim num1 As Integer
    Dim num2 As Integer

    num1 = 3
    num2 = 2
    addTwoNumbers = num1 + num2
End Function

Function addTwoNumbers2(num1 As Integer, num2 As Integer)
    addTwoNumbers2 = num1 + num2
End Function

Sub DisplayResult()
    MsgBox ("Total=" & addTwoNumbers2(34, 80))
End Sub
```

FIGURE 1.5. Standard module with subprocedures and functions.

## EXECUTING YOUR PROCEDURES

---

Now that you've filled the standard module with some procedures and functions, let's see how you can run them. There are many ways of running (executing) your code. In the next hands-on exercise, you will execute your code in four different ways using:

- Run menu (Run Sub/UserForm)
- Toolbar button (Run Sub/UserForm)
- Keyboard (F5)
- Immediate window



## Hands-On 1.2 Running Procedures and Functions

1. Place the insertion point anywhere within the ShowMessage procedure. The Procedure box in the top-right corner of the Module1 (Code) window should display ShowMessage. Choose **Run Sub/UserForm** from the **Run** menu. Access runs the selected procedure and displays the message box with the text "This is a message box in VBA."
2. Click **OK** to close the message box. Try running this procedure again, this time by pressing the **F5** key on the keyboard. Click **OK** to close the message box. If the Access window seems stuck and you can't activate any menu option, this is often an indication that there is a message box open in the background. Access will not permit you to do any operation until you close the pop-up window.
3. Now, run this procedure for the third time by clicking the **Run Sub/UserForm** button (▶) on the toolbar. This button has the same tooltip as the Run Sub/UserForm (F5) option on the Run menu.

**NOTE**

*Procedures that require arguments cannot be executed directly using the methods you just learned. You need to type some input values for these procedures to run. A perfect place to do this is the Immediate window, which is covered in detail in Chapter 2, "Getting to Know Visual Basic Editor (VBE)." For now, let's open this window and see how you can use it to run VBA procedures.*

4. Select **Immediate Window** from the **View** menu.

Access opens a small window and places it just below the Module1 (Code) window. You can size and reposition this window as needed. Figure 1.6 shows statements that you will run from the Immediate window in Steps 5–8.

5. Type the following in the Immediate window and press **Enter** to execute.

```
ShowMessage2 "I'm learning VBA."
```

Access executes the procedure and displays the message in a message box. Click **OK** to close the message box. Notice that to execute the ShowMessage2 procedure, you need to type the procedure name, a space, and the text you want to display. The text string must be surrounded by double quotation marks. In a similar way you can execute the ShowMessage3 procedure by providing two required text strings. For example, on a new line in the Immediate window, type the following statement and press **Enter** to execute:

```
ShowMessage3 "Keep on learning.", "John"
```

When you press the Enter key, Access executes the ShowMessage3 procedure and displays the text “John, your message is: Keep on learning.” Click **OK** to close this message box.

**NOTE**

You can also use the `Call` statement to run a procedure in the Immediate window. When using this statement, you must place the values of arguments within parentheses, as shown here:

```
Call ShowMessage3("Keep on learning.", "John")
```

Function procedures are executed using different methods. Step 6 demonstrates how to call the addTwoNumbers function.

6. On a new line in the Immediate window, type a question mark followed by the name of the function procedure and press **Enter**:

```
?addTwoNumbers
```

Access should display the result of this function (the number 5) on the next line in the Immediate window.

7. Now run the addTwoNumbers2 procedure. Type the following instruction in the Immediate window and press **Enter**:

```
?addTwoNumbers2(56, 24)
```

Access displays the result of adding these two numbers on the next line.

8. If you'd rather see the function result in a message box, type the following instruction in the Immediate window and press **Enter**:

```
MsgBox("Total=" & addTwoNumbers2(34, 80))
```

Access displays a message box with the text “Total=114”.

**NOTE**

See Chapter 2 for more information on running your procedures and functions from the Immediate window.

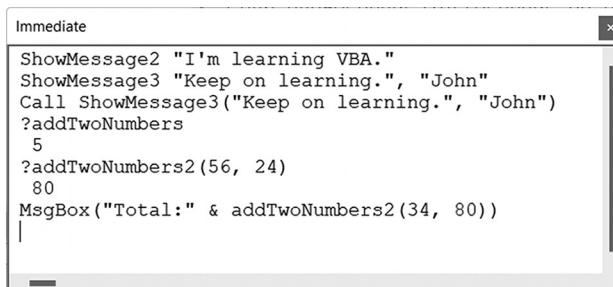


FIGURE 1.6. Running procedures and functions in the Immediate window.

Now that you've familiarized yourself a bit with standard modules, let's move on to another type of module known as the class module.

## UNDERSTANDING CLASS MODULES

---

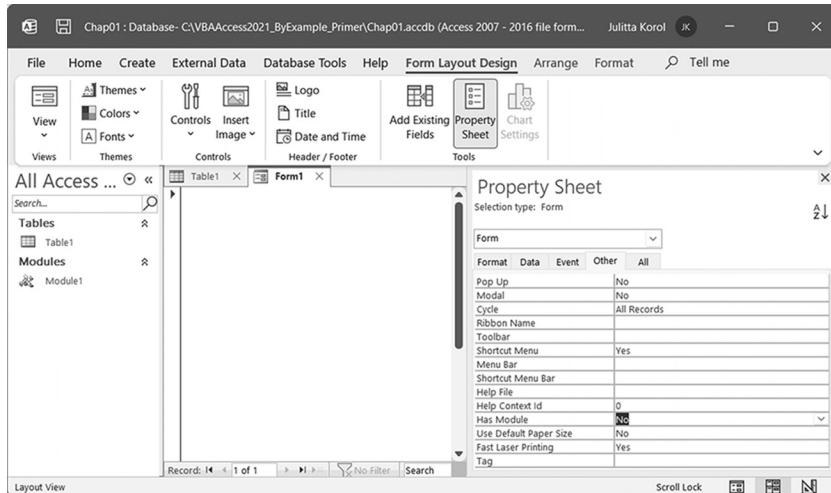
Class modules come in three varieties: *standalone class modules*, *form modules*, and *report modules*.

- 1. Standalone class modules**—These modules are used to create your own custom objects with their own properties and methods. You create a standalone class module by choosing **Insert | Class Module** in the Microsoft Visual Basic for Applications window. Access will create a default class module named *Class1* and will list it in the Class modules folder in the Project Explorer window. You will work with standalone class modules in Chapter 8.
- 2. Form modules**—Each Access form can contain a form module, which is a special type of a class module.
- 3. Report modules**—Each Access report can contain a report module, which is a special type of class module.

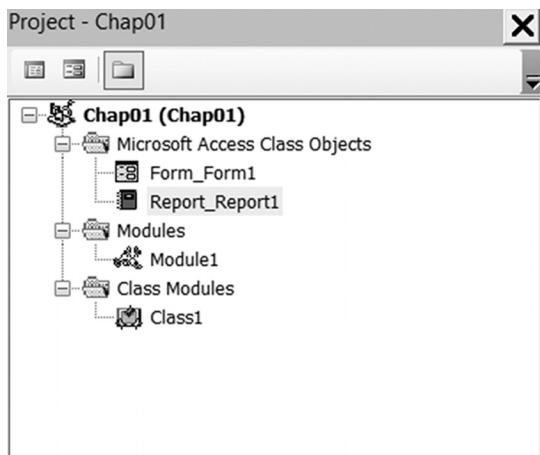
All newly created forms and reports are lightweight by design because they don't have modules associated with them when they're first created. Therefore, they load and display faster than forms and reports with modules. These lightweight forms and reports have their Has Module property set to No (see Figure 1.7). When you open a form or report in Design view and click the View Code button ( ) in the Tools section of the Form Design or Report Design tab, Access creates a form or report module. The Has Module property of a form or report is automatically set to Yes to indicate that the form or report now has a module associated with it. Note that this happens even if you have not written a single line of VBA code. Access opens a module window and assigns a name to the module that consists of three parts: the name of the object (e.g., form or report), an underscore character, and the name of the form or report. For example, a newly created form that has not been saved is named *Form\_Form1*, a form module in the Customers form is named *Form\_Customers*, and a report module in the Customers report is named *Report\_Customers* (see Figure 1.8).

As with report modules, form modules store event procedures for events recognized by the form and its controls, as well as general function procedures and subprocedures. You can also write Property Get, Property Let, and Property Set procedures to create custom properties for the form or report. The procedures

stored in their class modules are available only while you are using that form or report.



**FIGURE 1.7.** When you begin designing a new form in the Access user interface, the form does not have a module associated with it. Notice that the Has Module property on the form's property sheet is set to No.



**FIGURE 1.8.** Database modules are automatically organized in folders. Form and report modules are listed in the Microsoft Access Class Objects folder. Standard modules can be found in the Modules folder. The Class Modules folder organizes standalone class modules.

## EVENTS, EVENT PROPERTIES, AND EVENT PROCEDURES

---

To customize your database applications or to deliver products that fit your users' specific needs, you'll be doing quite a bit of event-driven programming. Access is an *event-driven* application. This means that whatever happens in an Access application is the result of an event that Access has detected. *Events* are things that happen to objects and can be triggered by the user or by the system, such as clicking a mouse button, pressing a key, selecting an item from a list, or changing a list of items available in a listbox. As a programmer, you will often want to modify the application's built-in response to a particular event. Before the application processes the user's mouseclicks and keypresses in the usual way, you can tell the application how to react to the activity. For example, if a user clicks a Delete button on your form, you can display a custom delete confirmation message to ensure that the user selected the intended record for deletion.

For each event defined for a form, form control, or report, there is a corresponding *event property*. If you open any Access form in Design view and choose Properties in the Tools section of the Form Design tab, and then click the Event tab of the property sheet, you will see a long list of events your form can respond to (see Figure 1.9).

Forms, reports, and the controls that appear on them have various event properties you can use to trigger desired actions. For example, you can open or close a form when a user clicks a command button, or you can enable or disable controls when the form loads.

To specify how a form, report, or control should respond to events, you can write *event procedures*. In your programming code, you may need to describe what should happen if a user clicks on a command button or selects from a combo box. For example, when you design a custom form, you should anticipate and program events that can occur at runtime (while the form is being used). The most common event is the Click event. Every time a command button is clicked, it triggers an event procedure to respond to the Click event for that button.

When you assign your event procedure to an event property, you set an *event trap*. Event trapping gives you considerable control in handling events because you basically interrupt the default processing that Access would normally carry out in response to the user's keypress or mouseclick. If a user clicks a command button to save a form, whatever code you've written in the Click event of that command button will run. The event programming code is stored as a part of a form, report, or control and is triggered only when user interaction with a form or report generates a specific event; therefore, it cannot be used as a standalone procedure.



FIGURE 1.9. Event properties for an Access form are listed on the Event tab in the property sheet.

## Why Use Events?

Events allow you to make your applications dynamic and interactive. To handle a specific event, you need to select the appropriate event property on the property sheet and then write an event handling procedure. Access will provide its own default response to those events you have not programmed. Events cannot be defined for tables or queries.

## Walking Through an Event Procedure

The following hands-on exercise demonstrates how to write event procedures. Your task is to change the background color of a text box control on a form when the text box is selected and then return the default background color when you tab or click out of that text box.



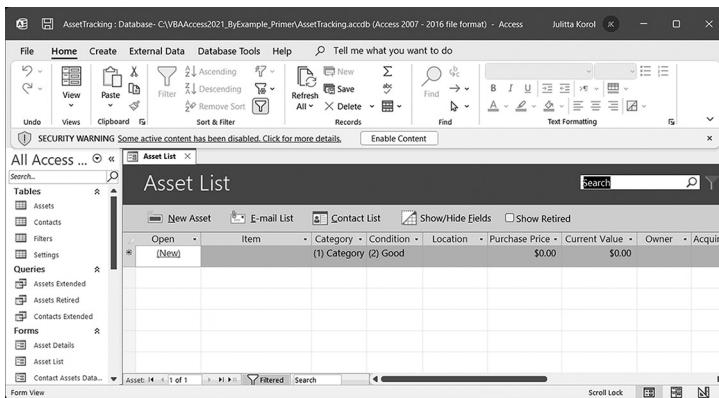
### Hands-On 1.3 Writing an Event Procedure

1. Close the **Chap01.accdb** database file used in Hands-On 1.1 and save changes to the file when prompted.
2. Copy the **AssetTracking.accdb** database from the companion files to your **C:\VBAAccess2021\_ByExample\_Primer** folder. This file is a copy of the Asset tracking database template provided by Microsoft.
3. Open the database **C:\VBAAccess2021\_ByExample\_Primer\AssetTracking.accdb**. Upon loading, when you see a Welcome screen, click the **Get Started** button.
4. Access opens the database and displays a security warning message (see Figure 1.10). To use the file, click the **Enable Content** button in the message bar. Access will close the database and reopen it. If you see the Welcome screen, click the Get Started button again.

#### **NOTE**

*The last section of this chapter explains how you can use trusted locations to keep Access from disabling the VBA code upon opening a database.*

*As Microsoft continues to improve security in Office, the default behavior and banners displayed in Access and other Office applications may be different from those presented in the instructions and images included in this book. For the most recent guidelines, please see <https://docs.microsoft.com/en-us/deployoffice/security/internet-macros-blocked>.*



**FIGURE 1.10.** Active content such as VBA Macros can contain viruses and other security hazards. By default, Access displays a Security Warning message when you first load a database file that contains active content. You should enable content only if you trust the contents of the file.

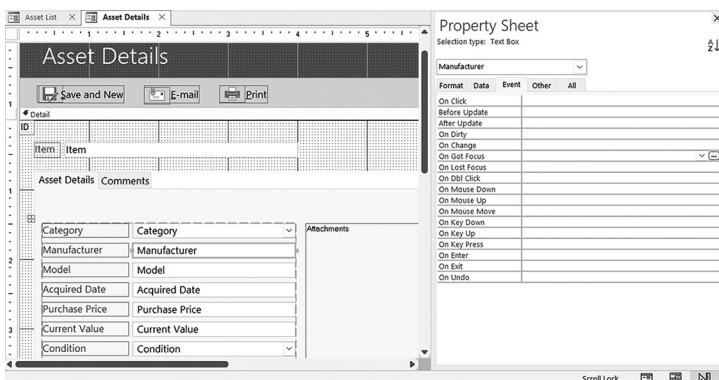
5. Open the **Asset Details** form in Design view. To do this, right-click the **Asset Details** form and choose **Design View** from the shortcut menu.

**NOTE**

*If the property sheet is not displayed next to the AssetDetails form, click the **Property Sheet** button in the **Tools** group of the Ribbon's **Form Design** tab.*

6. Click the **Manufacturer** text box control on the Asset Details form, and then click the **Event** tab in the property sheet. The property sheet will display **Manufacturer** in the control drop-down box.

The list of event procedures available for the text box control appears, as shown in Figure 1.11.



**FIGURE 1.11.** To create an event procedure for a form control, use the **Build** button, which is displayed as an ellipsis (...). This button is not available unless an event is selected.

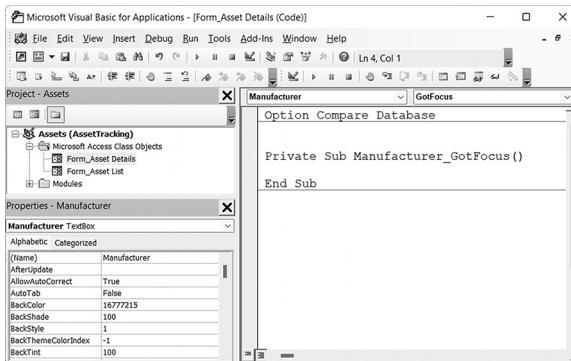
- Click in the column next to the **On Got Focus** event name, and then click the **Build** button (...), as shown in Figure 1.11 in the previous step. This will bring up the Choose Builder dialog box (see Figure 1.12).



**FIGURE 1.12.** To write VBA programming code for your event procedure, choose Code Builder in the Choose Builder dialog box.

- Select **Code Builder** in the Choose Builder dialog box and click **OK**. This brings up the VBE window that you've worked with using the Chap01.accdb database in earlier Hands-On exercises (see Figure 1.13).

Look at Figure 1.13. Access creates a skeleton of the GotFocus event procedure. The name of the event procedure consists of three parts: the object name (Manufacturer), an underscore character (\_), and the name of the event (GotFocus) occurring to that object. The word `Private` indicates that the event procedure cannot be triggered by an event from another form. The word `Sub` in the first line denotes the beginning of the event procedure. The words `End Sub` in the last line denote the end of the event procedure. The statements to be executed when the event occurs are written between these two lines.



**FIGURE 1.13.** Code Builder displays the event procedure Code window with a blank event procedure for the selected object. Here you can enter the code for Access to run when the specified GotFocus procedure is triggered.

Notice that the procedure name ends with a pair of empty parentheses ( ). Words such as `Sub`, `End`, or `Private` have special meaning to Visual Basic and are called *keywords* (reserved words). Visual Basic displays keywords in blue, but you can change the color of your keywords from the Editor Format tab in the Options dialog box (choose `Tools | Options` in the Visual Basic Editor window). All VBA keywords are automatically capitalized.

At the top of the Code window (see Figure 1.13), there are two drop-down listboxes. The one on the left is called Object. This box displays the currently selected control (Manufacturer). The box on the right is called Procedure. If you position the mouse over one of these boxes, the tooltip indicates the name of the box. Clicking on the down arrow at the right of the Procedure box displays a list of all possible event procedures associated with the object type selected in the Object box. You can close the drop-down listbox by clicking anywhere in the unused portion of the Code window.

9. To change the background color of a text box control to green, enter the following statement between the existing lines:

```
Me.Manufacturer.BackColor = RGB(0, 255, 0)
```

Notice that when you type each period, Visual Basic displays a list containing possible item choices. This feature, called List Properties/Methods, is a part of Visual Basic's on-the-fly syntax and programming assistance, and is covered in Chapter 2. When finished, your first event procedure should look as follows:

```
Private Sub Manufacturer_GotFocus()
    Me.Manufacturer.BackColor = RGB(0, 255, 0)
End Sub
```

The statement you just entered tells Visual Basic to change the background color of the Manufacturer text box to green when the cursor is moved into that control. The color is specified by using the `RGB` function. `Me` is the reference to the form that is currently active and contains the Manufacturer text box control.

#### RGB Colors

Color values are combinations of red, green, and blue components. The `RGB` function has the following syntax:

`RGB(red, green, blue)`

The intensity of red, green, and blue can range from 0 to 255. Here are some frequently used colors:

White	255, 255, 255	Dark Green	0, 128, 0
Black	0, 0, 0	Cyan	0, 255, 255
Gray	192, 192, 192	Dark Cyan	0, 128, 128
Red	255, 0, 0	Blue	0, 0, 255
Dark Red	128, 0, 0	Dark Blue	0, 0, 128
Yellow	255, 255, 0	Magenta	255, 0, 255
Dark Yellow	128, 128, 0	Dark Magenta	128, 0, 128
Green	0, 255, 0		

10. In the Visual Basic window, choose **File | Close and Return to Microsoft Access**. Notice that [Event Procedure] now appears next to the On Got Focus event property in the property sheet for the selected Manufacturer text box control (see Figure 1.14).

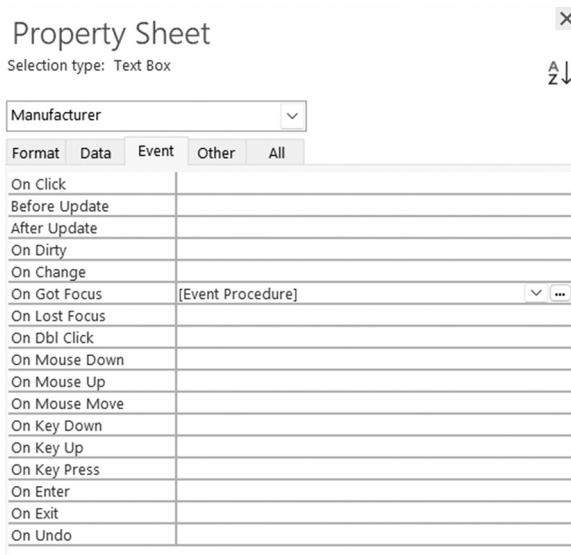


FIGURE 1.14. [Event Procedure] in the property sheet denotes that the text box's On Got Focus event has an event procedure associated with it.

11. To test your GotFocus event procedure, switch from the Design view of the Asset Details form to Form view by clicking the **View** button on the Ribbon's Design tab.
12. While in the Form view, click in the **Manufacturer** text box and notice the change in the background color.
13. Now, click on any other text box control on the Asset Details form.
14. Notice that the Manufacturer text box does not return to the original color. So far, you've told Visual Basic only what to do when the specified control receives the focus. If you want the background color to change when the focus moves to another control, there is one more event procedure to write—On Lost Focus.
15. To create the LostFocus procedure, return your form to Design view and click the **Manufacturer** control. In the property sheet for this control, select the **Event** tab, and then click the **Build** button to the right of the On Lost Focus event property. In the Choose Builder dialog box, select **Code Builder**.

16. To change the background color of a text box control to white, enter the following statement inside the Manufacturer\_LostFocus event procedure:

```
Me.Manufacturer.BackColor = RGB(255, 255, 255)
```

The completed On Lost Focus procedure is shown in Figure 1.15.

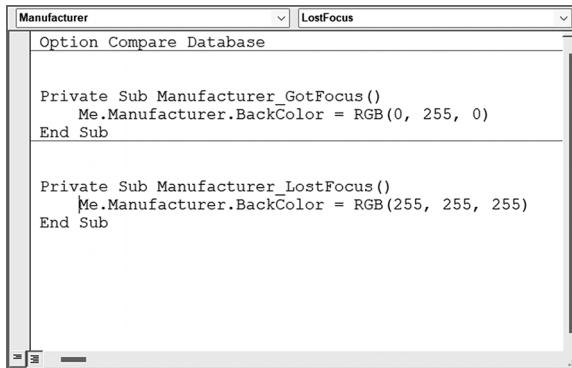


FIGURE 1.15. The GotFocus and LostFocus event procedures will now control the behavior of the Manufacturer control when the control is in focus and out of focus.

17. In the Visual Basic window, choose **File | Close and Return to Microsoft Access**. Notice that [Event Procedure] now appears next to the On Lost Focus event property in the property sheet for the selected Manufacturer text box control.
18. Repeat steps 11–12 to test both event procedures you have written.
19. When you are done, save the changes you made in the **Asset Tracking** database.

## **COMPIILING YOUR PROCEDURES**

The VBA code you write in the Visual Basic Editor Code window is automatically compiled by Access before you run it. The syntax of your VBA statements is first thoroughly checked for errors, and then your procedures are converted into executable format. If an error is discovered during the compilation process, Access stops compiling and displays an error message. It also highlights the line of code that contains the error. The compiling process can take from seconds to minutes or longer, depending on the number of procedures written and the number of modules used.

To ensure that your procedures have been compiled, you can explicitly compile them after you are done programming. You can do this by choosing **Debug | Compile** in the Visual Basic Editor window.

Access saves all the code in your database in its compiled form. Compiled code runs more quickly the next time you open it. You should always save your modules after you compile them. In Chapter 9, “Getting to Know Built-In Tools for Testing and Debugging,” you will learn how to test and troubleshoot your VBA procedures.

## PLACING A DATABASE IN A TRUSTED LOCATION

By default, the security features built into Access disable the VBA code and macros when you open a database. To make it easy to work with Access databases in this book, you will not want to bother with enabling content each time you open a database. To trust your databases permanently, you can place them in a *trusted location*—a folder on your local or network drive that you mark as trusted. You can get more information about the Enable Content button and access the Trust Center to set up a trusted folder by choosing File | Info (see Figure 1.16). This screen can also be activated by clicking the text message in the Security Warning message bar: “Some active content has been disabled. Click for more details.” See Figure 1.10 . Note that you will not see this message in the Info Section because you already told Access to trust the AssetTracking database by clicking on the Enable Content button. See the next Hands-On section on how to view this message.

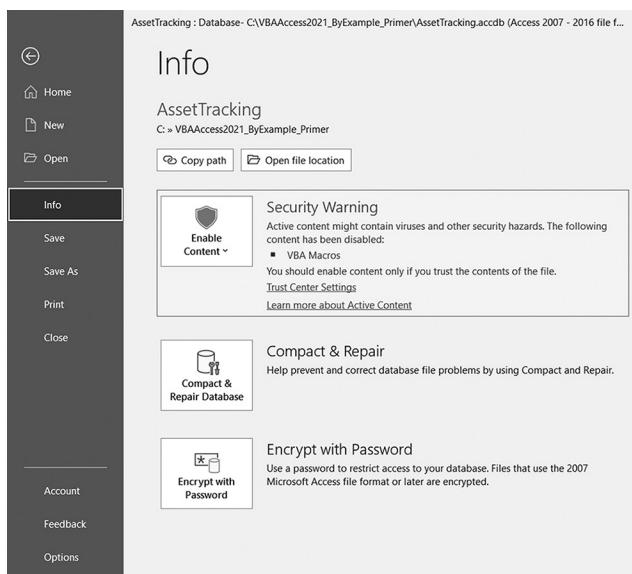


FIGURE 1.16. The Info tab with an explanation of the Security Warning message.

Hands-On 1.4 will take you through the process of setting up a trusted folder for your Access databases by using the Options button.

### ① Hands-On 1.4 Placing an Access Database in a Trusted Location

1. Close the AssetTracking database you worked with in the previous Hands-On and open the Chap01.accdb database. The database should load with the warning message. Do not click the Enable Content button.
2. Click the text of the warning message. Access will activate the Info section of the File tab, as shown in Figure 1.16.
3. In the same screen, click the **Options** button.
4. In the left pane of the Access Options dialog box, click **Trust Center**, and then click **Trust Center Settings** in the right pane, as shown in Figure 1.17.

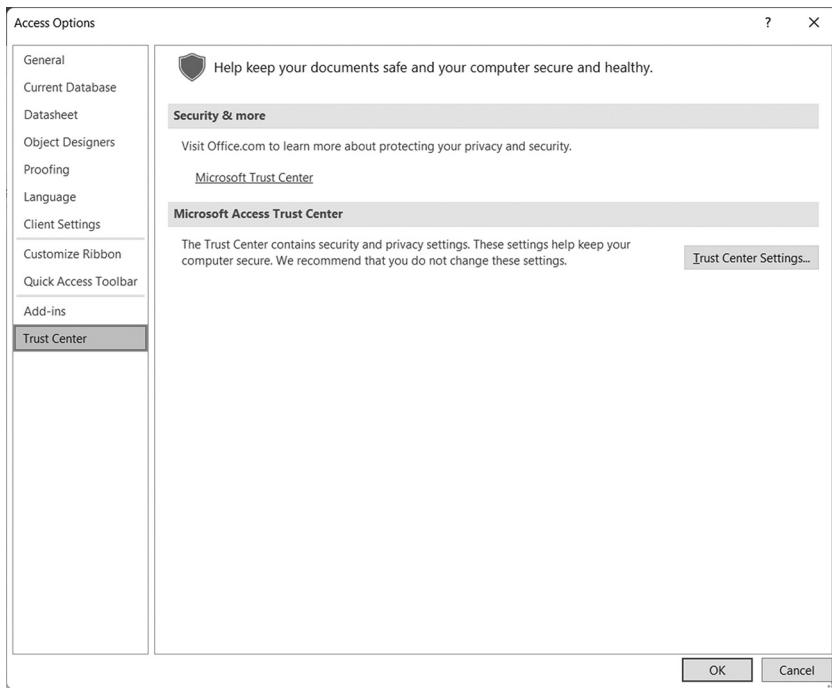


FIGURE 1.17. Working with the Trust Center (Step 1).

5. In the left pane of the Trust Center dialog box, click **Trusted Locations**, as shown in Figure 1.18, and click the **Add new location** button.

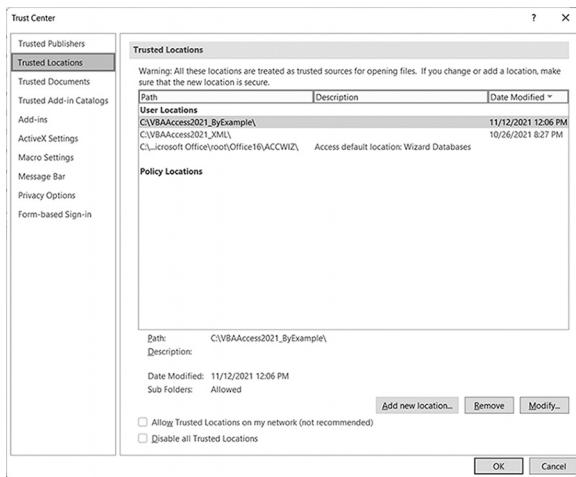


FIGURE 1.18. Working with the Trust Center (Step 2).

6. In the Path text box, type the path and folder name of the location on your local drive that you want to set up as a trusted source for opening files. Let's enter **C:\VBAAccess2021\_ByExample\_Primer** to designate this folder as a trusted location for Chapters 1-9 of this book's database programming exercises (see Figure 1.19). Fill in the Description as shown.

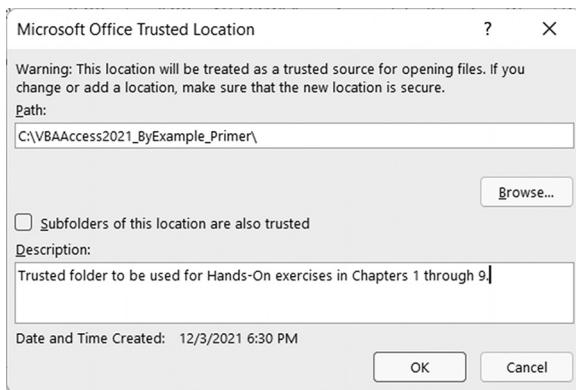


FIGURE 1.19. Working with the Trust Center (Step 3).

7. Click **OK** to close the Microsoft Office Trusted Location dialog box. The Trusted Locations list in the Trust Center dialog box now includes the **C:\VBAAccess2021\_ByExample\_Primer** folder as a trusted source (see Figure 1.20). Files put in a trusted location can be opened without being checked by the Trust Center security feature. On your own, create trusted folders for

the remaining chapters in this book. These folders are listed in Figure 1.20 as C:\VBAAccess2021\_ByExample and C:\VBAAccess2021\_XML.

8. Click **OK** to close the Trust Center dialog box.
9. Click **OK** to close the Access Options dialog box.
10. Close the **Chap01.accdb** database and exit Access.
11. Open the **Chap01.accdb** database file from your **C:\VBAPrimerAccess\_ByExample** folder and notice that Access no longer displays the Security Warning message.
12. Close the **Chap01.accdb** database.

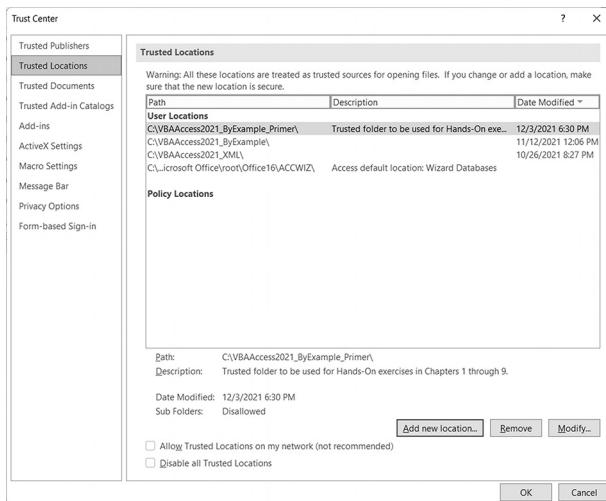


FIGURE 1.20. Working with the Trust Center (Step 4).

## SUMMARY

---

In this chapter, you learned about subroutine procedures, function procedures, property procedures, and event procedures. You also learned different ways of executing subroutines and functions. The main hands-on exercise in this chapter walked you through writing two event procedures in the Asset Details form's class module for a Manufacturer text control placed in the form. You finished this chapter by designating trusted location folders for your Access databases that you will create and work with in this book.

This chapter has given you a glimpse of the Microsoft Visual Basic programming environment built into Access. The next chapter will take you deeper into this interface, showing you various windows and shortcuts that you can use to program faster and with fewer errors.

# Chapter 2

# *GETTING TO KNOW VISUAL BASIC EDITOR (VBE)*

**N**ow that you know how to write procedures and functions in standard modules and event procedures in modules placed behind a form, we'll spend some time in the Visual Basic Editor window to become familiar with the multitude of tools it offers to simplify your programming tasks. With the tools located in the Visual Basic Editor window, you can:

- Write your own VBA procedures
- Create custom forms
- View and modify object properties
- Test and debug VBA procedures and locate errors

You can enter the VBA programming environment in either of the following ways:

- By selecting the Database Tools tab, and then Visual Basic in the Macro group
- From the keyboard, by pressing Alt+F11

## UNDERSTANDING THE PROJECT EXPLORER WINDOW

The Project Explorer window, located on the left side of the Visual Basic Editor window, provides access to modules behind forms and reports via the Microsoft Access Class Objects folder (see Figure 2.1). The Modules folder lists only standard modules that are not behind a form or report.

In addition to the Microsoft Access Class Objects and Modules folders, the VBA Project Explorer window can contain a Class Modules folder. Class modules are used for creating your own objects, as demonstrated in Chapter 8. Using the Project Explorer window, you can easily move between modules currently loaded into memory.

You can activate the Project Explorer window in one of three ways:

- From the View menu by selecting Project Explorer
- From the keyboard by pressing Ctrl-R
- From the Standard toolbar by clicking the Project Explorer button () as shown in Figure 2.2.



FIGURE 2.1. The Project Explorer window provides easy access to your VBA procedure code.

**NOTE**

If the Project Explorer window is visible but not active, activate it by clicking the Project Explorer title bar.

Buttons on the Standard toolbar (Figure 2.2) provide a quick way to access many Visual Basic features.



FIGURE 2.2. Use the toolbar buttons to quickly access frequently used features in the VBE window.

The Project Explorer window (see Figure 2.3) contains three buttons:

- **View Code**—Displays the Code window for the selected module.
- **View Object**—Displays the selected form or report in the Microsoft Access Class Objects folder. This button is disabled when an object in the Modules or Class Modules folder is selected.
- **Toggle Folders**—Hides and unhides the display of folders in the Project Explorer window.

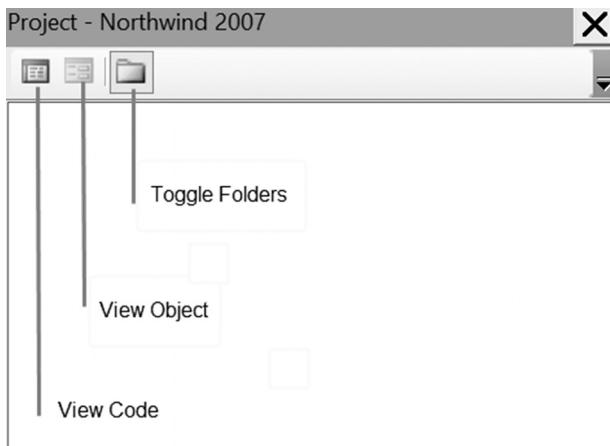


FIGURE 2.3. The VBE Project Explorer window contains three buttons that allow you to view code or objects and toggle folders.

## UNDERSTANDING THE PROPERTIES WINDOW

---

The Properties window allows you to review and set properties for the currently selected Access class or module. The name of the selected object is displayed in the Object box located just below the Properties window title bar. The Properties window displays the current settings for the selected object. Object properties can be viewed alphabetically or by category by clicking on the appropriate tab.

- **Alphabetic tab**—Lists all properties for the selected object alphabetically. You can change the property setting by selecting the property name, and then typing or selecting the new setting.
- **Categorized tab**—Lists all properties for the selected object by category. You can collapse the list so that you see only the category names, or you can expand a category to see the properties. The plus (+) icon to the left of the category name indicates that the category list can be expanded. The minus (-) indicates that the category is currently expanded.

The Properties window can be accessed in the following ways:

- From the View menu by selecting Properties Window
- From the keyboard by pressing F4
- From the Standard toolbar by clicking the Properties Window button () located to the right of the Project Explorer button

Figure 2.4 displays the properties of the E-mail Address text box control located in the Form\_Order Details form in the Northwind 2007 sample Access database. In order to access properties for a form control, you need to perform the steps outlined in Hands-On 2.1.

<b>NOTE</b>	<i>All code files and figures for the hands-on projects may be found in the companion files.</i>
-------------	--



### Hands-On 2.1 Using the Properties Window to View Control Properties

1. Copy the **Northwind 2007** sample database from the companion files to your **C:\VBAAccess2021\_ByExample\_Primer** folder.
2. Open and load the **C:\VBAAccess2021\_ByExample\_Primer\Northwind 2007.accdb** file. Log in to the database as **Andrew Cencini**.

3. When Northwind 2007 opens, press **Alt+F11** to activate the Visual Basic Editor window, or choose **Database Tools | Visual Basic**.
4. In the Project Explorer window, click the **Toggle Folders** button (□) and select the **Microsoft Access Class Objects** folder. Highlight the **Form\_Order Details** form (Figure 2.4) and click the **View Object** button (□) located to the left of the Toggle Folders button. This will open the selected form in Design view.
5. Press **Alt+F11** to return to the Visual Basic Editor. The Properties window will be filled with the properties for the Form\_Order Details form. To view the properties of the E-mail Address text box control on this form, as shown in Figure 2.4, select **E-mail Address** from the drop-down list located below the Properties window's title bar.

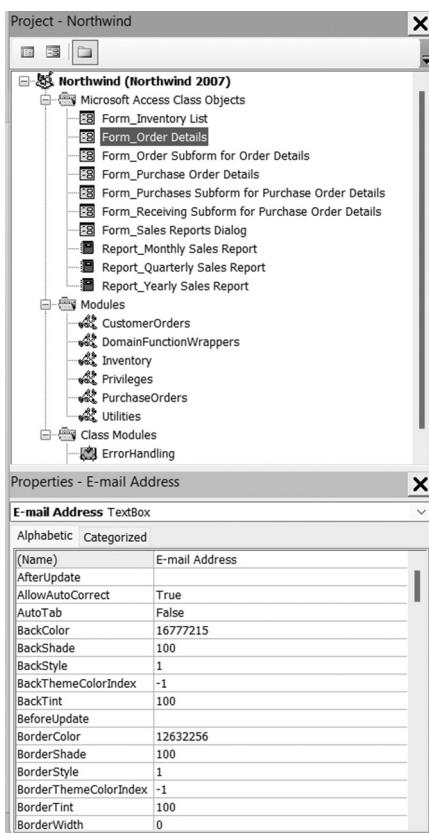


FIGURE 2.4. You can edit object properties in the Properties window, or you can edit them in the property sheet when a form or report is open in Design view.

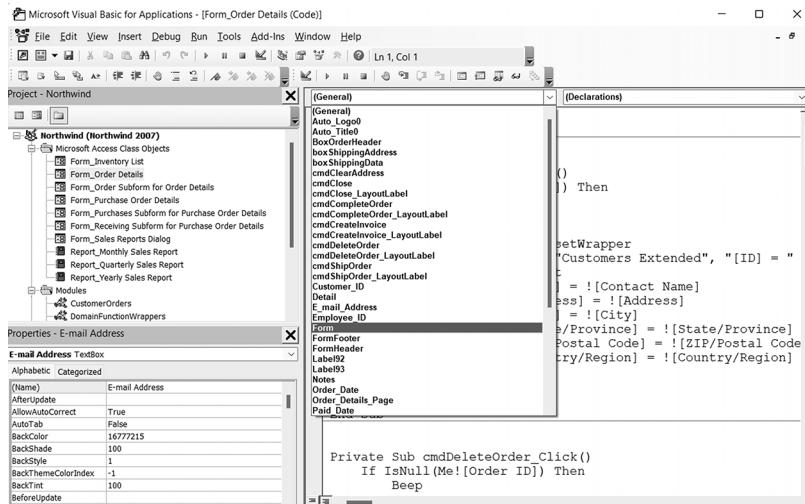
## UNDERSTANDING THE CODE WINDOW

The Code window is used for Visual Basic programming as well as for viewing and modifying the code of existing Visual Basic procedures. Each VBA module can be opened in a separate Code window.

There are several ways to activate the Code window:

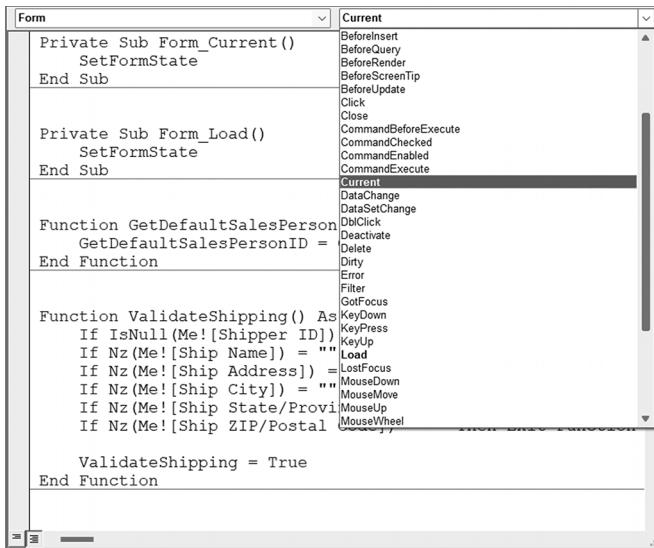
- From the Project Explorer window, choose the appropriate module and then click the View Code button (□)
- From the Microsoft Visual Basic menu bar, choose View | Code
- From the keyboard, press F7

At the top of the Code window there are two drop-down list boxes that allow you to move quickly within the Visual Basic code. In the Object box on the left side of the Code window, you can select the object whose code you want to view, as shown in Figure 2.5.



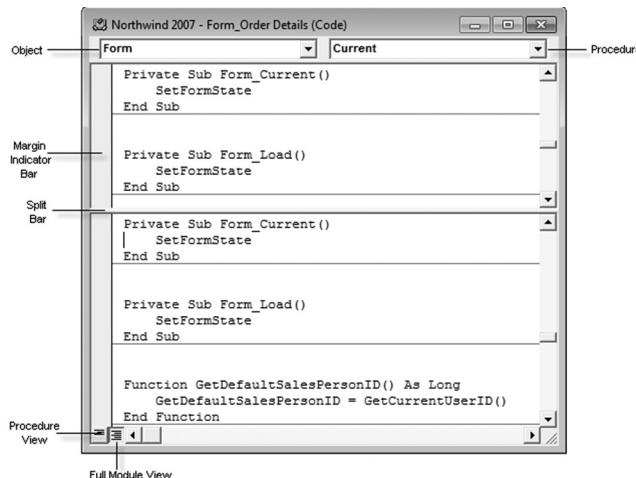
**FIGURE 2.5.** The Object drop-down box lists objects that are available in the module selected in the Project Explorer window.

The box on the right side of the Code window lets you select a procedure to view. When you click the down arrow at the right of this box, the names of all procedures located in a module are listed alphabetically, as shown in Figure 2.6. When you select a procedure in the Procedure box, the cursor will jump to the first line of that procedure.



**FIGURE 2.6.** The Procedure drop-down box lists events to which the object selected in the Object drop-down box can respond. If the selected module contains events written for the highlighted object, the names of these events appear in bold type.

By choosing Window | Split or dragging the split bar (located at the top of the vertical scroll bar) to a selected position in the Code window, you can divide the Code window into two panes, as shown in Figure 2.7.



**FIGURE 2.7.** By splitting the Code window, you can view different sections of a long procedure or a different procedure in each window pane.

Setting up the Code window for the two-pane display is useful for copying, cutting, and pasting sections of code between procedures in the same module. To return to a one-window display, drag the split bar all the way to the top of the Code window or choose Window | Split.

There are two icons at the bottom of the Code window (see Figure 2.7). The Procedure View icon changes the display to only one procedure at a time in the Code window. To select another procedure, use the Procedure drop-down box. The Full Module View icon changes the display to all the procedures in the selected module. Use the vertical scrollbar in the Code window to scroll through the module's code. The Margin Indicator bar is used by the Visual Basic Editor to display helpful indicators during editing and debugging.

## OTHER WINDOWS IN THE VBE

---

In addition to the Code window, there are several other windows that are frequently used in the Visual Basic environment, such as the Immediate, Locals, Watch, Project Explorer, Properties, and Object Browser windows. The Docking tab in the Options dialog box, shown in Figure 2.8, displays a list of available windows and allows you to choose which windows you want to be dockable. To access this dialog box, select Tools | Options in the Visual Basic Editor window.

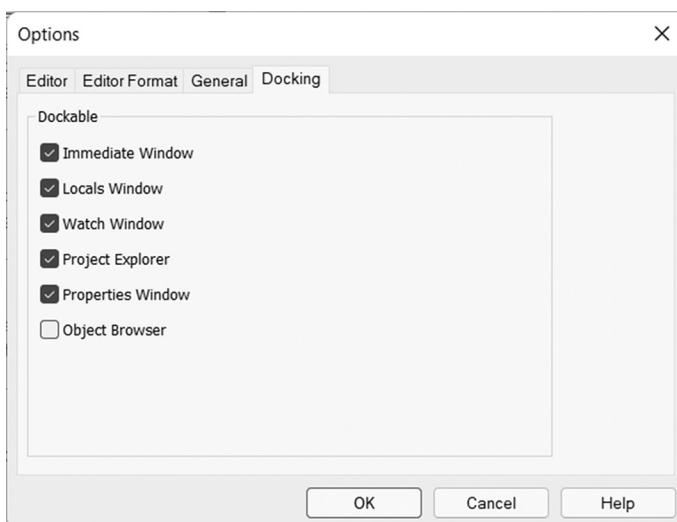


FIGURE 2.8. You can use the Docking tab in the Options dialog box to control which windows are currently displayed in the Visual Basic programming environment.

## ASSIGNING A NAME TO THE VBA PROJECT

A VBA Project is a set of Microsoft Access objects, modules, forms, and references.

When you create a Microsoft Access database and later switch to the VBE window, you will see in the Project Explorer window that Access had automatically assigned the database name to the VBA Project. For example, if your database is named Chap01.accdb, the Project Properties window displays Chap01 (Chap01) where the first “Chap01” denotes the VBA Project name and the “Chap01” in the parentheses is the name of the database. You can change the name of the VBA Project in one of the following ways:

- Choose Tools | <database name> Properties, enter a new name in the Project Name box of the Project Properties window (see Figure 2.9), and click OK.
- In the Project Explorer window, right-click the name of the project and select <database name> Properties. Enter a new name in the Project Name box of the Project Properties window (see Figure 2.9) and click OK.

To avoid naming conflicts between projects, make sure that you give your projects unique names.

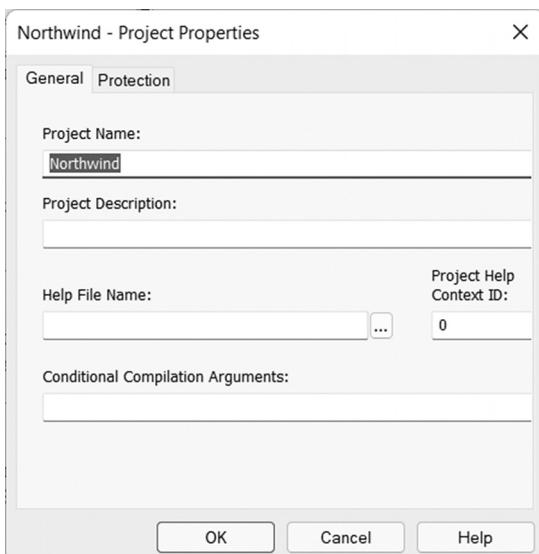


FIGURE 2.9. Use the Project Properties dialog box to rename the VBA Project.

## RENAMING A MODULE

---

When you insert a new module to your VBA Project, Access generates a default name for the module—Module1, Module2, and so on. You can rename your modules right after you insert them into the VBA project or when your project is being saved for the first time. In the latter case, Access will iterate through all the newly added (not saved) modules and will prompt you with the Save As dialog box to accept or change the module name. You can change the module name at any time via the Properties window. Simply select the module name (e.g., Module1) in the Project Explorer window and double-click the Name property in the Properties window. This action will highlight the default module name next to the Name property. Type the new name for the module and press Enter. The module name in the Project Explorer window should now reflect your change.

## SYNTAX AND PROGRAMMING ASSISTANCE

---

Writing procedures in Visual Basic requires that you use hundreds of built-in instructions and functions. Because most people cannot memorize the correct syntax of all the instructions available in VBA, the IntelliSense® technology provides you with syntax and programming assistance on demand while you are entering instructions. While working in the Code window, you can have special tools pop up and guide you through the process of creating correct VBA code. The Edit toolbar in the VBE window, shown in Figure 2.10, contains several buttons that let you enter correctly formatted VBA instructions with speed and ease. If the Edit toolbar isn't currently docked in the Visual Basic Editor window, you can turn it on by choosing View | Toolbars.

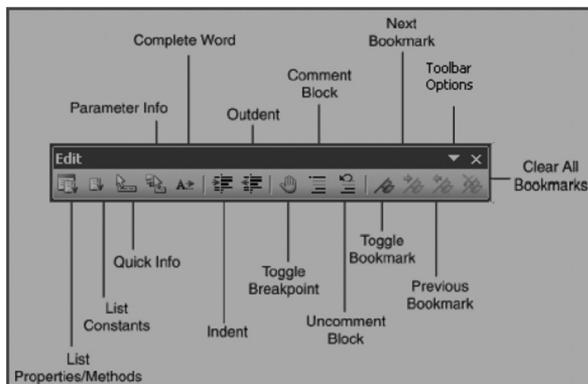


FIGURE 2.10. The Edit toolbar provides timesaving buttons while entering VBA code.

## List Properties/Methods

Each object can contain one or more properties and methods. When you enter the name of the object in the Code window followed by a period that separates the name of the object from its property or method, a pop-up menu may appear. This menu lists the properties and methods available for the object that precedes the period. To turn on this automated feature, choose Tools | Options. In the Options dialog box, click the Editor tab, and make sure the Auto List Members checkbox is selected. As you enter VBA instructions, Visual Basic suggests properties and methods that can be used with the object, as demonstrated in Figure 2.11.

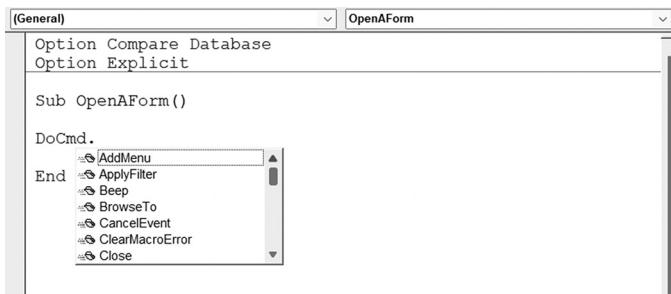


FIGURE 2.11. When Auto List Members is selected, Visual Basic suggests properties and methods that can be used with the object as you are entering the VBA instructions.

To choose an item from the pop-up menu, start typing the name of the property or method you want to use. When the correct item name is highlighted, press Enter to insert the item into your code and start a new line or press the Tab key to insert the item and continue writing instructions on the same line. You can also double-click the item to insert it in your code. To close the pop-up menu without inserting an item, press Esc. When you press Esc to remove the pop-up menu, Visual Basic will not display the menu for the same object again.

To display the Properties/Methods pop-up menu again, you can:

- Press Ctrl-J
- Use the Backspace key to delete the period, and then type the period again
- Right-click in the Code window, and select List Properties/Methods from the shortcut menu
- Choose Edit | List Properties/Methods
- Click the List Properties/Methods button ( ) on the Edit toolbar

## Parameter Info

Some VBA functions and methods can take one or more arguments (or parameters). If a Visual Basic function or method requires an argument, you can see the names of required and optional arguments in a tip box that appears just below the cursor as soon as you type the open parenthesis or enter a space. The Parameter Info feature (see Figure 2.12) makes it easy for you to supply correct arguments to a VBA function or method. In addition, it reminds you of two other things that are very important for the function or method to work correctly: the order of the arguments and the required data type of each argument. For example, if you enter in the Code window the instruction `DoCmd.` `OpenForm` and type a space after the `OpenForm` method, a tip box appears just below the cursor. Then as soon as you supply the first argument and enter the comma, Visual Basic displays the next argument in bold. Optional arguments are surrounded by square brackets [ ]. To close the Parameter Info window, all you need to do is press Esc.



FIGURE 2.12. A tip window displays a list of arguments used by a VBA function or method.

To open the tip box using the keyboard, enter the instruction or function, followed by the open parenthesis, and then press Ctrl-Shift-I. You can also click the Parameter Info button (🔍) on the Edit toolbar or choose Edit | Parameter Info from the menu bar.

You can also display the Parameter Info box when entering a VBA function. To try this out quickly, choose View | Immediate Window, and then type the following in the Immediate window:

```
Mkdir(
```

You should see the `MkDir(Path As String)` tip box just below the cursor. Now, type "C:\NewFolder" followed by the ending parenthesis. When you press Enter, Visual Basic will create a folder named NewFolder in the root directory of your computer. Activate File Explorer and check it out!

## List Constants

If there is a check mark next to the Auto List Members setting in the Options dialog box (Editor tab), Visual Basic displays a pop-up menu listing the constants that are valid for the property or method. A *constant* is a value that indicates a specific state or result. Access and other members of the Microsoft 365 have a number of predefined, built-in constants.

Suppose you want to open a form in Design view. In Microsoft Access, a form can be viewed in Design view (acDesign), Datasheet view (acFormDS), PivotChart view (acFormPivotChart), PivotTable view (acFormPivotTable), Form view (acNormal), and Print Preview (acPreview). Each of these options is represented by a built-in constant. Microsoft Access constant names begin with the letters “ac.” As soon as you enter a comma and a space following your instruction in the Code window (e.g., `DoCmd.OpenForm "Products"`), a pop-up menu will appear with the names of valid constants for the `OpenForm` method, as shown in Figure 2.13.

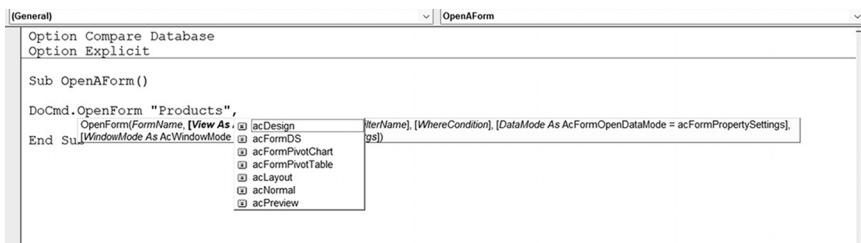


FIGURE 2.13. The List Constants pop-up menu displays a list of constants that are valid for the property or method typed.

The List Constants menu can be activated by pressing **Ctrl+Shift+J** or by clicking the List Constants button (⬇) on the Edit toolbar.

## Quick Info

When you select an instruction, function, method, procedure name, or constant in the Code window and then click the Quick Info button ( ⓘ ) on the Edit toolbar (or press **Ctrl+I**), Visual Basic will display the syntax of the highlighted item as well as the value of its constant (see Figure 2.14). The Quick Info feature can be turned on or off using the Options dialog box (Tools | Options). To use the feature, click the Editor tab in the Options dialog box, and make sure there is a check mark in the box next to Auto Quick Info.

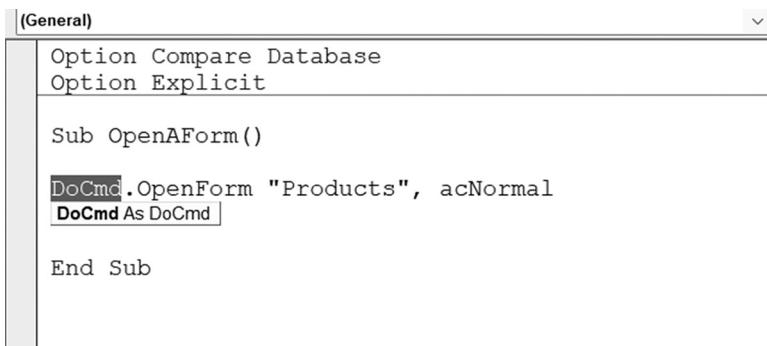


FIGURE 2.14. The Quick Info feature provides a list of function parameters, as well as constant values and VBA statement syntax.

## Complete Word

Another way to increase the speed of writing VBA procedures in the Code window is with the Complete Word feature. As you enter the first few letters of a keyword and click the Complete Word button ( A small button with a double arrow symbol) on the Edit toolbar, Visual Basic will complete the keyword entry for you. For example, if you enter the first three letters of the keyword `DoCmd` (DoC) in the Code window, and then click the Complete Word button on the Edit toolbar, Visual Basic will complete the rest of the command. In the place of `DoC` you will see the entire instruction, `DoCmd`.

If there are several VBA keywords that begin with the same letters, when you click the Complete Word button on the Edit toolbar, Visual Basic will display a pop-up menu listing all of them. To try this, enter only the first three letters of the word Application (App), and then press the Complete Word button on the toolbar. You can then select the appropriate word from the pop-up menu.

## Indent/Outdent

The Editor tab in the Options dialog box, shown in Figure 2.15, contains many settings you can enable to make automated features available in the Code window.

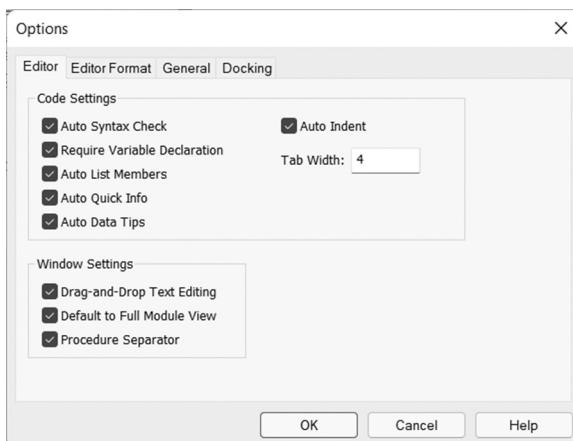


FIGURE 2.15. The Options dialog box lists features you can turn on and off to fit the VBA programming environment to your needs.

When the Auto Indent option is turned on, Visual Basic automatically indents the selected lines of code using the Tab Width value. The default entry for Auto Indent is four characters (see Figure 2.15). You can easily change the tab width by typing a new value in the text box. Why would you want to use indentation in your code? Indentation makes your VBA procedures more readable and easier to understand. Indenting is especially recommended for entering lines of code that make decisions or repeat actions.

Let's see how you can indent and outdent lines of code using the Form\_InventoryList form in the Northwind database that you opened in the previous hands-on exercise.



### Hands-On 2.2 Using the Indent/Outdent Feature

1. In the Project Explorer window in the Microsoft Access Class Objects folder, double-click **Form\_Inventory List**. The Code window should now show the CmdPurchase\_Click event procedure written for this form.
2. In the Code window, select the block of code beginning with the keyword **If** and ending with the keywords **End If**.

3. Click the **Indent** button ( ) on the Edit toolbar or press **Tab** on the keyboard. The selected block of code will move four spaces to the right. You can adjust the number of spaces to indent by choosing **Tools | Options** and entering the appropriate value in the Tab Width box on the Editor tab.
4. Now, click the **Outdent** button ( ) on the Edit toolbar or press **Shift+Tab** to return the selected lines of code to the previous location in the Code window. The Indent and Outdent options are also available from Visual Basic Editor's Edit menu.

### **Comment Block/Uncomment Block**

---

The apostrophe placed at the beginning of a line of code denotes a comment. Besides the fact that comments make it easier to understand what the procedure does, comments are also very useful in testing and troubleshooting VBA procedures. For example, when you execute a procedure, it may not run as expected. Instead of deleting the lines of code that may be responsible for the problems encountered, you may want to skip the lines for now and return to them later. By placing an apostrophe at the beginning of the line you want to avoid, you can continue checking the other parts of your procedure. While commenting one line of code by typing an apostrophe works fine for most people, when it comes to turning entire blocks of code into comments, you'll find the Comment Block and Uncomment Block buttons on the Edit toolbar very handy and easy to use. To comment a few lines of code, select the lines and click the Comment Block button ( ). To turn the commented code back into VBA instructions, click the Uncomment Block button ( ). If you click the Comment Block button without selecting a block of text, the apostrophe is added only to the line of code where the cursor is currently located.

## **USING THE OBJECT BROWSER**

---

If you want to move easily through the myriad of VBA elements and features, examine the capabilities of the Object Browser. This special built-in tool is available in the Visual Basic Editor window.

To access the Object Browser, use any of the following methods:

- Press F2
- Choose **View | Object Browser**
- Click the Object Browser button ( ) on the toolbar

The Object Browser allows you to browse through the objects available to your VBA procedures, as well as view their properties, methods, and events. With the aid of the Object Browser, you can quickly move between procedures in your database application and search for objects and methods across various type libraries.

The Object Browser window, shown in Figure 2.16, is divided into several sections. The top of the window displays the Project/Library drop-down listbox with the names of all currently available libraries and projects.

A *library* is a special file that contains information about the objects in an application. New libraries can be added via the References dialog box (select Tools | References). The entry for <All Libraries> lists the objects of all libraries installed on your computer. While the Access library contains objects specific to using Microsoft Access, the VBA library provides access to three objects (Debug, Err, and Collection), as well as several built-in functions and constants that give you flexibility in programming. You can send output to the Immediate window, get information about runtime errors, work with the Collection object, manage files, deal with text strings, convert data types, set date and time, and perform mathematical operations.

Below the Project/Library drop-down listbox is a search box (Search Text) that allows you to quickly find information in a library. This field remembers the last four items you searched for. To find only whole words, right-click anywhere in the Object Browser window, and then choose Find Whole Word Only from the shortcut menu. The Search Results section of the Object Browser displays the Library, Class, and Member elements that meet the criteria entered in the Search Text box. When you type the search text and click the Search button, Visual Basic expands the Object Browser window to show the search results. You can hide or show the Search Results section by clicking the button located to the right of the binoculars. In the lower section of the Object Browser window, the Classes listbox displays the available object classes in the selected library. If you select the name of the open database (e.g., Northwind) in the Project/Library listbox, the Classes list will display the objects as listed in the Explorer window.

In Figure 2.16, the Form\_Inventory List object class is selected. When you highlight a class, the list on the right side (Members) shows the properties, methods, and events available for that class. By default, members are listed alphabetically. You can, however, organize the Members list by group type (properties, methods, or events) using the Group Members command from the Object Browser shortcut menu (right-click anywhere in the Object Browser window to display this menu).

When you select the Northwind 2007 project in the Project/Library listbox, the Members listbox will list all the procedures available in this project. To examine a procedure's code, double-click its name. When you select a VBA library in the Project/Library listbox, you will see the Visual Basic built-in functions and constants. If you need more information on the selected class or member, click the question mark button located at the top of the Object Browser window. The bottom of the Object Browser window displays a code template area with the definition of the selected member. Clicking the green hyperlink text in the code template lets you jump to the selected member's class or library in the Object Browser window. Text displayed in the code template area can be copied and pasted to a Code window. If the Code window is visible while the Object Browser window is open, you can save time by dragging the highlighted code template and dropping it into the Code window. You can easily adjust the size of the various sections of the Object Browser window by dragging the dividing horizontal and vertical lines.

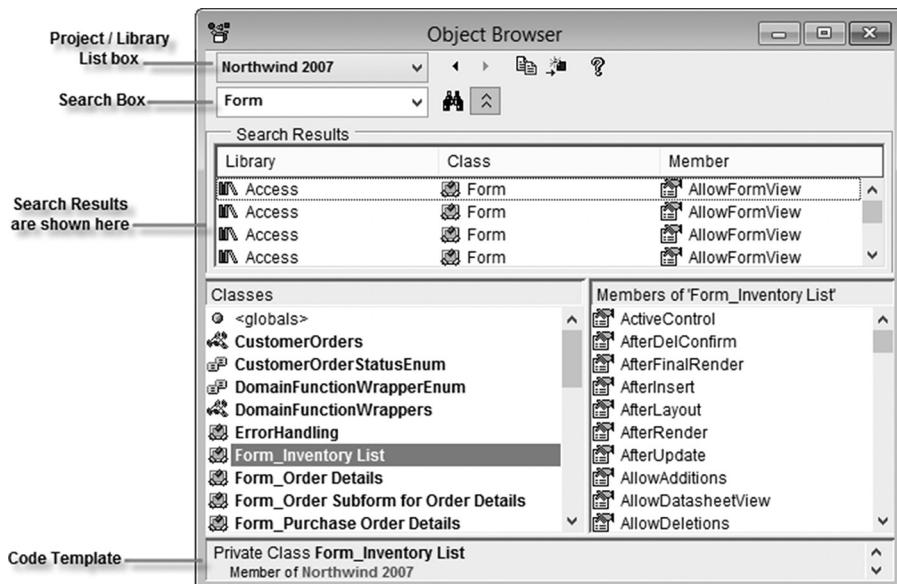


FIGURE 2.16. The Object Browser window allows you to browse through all the objects, properties, and methods available to the current VBA project.

Let's put the Object Browser to use in VBA programming. Assume that you want to write a VBA procedure to control a checkbox placed on a form and would like to see the list of properties and methods that are available for working with checkboxes.



### Hands-On 2.3 Using the Object Browser

1. In the Visual Basic Editor window, press **F2** to display the Object Browser.
2. In the Project/Library listbox (see Figure 2.16), click the drop-down arrow and select the **Access** library.
3. Type **checkbox** in the Search Text box and click the **Search** button (🔍). Make sure you don't enter a space in the search string.

Visual Basic begins to search the Access library and displays the search results. By analyzing the search results in the Object Browser window, you can find the appropriate VBA instructions for writing your VBA procedures. For example, looking at the Members list lets you quickly determine that you can enable or disable a checkbox by setting the **Enabled** property. To get detailed information on any item found in the Object Browser, select the item and press **F1** to activate online help.

## USING THE VBA OBJECT LIBRARY

---

While programming in Microsoft Access you will need to rely on some functions that are general in nature. Functions that are available in the VBA Objects Library will allow you to manage files and folders, set the date and time, interact with users, convert data types, deal with text strings, or perform mathematical calculations. In the following exercise, you will see how to use one of these functions to create a new subfolder without leaving Access.



### Hands-On 2.4 Using Built-In VBA Functions

1. In the Visual Basic Editor window with the Northwind 2007 database open, choose **Insert | Module** to create a new standard module.
2. In the Properties Window, change the Name property of Module1 to **VBA\_Chap2**.
3. In the Code window, enter **Sub NewFolder()** as the name of the procedure and press **Enter**. Visual Basic will enter the ending keywords: **End Sub**.
4. Press **F2** to display the Object Browser.
5. Click the drop-down arrow in the Project/Library listbox and select **VBA**.
6. Enter **file** in the Search Text box and press **Enter**.
7. Scroll down in the Members listbox and highlight the **MkDir** method.
8. Click the **Copy** button in the Object Browser window to copy the selected method name to the Windows clipboard.

9. Close the Object Browser and return to the Code window. Paste the copied instruction inside the **NewFolder** procedure.
10. Now, enter a space, followed by “**C:\Study**”. Be sure to enter the name of the entire path and the quotation marks. Your NewFolder procedure should look like the following:

```
Sub NewFolder()
    MkDir "C:\Study"
End Sub
```

11. Choose **Run | Run Sub/UserForm** to run the NewFolder procedure.

After you run the NewFolder procedure, Visual Basic creates a new folder on drive C called Study. To see the folder, activate File Explorer. After creating a new folder, you may realize that you don’t need it after all. Although you could easily delete the folder while in File Explorer, how about getting rid of it programmatically?

The Object Browser contains many other methods that are useful for working with folders and files. The **RmDir** method is just as simple to use as the **MkDir** method. To remove the Study folder from your hard drive, replace the **MkDir** method with the **RmDir** method and rerun the NewFolder procedure. Or create a new procedure called RemoveFolder, as shown here:

```
Sub RemoveFolder()
    RmDir "C:\Study"
End Sub
```

When writing procedures from scratch, it’s a good idea to consult the Object Browser for names of the built-in VBA functions.

## USING THE IMMEDIATE WINDOW

---

The Immediate window is a sort of VBA programmer’s scratch pad. Here you can test VBA instructions before putting them to work in your VBA procedures. It is a great tool for experimenting with your new language. Use it to try out your statements. If the statement produces the expected result, you can copy the statement from the Immediate window into your procedure (or you can drag it right onto the Code window if the window is visible).

To activate the Immediate window, choose **View | Immediate Window** in the Visual Basic Editor, or press **Ctrl+G** while in the Visual Basic Editor window.

The Immediate window can be moved anywhere on the Visual Basic Editor window, or it can be docked so that it always appears in the same area of the

screen. The docking setting can be turned on and off from the Docking tab in the Options dialog box (Tools | Options).

To close the Immediate window, click the Close button in the top-right corner of the window.

The following hands-on exercise demonstrates how to use the Immediate window to check instructions and get answers.



### Hands-On 2.5 Experiments in the Immediate Window

1. If you are not in the VBE window, press **Alt+F11** to activate it.
2. Press **Ctrl+G** to activate the Immediate window or choose **View | Immediate Window**.
3. In the Immediate window, type the following instruction and press **Enter**:  
`DoCmd.OpenForm "Inventory List"`
4. If you entered the preceding VBA statement correctly, Visual Basic opens the Inventory List form, assuming the Northwind database is open.
5. Enter the following instruction in the Immediate window:

```
Debug.Print Forms! [Inventory List].RecordSource
```

When you press **Enter**, Visual Basic indicates that `Inventory` is the RecordSource for the Inventory List form. Every time you type an instruction in the Immediate window and press Enter, Visual Basic executes the statement on the line where the insertion point is located. If you want to execute the same instruction again, click anywhere in the line containing the instruction and press Enter. For more practice, rerun the statements shown in Figure 2.17. Start from the instruction displayed in the first line of the Immediate window. Execute the instructions one by one by clicking in the appropriate line and pressing Enter.

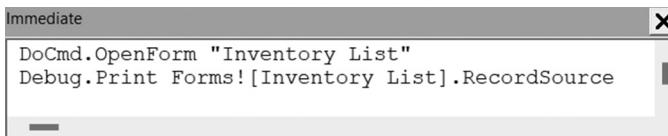


FIGURE 2.17. Use the Immediate window to evaluate and try Visual Basic statements.

So far you have used the Immediate window to perform some actions. The Immediate window also allows you to ask questions. Suppose you want to find out the answers to “How many controls are in the Inventory List form?” or

“What’s the name of the current application?” When working in the Immediate window, you can easily get answers to these and other questions.

In the preceding exercise, you entered two instructions. Let’s return to the Immediate window to ask some questions. Access remembers the instructions entered in the Immediate window even after you close this window. The contents of the Immediate window are automatically deleted when you exit Microsoft Access. You can also clear the Immediate Window at any time by pressing Ctrl+A followed by the Delete key.



### Hands-On 2.6 Asking Questions in the Immediate Window

1. Click in a new line of the Immediate window and enter the following statement to find out the number of controls in the Inventory List form:

```
?Forms! [Inventory List].Controls.Count
```

When you press **Enter**, Visual Basic enters the number of controls on a new line in the Immediate window.

2. Click in a new line of the Immediate window, and enter the following statement:

```
?Application.Name
```

When you press **Enter**, Visual Basic enters the name of the active application on a new line in the Immediate window.

3. In a new line in the Immediate window, enter the following instruction:

```
?12/3
```

When you press **Enter**, Visual Basic shows the result of the division on a new line. But what if you want to know the result of  $3 + 2$  and  $12 * 8$  right away? Instead of entering these instructions on separate lines, you can enter them on one line as in the following example:

```
?3+2 : ?12*8
```

Notice the colon separating the two blocks of instructions. When you press the **Enter** key, Visual Basic displays the results 5 and 96 on separate lines in the Immediate window.

Here are a couple of other statements you may want to try out on your own in the Immediate window:

```
?Application.GetOption("Default Database Directory")  
?Application.CodeProject.Name
```

Instead of using the question mark, you may precede the statement typed in the Immediate window with the Print command, like this:

```
Print Application.CodeProject.Name
```

To delete the instructions from the Immediate window, highlight all the lines and press **Delete**.

4. In the Visual Basic Editor window, choose **File | Close and Return to Microsoft Access**.
5. Close the **Northwind 2007.accdb** database.

**NOTE**

*Recall that in Chapter 1 you learned how to run subroutine procedures and functions from the Immediate window. You will find other examples of running procedures and functions from this window in subsequent chapters.*

## SUMMARY

---

Programming in Access requires a working knowledge of objects and collections of objects. In this chapter, you explored features of the Visual Basic Editor window that can assist you in writing VBA code. Here are some important points:

- When in doubt about objects, properties, or methods in an existing VBA procedure, highlight the instruction in question and fire up the online help by pressing F1.
- When you need on-the-fly programming assistance while typing your VBA code, use the shortcut keys or buttons available on the Edit toolbar.
- If you need a quick listing of properties and methods for every available object, or have trouble locating a hard-to-find procedure, go with the Object Browser.
- If you want to experiment with VBA and see the results of VBA commands immediately, use the Immediate window.

In the next chapter, you will learn how you can remember certain values in your VBA procedures by using various types of variables and constants.



# Chapter 3

# ACCESS VBA FUNDAMENTALS

In Chapter 2, you used the question mark to have Visual Basic return some information in the Immediate window. Unfortunately, when you write Visual Basic procedures outside the Immediate window, you can't use the question mark. So how do you obtain answers to your questions in VBA procedures? To find out what a VBA instruction (statement) has returned, you must tell Visual Basic to memorize it. This is done by using variables. This chapter introduces you to many types of variables, data types, and constants that you can and should use in your VBA procedures.

## INTRODUCTION TO DATA TYPES

---

When you create Visual Basic procedures, you have a purpose in mind: You want to manipulate data. Because your procedures will handle different kinds of information, you should understand how Visual Basic stores data.

The *data type* determines how the data is stored in the computer's memory. For example, data can be stored as a number, text, date, object, etc. If you forget to tell Visual Basic the data type, it is assigned the Variant data type. The *Variant* type can figure out on its own what kind of data is being manipulated and then take on that type. The Visual Basic data types are shown in Table 3.1. In addition to the built-in data types, you can define your own data types; these are

known as *user-defined data types*. Because data types take up different amounts of space in the computer's memory, some of them are more expensive than others. Therefore, to conserve memory and make your procedure run faster, you should select the data type that uses the fewest bytes but at the same time can handle the data that your procedure has to manipulate.

TABLE 3.1 VBA data types.

Data Type	Storage Size	Range
Byte	1 byte	A number in the range of 0 to 255.
Boolean	2 bytes	Stores a value of True (0) or False (-1).
Integer	2 bytes	A number in the range of -32,768 to 32,767. The type declaration character for Integer is the percent sign (%).
Long (long integer)	4 bytes	A number in the range of -2,147,483,648 to 2,147,483,647. The type declaration character for Long is the ampersand (&).
LongLong	8 bytes	Stored as a signed 64-bit (8-byte) number ranging in value from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The type declaration character for LongLong is the caret (^). LongLong is a valid declared type only on 64-bit platforms.
LongPtr (Long integer on 32-bit systems; LongLong integer on 64-bit systems)	4 bytes on 32-bit; 8 bytes on 64-bit	Numbers ranging in value from -2,147,483,648 to 2,147,483,647 on 32-bit systems; -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 on 64-bit systems. Using LongPtr enables writing code that can run in both 32-bit and 64-bit environments.
Single (single-precision floating-point)	4 bytes	Single-precision floating-point real number ranging in value from -3.402823E38 to -1.401298E-45 for negative values and from 1.401298E-45 to 3.402823E38 for positive values. The type declaration character for Single is the exclamation point (!).
Double (double-precision floating-point)	8 bytes	Double-precision floating-point real number in the range of -1.79769313486231E308 to -4.94065645841247E-324 for negative values and 4.94065645841247E-324 to 1.79769313486231E308 for positive values. The type declaration character for Double is the number sign (#).
Currency (scaled integer)	8 bytes	Monetary values used in fixed-point calculations: -922,337,203,685,477.5808 to 922,337,203,685,477.5807. The type declaration character for Currency is the at sign (@).

Data Type	Storage Size	Range
Decimal	14 bytes	<p>96-bit (12-byte) signed integer scaled by a variable power of 10. The power of 10 scaling factor specifies the number of digits to the right of the decimal point, and ranges from 0 to 28. With no decimal point (scale of 0), the largest value is <math>+/-79,228,162,514,264,337,593,543,950,335</math>. With 28 decimal places, the largest value is <math>+/-7.9228162514264337593543950335</math>. The smallest nonzero value is <math>+/-0.00000000000000000000000000000001</math>.</p> <p>You cannot declare a variable to be of type Decimal. You must use the Variant data type. Use the CDec function to convert a value to a decimal number:</p> <pre>Dim numDecimal As Variant numDecimal = CDec(0.02 * 15.75 * 0.0006)</pre>
Date	8 bytes	Date from January 1, 100, to December 31, 9999, and times from 0:00:00 to 23:59:59. Date literals must be enclosed within number signs (#); for example: #January 1, 2011#
Object	4 bytes	<p>Any Object reference.</p> <p>Use the Set statement to declare a variable as an Object.</p>
String (variable-length)	10 bytes + string length	<p>A variable-length string can contain up to approximately 2 billion characters.</p> <p>The type declaration character for String is the dollar sign (\$).</p>
String (fixed-length)	Length of string	A fixed-length string can contain 1 to approximately 65,400 characters.
Variant (with numbers)	16 bytes	Any numeric value up to the range of a Double.
Variant (with characters)	22 bytes + string length	Any valid nonnumeric data type in the same range as for a variable-length string.
User-defined (using Type)	One or more elements	<p>A data type you define using the Type statement. User-defined data types can contain one or more elements of a data type, an array, or a previously defined user-defined type. For example:</p> <pre>Type custInfo     custFullName as String     custTitle as String     custBusinessName as String     custFirstOrderDate as Date End Type</pre>

In addition to the above data types, Access 2021 introduces a new Date/Time Extended data type. This data type provides a larger date range, higher fractional precision, and compatibility with the SQL Server datetime2 data type. This data

type is not compatible with previous versions of Microsoft Access. The versions of Access that do not include this feature will not be able to open the database. For more information on using the Date/Time Extended data type, see the following Microsoft Support link:

<https://support.microsoft.com/en-us/office/using-the-date-time-extended-data-type-708c32da-a052-4cc2-9850-9851042e0024#>

## UNDERSTANDING AND USING VARIABLES

---

A *variable* is a name used to refer to an item of data. Each time you want to remember the result of a VBA instruction, think of a name that will represent it. For example, if you want to keep track of the number of controls on a form, you can make up a name such as NumOfControls, TotalControls, or FormsControlCount.

The names of variables can contain characters, numbers, and punctuation marks except for the following:

, # \$ % & @ !

The name of a variable cannot begin with a number or contain a space. If you want the name of the variable to include more than one word, use the underscore (\_) as a separator. Although a variable name can contain as many as 254 characters, it's best to use short and simple names. Using short names will save you typing time when you need to reuse the variable in your Visual Basic procedure. Visual Basic doesn't care whether you use uppercase or lowercase letters in variable names; however, most programmers use lowercase letters. When the variable name is composed of more than one word, most programmers capitalize the first letter of each word, as in the following: NumOfControls, First\_Name.

---

### SIDE BAR *Reserved Words Can't Be Used for Variable Names*

You can use any label you want for a variable name except for the reserved words that VBA uses. Visual Basic function names and words that have a special meaning in VBA cannot be used as variable names. For example, words such as Name, Len, Empty, Local, Currency, or Exit will generate an error message if used as a variable name.

---

Give your variables names that can help you remember their roles. Some programmers use a prefix to identify the variable's type. A variable name preceded

with “str,” such as strName, can be quickly recognized within the procedure code as the variable holding the text string.

## Declaring Variables

---

You can create a variable by declaring it with a special command or by just using it in a statement. When you declare your variable, you make Visual Basic aware of the variable’s name and data type. This is called *explicit variable declaration*.

### SIDE BAR *Advantages of Explicit Variable Declaration*

---

Explicit variable declaration:

- Speeds up the execution of your procedure. Since Visual Basic knows the data type, it reserves only as much memory as is necessary to store the data.
- Makes your code easier to read and understand because all the variables are listed at the very beginning of the procedure.
- Helps prevent errors caused by misspelling a variable name. Visual Basic automatically corrects the variable name based on the spelling used in the variable declaration.

If you don’t let Visual Basic know about the variable prior to using it, you are implicitly telling VBA that you want to create this variable. *Implicit variables* are automatically assigned the Variant data type (see Table 3.1 earlier in the chapter). Although implicit variable declaration is convenient (it allows you to create variables on the fly and assign values to them without knowing in advance the data type of the values being assigned), it can cause several problems.

### SIDE BAR *Disadvantages of Implicit Variable Declaration*

---

- If you misspell a variable name in your procedure, Visual Basic may display a runtime error or create a new variable. You are guaranteed to waste some time troubleshooting problems that could easily have been avoided had you declared your variable at the beginning of the procedure.
- Since Visual Basic does not know what type of data your variable will store, it assigns it a Variant data type. This causes your procedure to run slower because Visual Basic must check the data type every time it deals with your variable. And because Variant variables can store any type of data, Visual Basic must reserve more memory to store your data.

You declare a variable with the `Dim` keyword. `Dim` stands for “dimension.” The `Dim` keyword is followed by the variable’s name and type.

Suppose you want the procedure to display the age of an employee. Before you can calculate the age, you must feed the procedure the employee’s date of birth. To do this, you declare a variable called `dateOfBirth`, as follows:

```
Dim dateOfBirth As Date
```

Notice that the `Dim` keyword is followed by the name of the variable (`dateOfBirth`). If you don’t like this name, you are free to replace it with another word, as long as the word you are planning to use is not one of the VBA keywords. You specify the data type the variable will hold by including the `As` keyword followed by one of the data types from Table 3.1. The `Date` data type tells Visual Basic that the variable `dateOfBirth` will store a date.

To store the employee’s age, you declare the variable as follows:

```
Dim intAge As Integer
```

The `intAge` variable will store the number of years between today’s date and the employee’s date of birth. Because age is displayed as a whole number, the `intAge` variable has been assigned the `Integer` data type. You may also want your procedure to keep track of the employee’s name, so you declare another variable to hold the employee’s first and last name:

```
Dim strFullName As String
```

Because the word `Name` is on the VBA list of reserved words, using it in your VBA procedure would guarantee an error. To hold the employee’s full name, we used the variable `strFullName` and declared it as the `String` data type because the data it will hold is text. Declaring variables is regarded as good programming practice because it makes programs easier to read and helps prevent certain types of errors.

### SIDE BAR *Informal (Implicit) Variables*

Variables that are not explicitly declared with `Dim` statements are said to be implicitly declared. These variables are automatically assigned a data type called Variant. They can hold numbers, strings, and other types of information. You can create an informal variable by assigning some value to a variable name anywhere in your VBA procedure. For example, you implicitly declare a variable in the following way:

```
intDaysLeft = 100
```

Now that you know how to declare your variables, let's write a procedure that uses them.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 3.1 Using Variables

1. Start Microsoft Access and create a new database named **Chap03.accdb** in your C:\VBAAccess2021\_ByExample\_Primer folder.
2. Once your new database is opened, press **Alt+F11** to switch to the Visual Basic Editor window.
3. Choose **Insert | Module** to add a new standard module, and notice **Module1** under the Modules folder in the Project Explorer window.
4. In the **Module1 (Code)** window, enter the following **AgeCalc** procedure.

```
Sub AgeCalc()
    ' variable declaration
    Dim strFullName As String
    Dim dateOfBirth As Date
    Dim intAge As Integer

    ' assign values to variables
    strFullName = "John Smith"
    dateOfBirth = #1/3/1967#

    ' calculate age
    IntAge = Year(Now()) - Year(dateOfBirth)

    ' print results to the Immediate window
    Debug.Print strFullName & " is " & intAge & " years old."
End Sub
```

Notice that in the **AgeCalc** procedure the variables are declared on separate lines at the beginning of the procedure. You can also declare several variables on the same line, separating each variable name with a comma, as shown here (be sure to enter this on one line):

```
Dim strFullName As String, dateOfBirth As Date, intAge As Integer
```

When you list all your variables on one line, the **Dim** keyword appears only once at the beginning of the variable declaration line.

5. If the Immediate window is not open, press **Ctrl+G** or choose **View | Immediate Window**. Because the example procedure writes the results to the Immediate window, you should ensure that this window is open prior to executing Step 6.
6. To run the AgeCalc procedure, click any line between the `Sub` and `End Sub` keywords and press **F5**.

---

**SIDE BAR** *What Is the Variable Type?*

You can find out the type of a variable used in your procedure by right-clicking the variable name and selecting Quick Info from the shortcut menu.

---

When Visual Basic executes the variable declaration statements, it creates the variables with the specified names and reserves memory space to store their values. Then specific values are assigned to these variables. To assign a value to a variable, you begin with a variable name followed by an equal sign. The value entered to the right of the equal sign is the data you want to store in the variable. The data you enter here must be of the type stated in the variable declaration. Text data should be surrounded by quotation marks and dates by # characters.

Using the data supplied by the `dateOfBirth` variable, Visual Basic calculates the age of an employee and stores the result of the calculation in the variable called `intAge`. Then, the full name of the employee and the age are printed to the Immediate window using the instruction

```
Debug.Print strFullName & " is " & intAge & " years old."
```

---

**SIDE BAR** *Concatenation*

You can combine two or more strings to form a new string. The joining operation is called *concatenation*. You saw an example of concatenated strings in the `AgeCalc` procedure in Hands-On 3.1. Concatenation is represented by an ampersand character (&). For instance, "His name is " & `strFirstName` will produce a string like: His name is John or His name is Michael. The name of the person is determined by the contents of the `strFirstName` variable. Notice that there is an extra space between "is" and the ending quotation mark: "His name is ". Concatenation of strings can also be represented by a plus sign (+); however, many programmers prefer to restrict the plus sign to numerical operations to eliminate ambiguity.

---

---

**Specifying the Data Type of a Variable**

If you don't specify the variable's data type in the `Dim` statement, you end up with the *untyped* variable. Untyped variables in VBA are always assigned the

Variant data type. Variant data types can hold all the other data types (except for user-defined data types). This feature makes Variant a very flexible and popular data type. Despite this flexibility, it is highly recommended that you create typed variables. When you declare a variable of a certain data type, your VBA procedure runs faster because Visual Basic does not have to stop to analyze the variable to determine its type.

Visual Basic can work with many types of numeric variables. Integer variables can hold only whole numbers from -32,768 to 32,767. Other types of numeric variables are Long, Single, Double, and Currency. The Long variables can hold whole numbers in the range -2,147,483,648 to 2,147,483,647. As opposed to Integer and Long variables, Single and Double variables can hold decimals.

String variables are used to refer to text. When you declare a variable of the String data type, you can tell Visual Basic how long the string should be. For instance, `Dim strExtension As String * 3` declares the fixed-length String variable named `strExtension` that is three characters long. If you don't assign a specific length, the String variable will be *dynamic*. This means that Visual Basic will make enough space in computer memory to handle whatever text length is assigned to it.

After a variable is declared, it can store only the type of information that you stated in the declaration statement.

Assigning string values to numeric variables or numeric values to string variables results in the error message "Type Mismatch" or causes Visual Basic to modify the value. For example, if your variable was declared to hold whole numbers and your data uses decimals, Visual Basic will disregard the decimals and use only the whole part of the number.

Let's use the `MyNumber` procedure in Hands-On 3.2 as an example of how Visual Basic modifies the data according to the assigned data types.



### Hands-On 3.2 Understanding the Data Type of a Variable

This hands-on exercise uses the `C:\VBAAccess2021_ByExample_Primer\Chap03.accdb` database that you created in Hands-On 3.1.

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new module.
2. Enter the following procedure code for **MyNumber** in the new module's Code window.

```
Sub MyNumber()  
    Dim intNum As Integer
```

```

intNum = 23.11
MsgBox intNum
End Sub

```

3. To run the procedure, click any line between the `Sub` and `End Sub` keywords and press F5 or choose **Run | Run Sub/UserForm**.

When you run this procedure, Visual Basic displays the contents of the variable `intNum` as 23, and not 23.11, because the `intNum` variable was declared as an Integer data type.

### ***Using Type Declaration Characters***

---

If you don't declare a variable with a `Dim` statement, you can still designate a type for it by using a special character at the end of the variable name. For example, to declare the `FirstName` variable as String, you append the dollar sign to the variable name:

```
Dim FirstName$
```

This is the same as `Dim FirstName As String`. Other type declaration characters are shown in Table 3.2. Notice that the type declaration characters can be used only with six data types. To use the type declaration character, append the character to the end of the variable name.

**TABLE 3.2** Type declaration characters.

Data Type	Character
Integer	%
Long	&
Single	!
Double	#
Currency	@
String	\$

### **SIDE BAR** *Declaring Typed Variables*

---

The variable type can be indicated by the `As` keyword or by attaching a type symbol. If you don't add the type symbol or the `As` command, VBA will default the variable to the Variant data type.



### Hands-On 3.3 Using Type Declaration Characters in Variable Names

This hands-on exercise uses the Chap03.accdb database that you created in Hands-On 3.1.

1. In the Visual Basic window, choose **Insert | Module** to add a new module.
2. Enter the **AgeCalc2** procedure code in the new module's Code window.

```
Sub AgeCalc2()
    ' variable declaration
    Dim FullName$
    Dim DateOfBirth As Date
    Dim age%

    ' assign values to variables
    FullName$ = "John Smith"
    DateOfBirth = #1/3/1967#

    ' calculate age
    age% = Year(Now()) - Year(DateOfBirth)

    ' print results to the Immediate window
    Debug.Print FullName$ & " is " & age% & " years old."
End Sub
```

3. To run the procedure, click any line between the **Sub** and **End Sub** keywords and press **F5** or choose **Run | Run Sub/UserForm**.

## Assigning Values to Variables

---

Now that you know how to correctly name and declare variables, it's time to learn how to initialize them.



### Hands-On 3.4 Assigning Values to Variables

This hands-on exercise uses the **C:\VBAAccess2021\_ByExample\_Primer\Chap03.accdb** database that you created in Hands-On 3.1.

1. In the Visual Basic window, choose **Insert | Module** to add a new module.
2. Enter the code of the **CalcCost** procedure in the new module's Code window.

```
Sub CalcCost()
    slsPrice = 35
    slsTax = 0.085
    cost = slsPrice + (slsPrice * slsTax)
```

```
strMsg = "The calculator total is $" & cost & "."
MsgBox strMsg
End Sub
```

3. To run the procedure, click any line between the `Sub` and `End Sub` keywords and press **F5** or choose **Run | Run Sub/UserForm**.
4. Change the calculation of the `cost` variable in the **CalcCost** procedure as follows:

```
cost = Format(slsPrice + (slsPrice * slsTax), "0.00")
```

5. To run the modified procedure, click any line between the `Sub` and `End Sub` keywords and press **F5** or choose **Run | Run Sub/UserForm**.

The `CalcCost` procedure uses four variables: `slsPrice`, `slsTax`, `cost`, and `strMsg`. Because none of these variables have been explicitly declared with the `Dim` keyword and a specific data type, they all have the same data type—Variant. The variables `slsPrice` and `slsTax` were created by assigning some values to the variable names at the beginning of the procedure. The `cost` variable was assigned the value resulting from the calculation `slsPrice + (slsPrice * slsTax)`. The `cost` calculation uses the values supplied by the `slsPrice` and `slsTax` variables. The `strMsg` variable puts together a text message to the user. This message is then displayed with the `MsgBox` function.

When you assign values to variables, you follow the name of the variable with the equal sign. After the equal sign you enter the value of the variable. This can be text surrounded by quotation marks, a number, or an expression. While the values assigned to the variables `slsPrice`, `slsTax`, and `cost` are easily understood, the value stored in the `strMsg` variable is a little more involved.

Let's examine the content of the `strMsg` variable:

```
strMsg = "The calculator total is $" & cost & "."
```

- The string "The calculator total is \$" begins and ends with quotation marks. Notice the extra space before the ending quotation mark.
- The `&` symbol allows one string to be appended to another string or to the contents of a variable and must be used every time you want to append a new piece of information to the previous string.
- The `cost` variable is a placeholder. The actual cost of the calculator will be displayed here when the procedure runs.
- The `&` symbol attaches yet another string.
- The period (.) is a character and must be surrounded by quotation marks. When you require a period at the end of the sentence, you must attach it separately when it follows the name of a variable.

**SIDE BAR** *Variable Initialization*

Visual Basic automatically initializes a new variable to its default value when it is created. Numerical variables are set to zero (0), Boolean variables are initialized to False, string variables are set to the empty string (""), and Date variables are set to December 30, 1899.

Notice that the cost displayed in the message box has three decimal places. To display the cost of a calculator with two decimal places, you need to use a function. VBA has special functions that allow you to change the format of data. To change the format of the cost variable you should use the Format function. This function has the following syntax:

```
Format(expression, format)
```

where `expression` is a value or variable you want to format, and `format` is the type of format you want to apply.

After having tried the CalcCost procedure, you may wonder why you should bother declaring variables if Visual Basic can handle undeclared variables so well. The CalcCost procedure is very short, so you don't need to worry about how many bytes of memory will be consumed each time Visual Basic uses the Variant variable. In short procedures, however, it is not the memory that matters but the mistakes you are bound to make when typing variable names. What will happen if the second time you use the `cst` variable you omit the "o" and refer to it as `cst`?

```
strMsg = "The calculator total is " & "$" & cst & ".."
```

And what will you end up with if, instead of `slsTax`, you use the word `tax` in the formula?

```
cost = Format(slsPrice + (slsPrice * tax), "0.00")
```

When you run the procedure with the preceding errors introduced, Visual Basic will not show the cost of the calculator because it does not find the assignment statement for the `cst` variable. And because Visual Basic does not know the sales tax, it displays the price of the calculator as the total cost. Visual Basic does not guess—it simply does what you tell it to do. This brings us to the next section, which explains how to make sure that errors of this sort don't occur.

**NOTE**

*Before you continue with this chapter, be sure to replace the names of the variables `cst` and `tax` with `cost` and `slsTax`.*

## Forcing Declaration of Variables

Visual Basic has an `Option Explicit` statement that you can use to automatically remind yourself to formally declare all your variables. This statement must be entered at the top of each of your modules. The `Option Explicit` statement will cause Visual Basic to generate an error message when you try to run a procedure that contains undeclared variables like the one in the previous Hands-On example.



### Hands-On 3.5 Forcing Declaration of Variables

1. Return to the Code window where you entered the `CalcCost` procedure (see Hands-On 3.4).
2. At the top of the module window (below the `Option Compare Database` statement), enter

```
Option Explicit
```

and press **Enter**. Visual Basic will display the statement in blue.

3. Position the insertion point anywhere within the `CalcCost` procedure and press **F5** to run it. Visual Basic displays this error message: “Compile error: Variable not defined.”
4. Click **OK** to exit the message box. Visual Basic selects the name of the variable, `slsPrice`, and highlights in yellow the name of the procedure, `Sub CalcCost()`. The titlebar displays “Microsoft Visual Basic for Applications—Chap03 [break]—[Module4 (Code)].” The Visual Basic Break mode allows you to correct the problem before you continue. Now you must formally declare the `slsPrice` variable.
5. Enter the declaration statement

```
Dim slsPrice As Currency
```

on a new line just below `Sub CalcCost()` and press **F5** to continue. When you declare the `slsPrice` variable and rerun your procedure, Visual Basic will generate the same compile error as soon as it encounters another variable name that was not declared. To fix the remaining problems with the variable declaration in this procedure, choose **Run | Reset** to exit the Break mode.

6. Enter the following declarations at the beginning of the `CalcCost` procedure:

```
' declaration of variables
Dim slsPrice As Currency
Dim slsTax As Single
Dim cost As Currency
Dim strMsg As String
```

7. To run the procedure, click any line between the `Sub` and `End Sub` keywords and press **F5** or choose **Run | Run Sub/UserForm**. Your revised `CalcCost` procedure looks like this:

```
' revised CalcCost procedure with variable declarations

Sub CalcCost_Revised()
    ' declaration of variables
    Dim slsPrice As Currency
    Dim slsTax As Single
    Dim cost As Currency
    Dim strMsg As String

    slsPrice = 35
    slsTax = 0.085

    cost = Format(slsPrice + (slsPrice * slsTax), "0.00")
    strMsg = "The calculator total is $" & cost & "."

    MsgBox strMsg
End Sub
```

The `Option Explicit` statement you entered at the top of the module Code window (see step 2) forced you to declare variables. Because you must include the `Option Explicit` statement in each module where you want to require variable declaration, you can have Visual Basic enter this statement for you each time you insert a new module.

To automatically include `Option Explicit` in every new module you create, follow these steps:

1. Choose **Tools | Options**.
2. Ensure that the **Require Variable Declaration** checkbox is selected in the Options dialog box (Editor tab).
3. Choose **OK** to close the Options dialog box.

From now on, every new module you add to your database will have the `Option Explicit` statement. If you want to require variables to be explicitly declared in a module you created prior to enabling `Require Variable Declaration` in the Options dialog box, you must enter the `Option Explicit` statement manually by editing the module yourself.

---

**SIDE BAR** *More about Option Explicit*

Option Explicit forces formal (explicit) declaration of all variables in a module. One big advantage of using Option Explicit is that misspellings of variable names will be detected at compile time (when Visual Basic attempts to translate the source code to executable code). The Option Explicit statement must appear in a module before any procedures.

---

### **Understanding the Scope of Variables**

---

Variables can have different ranges of influence in a VBA procedure. *Scope* defines the availability of a variable to the same procedure or other procedures. Variables can have the following three levels of scope in Visual Basic for Applications:

- Procedure-level scope
- Module-level scope
- Project-level scope

#### ***Procedure-Level (Local) Variables***

---

From this chapter you already know how to declare a variable using the Dim statement. The position of the Dim statement in the module determines the scope of a variable. Variables declared with the Dim statement within a VBA procedure have a *procedure-level* scope. Procedure-level variables can also be declared by using the Static statement (see “Using Static Variables” later in this chapter).

Procedure-level variables are frequently referred to as *local* variables, which can be used only in the procedure where they were declared. Undeclared variables always have a procedure-level scope.

A variable’s name must be unique within its scope. This means that you cannot declare two variables with the same name in the same procedure. However, you can use the same variable name in different procedures. In other words, the CalcCost procedure can have the slsTax variable, and the ExpenseRep procedure in the same module can have its own variable called slsTax. Both variables are independent of each other.

**SIDE BAR** **Local Variables: With Dim or Static?**

When you declare a local variable with the `Dim` statement, the value of the variable is preserved only while the procedure in which it is declared is running. As soon as the procedure ends, the variable dies. The next time you execute the procedure, the variable is reinitialized.

When you declare a local variable with the `Static` statement, the value of the variable is preserved after the procedure in which the variable was declared has finished running. Static variables are reset when you quit Access or when a runtime error occurs while the procedure is running.

**Module-Level Variables**

Often you want the variable to be available to other VBA procedures in the module after the procedure in which the variable was declared has finished running. This situation requires that you change the variable's scope to *module-level*.

Module-level variables are declared at the top of the module (above the first procedure definition) by using the `Dim` or `Private` statement. These variables are available to all of the procedures in the module in which they were declared but are not available to procedures in other modules.

For instance, to make the `slsTax` variable available to any other procedure in the module, you could declare it by using the `Dim` or `Private` statement:

```
Option Explicit  
Dim slsTax As Single ' module-level variable declared with  
' Dim statement  
  
Sub CalcCost()  
...Instructions of the procedure...  
End Sub
```

Notice that the `slsTax` variable is declared at the top of the module, just below the `Option Explicit` statement and before the first procedure definition. You could also declare the `slsTax` variable like this:

```
Option Explicit  
Private slsTax As Single ' module-level variable declared with  
' Private statement  
  
Sub CalcCost()  
...Instructions of the procedure...  
End Sub
```

There is no difference between module-level variables declared with `Dim` or `Private` statements.

Before you can see how module-level variables work, you need to create another procedure that also uses the `slsTax` variable.



### Hands-On 3.6 Understanding Module-Level Variables

This hands-on exercise requires the prior completion of Hands-On 3.4 and 3.5.

1. In the Code window, in the same module where you entered the `CalcCost_Revised` procedure, copy the declaration line `Dim slsTax As Single` and paste it at the top of the module sheet, below the `Option Explicit` statement.
2. Comment the declaration line `Dim slsTax As Single` inside the `CalcCost_Revised` procedure.
3. Enter the following code of the `ExpenseRep` procedure in the same module where the `CalcCost_Revised` procedure is located (see Figure 3.1).

```
Sub ExpenseRep()
    Dim slsPrice As Currency
    Dim cost As Currency

    slsPrice = 55.99
    cost = slsPrice + (slsPrice * slsTax)

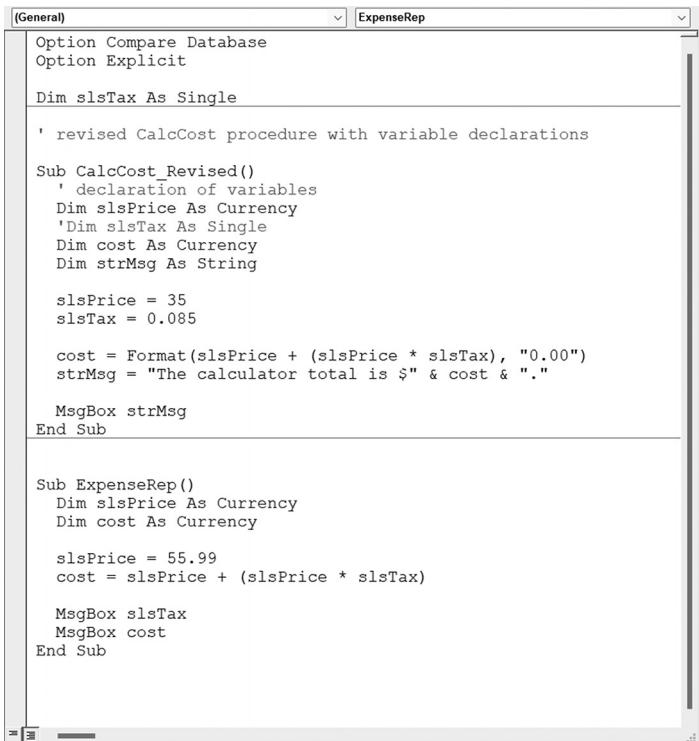
    MsgBox slsTax
    MsgBox cost
End Sub
```

The `ExpenseRep` procedure declares two `Currency` type variables: `slsPrice` and `cost`. The `slsPrice` variable is then assigned a value of 55.99. The `slsPrice` variable is independent of the `slsPrice` variable declared within the `CalcCost` procedure.

The `ExpenseRep` procedure calculates the cost of a purchase. The cost includes the sales tax. Because the sales tax is the same as the one used in the `CalcCost_Revised` procedure, the `slsTax` variable has been declared at the module level. After Visual Basic executes the `CalcCost_Revised` procedure, the contents of the `slsTax` variable equals 0.085. If `slsTax` were a local variable, the contents of this variable would be empty upon the termination of the `CalcCost_Revised` procedure. The `ExpenseRep` procedure ends by displaying the value of the `slsTax` and `cost` variables in two separate message boxes.

After running the `CalcCost_Revised` procedure, Visual Basic erases the contents of all the variables except for the `slsTax` variable, which was declared

at a module level. As soon as you attempt to calculate the cost by running the ExpenseRep procedure, Visual Basic retrieves the value of the `slsTax` variable and uses it in the calculation.



```
[General] [ExpenseRep]
Option Compare Database
Option Explicit

Dim slsTax As Single

' revised CalcCost procedure with variable declarations

Sub CalcCost_Revised()
    ' declaration of variables
    Dim slsPrice As Currency
    'Dim slsTax As Single
    Dim cost As Currency
    Dim strMsg As String

    slsPrice = 35
    slsTax = 0.085

    cost = Format(slsPrice + (slsPrice * slsTax), "0.00")
    strMsg = "The calculator total is $" & cost & "."

    MsgBox strMsg
End Sub

Sub ExpenseRep()
    Dim slsPrice As Currency
    Dim cost As Currency

    slsPrice = 55.99
    cost = slsPrice + (slsPrice * slsTax)

    MsgBox slsTax
    MsgBox cost
End Sub
```

FIGURE 3.1 The `slsTax` variable is declared as a module-level variable so it can be accessed by other procedures in the same module.

4. Click anywhere inside the revised `CalcCost_Revised` procedure and press F5 to run it.
5. As soon as the `CalcCost_Revised` procedure finishes executing, run the `ExpenseRep` procedure.

### **Project-Level Variables**

In the previous sections, you learned that declaring a variable with the `Dim` or `Private` keyword at the top of the module makes it available to other procedures in that module. Module-level variables that are declared with the `Public` keyword (instead of `Dim` or `Private`) have a project-level scope. This means that

they can be used in any Visual Basic for Applications module. When you want to work with a variable in all the procedures in all the open VBA projects, you must declare it with the `Public` keyword—for instance:

```
Option Explicit
Public gs1sTax As Single

Sub CalcCost()
    ...Instructions of the procedure...
End Sub
```

Notice that the `gs1sTax` variable declared at the top of the module with the `Public` keyword will now be available to any VBA modules that your code references. This type of variable is called a *global variable*. It is customary to use the prefix “g” to indicate its global scope.

When using global variables, it’s important to keep in mind the following:

- The value of the global variable can be changed anywhere in your program. An unexpected change in the value of a variable is a common cause of problems. Be careful not to write a block of code that modifies a global variable. If you need to change the value of a variable within your application, make sure you are using a local variable.
- Values of all global variables declared with the `Public` keyword are cleared when Access encounters an error. Since the release of the Access 2007 database format (ACCDB), you can use the `TempVars` collection for your global variable needs (see “Using Temporary Variables” later in this chapter).
- Don’t put your global variable declaration in a form class module. Variables in the code module behind the form are never global even if you declare them as such. You must use a standard code module (Insert | Module) to declare variables to be available in all modules and forms. Variables declared in a standard module can be used in the code for any form.
- Use constants as much as possible whenever your application requires global variables. Constants are much more reliable because their values are static. Constants are covered later in this chapter.

---

**SIDE BAR** ***Public Variables and the Option Private Module Statement***

Variables declared using the `Public` keyword are available to all procedures in all modules across all applications. To restrict a public module-level variable

to the current database, include the `Option Private Module` statement in the declaration section of the standard or class module in which the variable is declared.

---

### **Understanding the Lifetime of Variables**

In addition to scope, variables have a *lifetime*. The *lifetime* of a variable determines how long a variable retains its value. Module-level and project-level variables preserve their values as long as the project is open. Visual Basic, however, can reinitialize these variables if required by the program's logic. Local variables declared with the `Dim` statement lose their values when a procedure has finished. Local variables have a lifetime while a procedure is running, and they are reinitialized every time the program is run. Visual Basic allows you to extend the lifetime of a local variable by changing the way it is declared.

### **Using Temporary Variables**

In the previous section, you learned that you could declare a global variable with the `Public` keyword and use it throughout your entire application. You also learned that these variables can be quite problematic, especially when you or another programmer accidentally changes the value of the variable, or your application encounters an error and the values of the variables you have initially set for your application to use are completely wiped out. To avoid such problems, many programmers resort to using separate global variables form to hold their global variables. And if they need certain values to be available the next time the application starts, they create a separate database table to store these values. A *global variables form* is simply a blank Access form where you can place both bound and unbound controls. Bound controls are used to pull the data from the table where global variables have been stored. You can use unbound controls on a form to store values of global variables that are not stored in a separate table. Simply set the `ControlSource` property of the unbound control by typing a value in it or use a VBA procedure to set the value of the `ControlSource`. The form set up as a global variables form must be open while the application is running for the values of the bound and unbound controls to be available to other forms, reports, and queries in the database. A global variables form can be hidden if the values of the global variables are pulled from a database table or set using VBA procedures or macro actions.

If your database is in the ACCDB format, instead of using a database table or global variables, you can use the `TempVars` collection to store the Variant

values you want to reuse. `TempVars` stands for *temporary variables*. Temporary variables are global. You can refer to them in VBA modules, event procedures, queries, expressions, add-ins, and in any referenced databases. Access .ACCDB databases allow you to define up to 255 temporary variables at one time. These variables remain in memory until you close the database (unless you remove them when you are finished working with them). Unlike public variables, temporary variable values are not cleared when an error occurs.

### ***Creating a Temporary Variable with a TempVars Collection Object***

---

Let's look at some examples of using the `TempVars` collection first introduced in Access 2007. Assume your application requires three variables named `gtvUserName`, `gtvUserFolder`, and `gtvEndDate`.

To try this out, open the Immediate window and type the following statements. The variable is created as soon as you press Enter after each statement.

```
TempVars("gtvUserName").Value = "John Smith"  
TempVars("gtvUserFolder").Value = Environ("HOMEPATH")  
TempVars("gtvEndDate").Value = Format(now(), "mm/dd/yyyy")
```

Notice that to create a temporary variable, all you must do is specify its value. If the variable does not already exist, Access adds it to the `TempVars` collection. If the variable exists, Access modifies its value.

You can explicitly add a global variable to the `TempVars` collection by using the `Add` method, like this:

```
TempVars.Add "gtvCompleted", "true"
```

### ***Retrieving Names and Values of TempVar Objects***

---

Each `TempVar` object in the `TempVars` collection has `Name` and `Value` properties that you can use to access the variable and read its value from any procedure. By default, the items in the collection are numbered from zero (0), with the first item being zero, the second item being one, the third two, and so on. Therefore, to find the value of the second variable in the `TempVars` you have entered (`gtvUserFolder`), type the following statement in the Immediate window:

```
?TempVars(1).Value
```

When you press Enter, you will see the location of the user's private folder on the computer. In this case, it is your private folder. The folder information was returned by passing the "HOMEPATH" parameter to the built-in `Environ` func-

tion. Functions and parameter passing are covered in Chapter 4.

You can also retrieve the value of the variable from the `TempVars` collection by using its name, like this:

```
?TempVars("gtvUserFolder").Value
```

You can iterate through the `TempVars` collection to see the names and values of all global variables that you have placed in it. To do this from the Immediate window, you need to use the colon operator (`:`) to separate lines of code. Type the following statement all on one line to try this out:

```
For Each gtv In TempVars : Debug.Print gtv.Name & ":"  
& gtv.Value : Next
```

When you press Enter, the `Debug.Print` statement will write to the Immediate window a name and value for each variable that is currently stored in the `TempVars` collection:

```
gtvUserName:John Smith  
gtvUserFolder:\Users\Julitta  
gtvEndDate:11/30/2021  
gtvCompleted:true
```

The `For Each...Next` statement, a popular VBA programming construct, is covered in detail in Chapter 6. The “`gtv`” is an object variable used as an iterator. An *iterator* allows you to traverse through all the elements of a collection. You can use any variable name as an iterator provided it is not a VBA keyword. Object variables are discussed later in this chapter. For more information on working with collections, see Chapter 8.

### **Using Temporary Global Variables in Expressions**

---

You can use temporary global variables anywhere expressions can be used. For example, you can set the value of the unbound text box control on a form to display the value of your global variable by activating the property sheet and typing the following in the `ControlSource` property of the text box:

```
= [TempVars]![gtvCompleted]
```

You can also use a temporary variable to pass selection criteria to queries:

```
SELECT * FROM Orders WHERE Order_Date = TempVars!gtvEndDate
```

### ***Removing a Temporary Variable from a TempVars Collection Object***

---

When you are done using a variable, you can remove it from the `TempVars` collection with the `Remove` method, like this:

```
TempVars.Remove "gtvUserFolder"
```

To check the number of the `TempVar` objects in the `TempVars` collection, use the `Count` property in the Immediate window:

```
?TempVars.Count
```

Finally, to quickly remove all global variables (`TempVar` objects) from the `TempVars` collection, simply use the `RemoveAll` method, like this:

```
TempVars.RemoveAll
```

---

**SIDE BAR** ***The TempVars Collection Is Exposed to Macros***

The following three macros allow macro users to set and remove `TempVar` objects:

- `SetTempVar`—Sets a `TempVar` to a given value. You must specify the name of the temporary variable and the expression that will be used to set the value of this variable. Expressions must be entered without an equal sign (=).
- `RemoveTempVar`—Removes the `TempVar` from the `TempVars` collection. You must specify the name of the temporary variable you want to remove.
- `RemoveAllTempVars`—Clears the `TempVars` collection.

The values of `TempVar` objects can be used in the arguments and in the condition columns of macros.

---

### ***Using Static Variables***

---

A variable declared with the `Static` keyword is a special type of local variable. *Static variables* are declared at the procedure level. Unlike the local variables declared with the `Dim` keyword, static variables remain in existence and retain their values when the procedure in which they were declared ends.

The `CostOfPurchase` procedure (see Hands-On 3.7) demonstrates the use of the static variable `allPurchase`. The purpose of this variable is to keep track of the running total.



### Hands-On 3.7 Using Static Variables

This hands-on exercise uses the **C:\VBAAccess2021\_ByExample\_Primer\Chap03.accdb** database that you created in Hands-On 3.1.

1. In the Visual Basic window, choose **Insert | Module** to add a new module.
2. Enter the following **CostOfPurchase** procedure code in the new module's Code window.

```
Sub CostOfPurchase()
    ' declare variables
    Static allPurchase
    Dim newPurchase As String
    Dim purchCost As Single

    newPurchase = InputBox("Enter the cost of a purchase:")
    purchCost = CSng(newPurchase)
    allPurchase = allPurchase + purchCost

    ' display results
    MsgBox "The cost of a new purchase is: " & newPurchase
    MsgBox "The running cost is: " & allPurchase
End Sub
```

This procedure begins with declaring a static variable named `allPurchase` and two local variables named `newPurchase` and `purchCost`. The `InputBox` function is used to get a user's input while the procedure is running. As soon as the user inputs the value and clicks OK, Visual Basic assigns the value to the `newPurchase` variable. Because the result of the `InputBox` function is always a string, the `newPurchase` variable was declared as the String data type. You cannot use strings in mathematical calculations, so the next instruction uses a *type conversion* function (`CSng`) to translate the text value into a numeric value, which is stored as a Single data type in the variable `purchCost`. The `CSng` function requires only one argument: the value you want to translate. Refer to Chapter 4 for more information about converting data types.

The next instruction, `allPurchase = allPurchase + purchCost`, adds the new value supplied by the `InputBox` function to the current purchase value. When you run this procedure for the first time, the value of the `allPurchase` variable is the same as the value of the `purchCost` variable. During the second run, the value of the static variable is increased by the new value entered in the dialog box. You can run the `CostOfPurchase` procedure as many times as

you want. The `allPurch` variable will keep the running total for as long as the project is open.

3. To run the procedure, position the insertion point anywhere within the `CostOfPurchase` procedure and press **F5**.
4. When the dialog box appears, enter a number. For example, type **100** and press **Enter**. Visual Basic displays the message “The cost of a new purchase is: 100.”
5. Click **OK** in the message box. Visual Basic displays the second message “The running cost is: 100.”
6. Rerun the same procedure.
7. When the input box appears, enter another number. For example, type **50** and press **Enter**. Visual Basic displays the message “The cost of a new purchase is: 50.”
8. Click **OK** in the message box. Visual Basic displays the second message “The running cost is: 150.”
9. Run the procedure a couple of times to see how Visual Basic keeps track of the running total.

---

**SIDE BAR** *Type Conversion Functions*

To learn more about the `Csng` function, position the insertion point anywhere within the word `Csng` and press F1.

---

---

## Using Object Variables

---

The variables we've introduced so far are used to store data, which is the main reason for using “normal” variables in your procedures. There are also special variables that refer to the Visual Basic objects. These variables are called *object variables*. Object variables don't store data; they store the location of the data. You can use them to reference databases, forms, and controls as well as objects created in other applications. Object variables are declared in a similar way as the variables you've already seen. The only difference is that after the `As` keyword, you enter the type of object your variable will point to—for instance:

```
Dim myControl As Control
```

This statement declares the object variable called `myControl` of type `Control`.

```
Dim frm As Form
```

This statement declares the object variable called `frm` of type `Form`.

You can use object variables to refer to objects of a generic type, such as `Application`, `Control`, `Form`, or `Report`, or you can point your object variable

to specific object types, such as TextBox, ToggleButton, CheckBox, CommandButton, ListBox, OptionButton, Subform or Subreport, Label, BoundObjectFrame or UnboundObjectFrame, and so on. When you declare an object variable, you also must assign it a specific value before you can use it in your procedure. You assign a value to the object variable by using the `Set` keyword followed by the equal sign and the value that the variable refers to—for example:

```
Set myControl = Me!CompanyName
```

The preceding statement assigns a value to the object variable called `myControl`. This object variable will now point to the `CompanyName` control on the active form. If you omit the word `Set`, Visual Basic will display the error message “Runtime error 91: *Object variable or With block variable not set.*”

Again, it’s time to see a practical example. The `HideControl` procedure in Hands-On 3.8 demonstrates the use of two object variables `frm` and `myControl`.



### Hands-On 3.8 Working with Object Variables

1. Close the currently open Access database **Chap03.accdb**. When prompted to save changes in the modules, click **OK**. Save the modules with the suggested default names `Module1`, `Module2`, and so on.
2. Copy the **HandsOn\_03\_8.accdb** database from the companion files to your **C:\VBAAccess2021\_ByExample\_Primer** folder. This database contains a `Customer` table and a simple `Customer` form imported from the Northwind.mdb sample database that shipped with an earlier version of Access.
3. Open Access and load the **C:\VBAAccess2021\_ByExample\_Primer\HandsOn\_03\_8.accdb** database file.
4. Open the **Customers** form in Form view.
5. Press **Alt+F11** to switch to the Visual Basic Editor window.
6. Choose **Insert | Module** to add a new module.
7. Enter the following **HideControl** procedure code in the new module’s Code window.

```
Sub HideControl()
    Dim frm As Form
    Dim myControl As Control
    Dim strFormName As String

    strFormName = "Customers"

    'Open the specified form
    DoCmd.OpenForm strFormName, acNormal
```

```
' set an object variable pointing to the form
Set frm = Forms!Customers
' set an object variable pointing to the CompanyName control
Set myControl = frm.CompanyName

' manipulate the visibility of the form control
myControl.Visible = False
End Sub
```

8. To run the procedure, click any line between the `Sub` and `End Sub` keywords and press F5 or choose **Run | Run Sub/UserForm**.

The procedure begins with the declaration of two object variables called `frm` and `myControl`. The object variable `frm` is set to reference the Customers form. For the procedure to work, the referenced form must be open. We can open the form using this statement:

```
DoCmd.OpenForm strFormName, acNormal
```

Next, the `myControl` object variable is set to point to the `CompanyName` control located on the Customers form.

Instead of using the object's entire address, you can use the shortcut—the name of the object variable. For example, the statement

```
Set myControl = frm.CompanyName
```

is the same as

```
Set myControl = Forms!Customers.CompanyName
```

The purpose of this procedure is to hide the control referenced by the object variable `myControl`. After running the `HideControl` procedure, switch to the Access window containing the open Customers form. The `CompanyName` control should not be visible on the form. Change the visibility of the control to bring it back by changing the `Visible` property of `myControl` to `True` and rerun the procedure. To programmatically close the open form, use the following statement:

```
DoCmd.Close acForm, strFormName, acSaveYes
```

### SIDE BAR *Advantages of Using Object Variables*

The advantages of object variables are:

- They can be used instead of the actual object.
- They are shorter and easier to remember than the actual values they point to.

- You can change their meaning while your procedure is running.

### ***Disposing of Object Variables***

---

When the object variable is no longer needed, you should assign Nothing to it. This frees up memory and system resources:

```
Set frm = Nothing  
Set myControl = Nothing
```

### **Finding a Variable Definition**

---

When you find an instruction that assigns a value to a variable in a VBA procedure, you can quickly locate the definition of the variable by selecting the variable name and pressing Shift+F2. Alternately, you can choose View | Definition. Visual Basic will jump to the variable declaration line. To return your mouse pointer to its previous position, press Ctrl+Shift+F2 or choose View | Last Position. Let's try it out.



#### **Hands-On 3.9 Finding a Variable Definition**

This hands-on exercise requires prior completion of Hands-On 3.8.

1. Locate the code of the procedure **HideControl** you created in Hands-On 3.8.
2. Locate the statement **myControl.Visible = .**
3. Right-click the **myControl** variable name and choose **Definition** from the shortcut menu.
4. Press **Ctrl+Shift+F2** to return to the previous location in the procedure code (**myControl.Visible = .**).

### **Determining the Data Type of a Variable**

---

Visual Basic has a built-in **VarType** function that returns an integer indicating the variable's type. Let's see how you can use this function in the Immediate window.



#### **Hands-On 3.10 Asking Questions about the Variable Type**

1. Open the Immediate window (**View | Immediate Window**) and type the following statements that assign values to variables:

```
age = 28  
birthdate = #1/1/1981#  
firstName = "John"
```

2. Now, ask Visual Basic what type of data each variable holds:

```
?varType (age)
```

When you press **Enter**, Visual Basic returns 2. The number 2 represents the Integer data type, as shown in Table 3.3.

```
?varType (birthdate)
```

Now Visual Basic returns 7 for Date. If you make a mistake in the variable name (let's say you type `birthday` instead of `birthdate`), Visual Basic returns zero (0).

```
?varType (firstName)
```

Visual Basic tells you that the value stored in the `firstName` variable is a String (8).

**TABLE 3.3** Values returned by the `VarType` function.

Constant	Value	Description
<code>vbEmpty</code>	0	Empty (uninitialized)
<code>vbNull</code>	1	Null (no valid data)
<code>vbInteger</code>	2	Integer
<code>vbLong</code>	3	Long integer
<code>vbSingle</code>	4	Single-precision floating-point number
<code>vbDouble</code>	5	Double-precision floating-point number
<code>vbCurrency</code>	6	Currency value
<code>vbDate</code>	7	Date value
<code>vbString</code>	8	String
<code>vbObject</code>	9	Object
<code>vbError</code>	10	Error value
<code>vbBoolean</code>	11	Boolean value
<code>vbVariant</code>	12	Variant (used only with arrays of variants)
<code>vbDataObject</code>	13	Data access object
<code>vbDecimal</code>	14	Decimal value
<code>vbByte</code>	17	Byte value
<code>vbLongLong</code>	20	Long Long integer (on 64-bit platform only)
<code>vbUserDefinedType</code>	36	Variants that contain user-defined types
<code>vbArray</code>	8192	Array

## USING CONSTANTS IN VBA PROCEDURES

---

The value of a variable can change while your procedure is executing. If your procedure needs to refer to unchanged values repeatedly, you should use constants. A *constant* is like a named variable that always refers to the same value. Visual Basic requires that you declare constants before you use them.

You declare constants by using the `Const` statement, as in the following examples:

```
Const dialogName = "Enter Data" As String
Const slsTax = 8.5
Const Discount = 0.5
Const ColorIdx = 3
```

A constant, like a variable, has a scope. To make a constant available within a single procedure, you declare it at the procedure level, just below the name of the procedure—for instance:

```
Sub WedAnniv()
    Const Age As Integer = 25
    ...instructions...
End Sub
```

If you want to use a constant in all the procedures of a module, use the `Private` keyword in front of the `Const` statement—for instance:

```
Private Const dsk = "B: " As String
```

The `Private` constant must be declared at the top of the module, just before the first `Sub` statement.

If you want to make a constant available to all modules in your application, use the `Public` keyword in front of the `Const` statement—for instance:

```
Public Const NumOfChar As Integer = 255
```

The `Public` constant must be declared at the top of the module, just before the first `Sub` statement.

When declaring a constant, you can use any one of the following data types: Boolean, Byte, Integer, Long, Currency, Single, Double, Date, String, or Variant.

Like variables, constants can be declared on one line if separated by commas—for instance:

```
Const Age As Integer = 25, PayCheck As Currency = 350
```

Using constants makes your VBA procedures more readable and easier to maintain. For example, if you need to refer to a certain value several times in your procedure, use a constant instead of using a value. This way, if the value changes (e.g., the sales tax rate goes up), you can simply change the value in the declaration of the `Const` statement instead of tracking down every occurrence of the value.

### Intrinsic Constants

---

Both Access and Visual Basic for Applications have a long list of predefined (intrinsic) constants that do not need to be declared. These built-in constants can be looked up using the Object Browser window, which was discussed in detail in Chapter 2.

Let's open the Object Browser to look at the list of constants in Access.



#### Hands-On 3.11 Exploring Access Constants

1. In the Visual Basic Editor window, choose **View | Object Browser**.
2. In the Project/Library list box, click the drop-down arrow and select the **Access** library.
3. Enter **constants** as the search text in the Search Text box and either press **Enter** or click the **Search** button. Visual Basic shows the results of the search in the Search Results area. The right side of the Object Browser window displays a list of all built-in constants available in the Microsoft Access Object Library (see Figure 3.2). Notice that the names of all the constants begin with the prefix “ac.”
4. To look up VBA constants, choose **VBA** in the Project/Library list box. Notice that the names of the VBA built-in constants begin with the prefix “vb.”

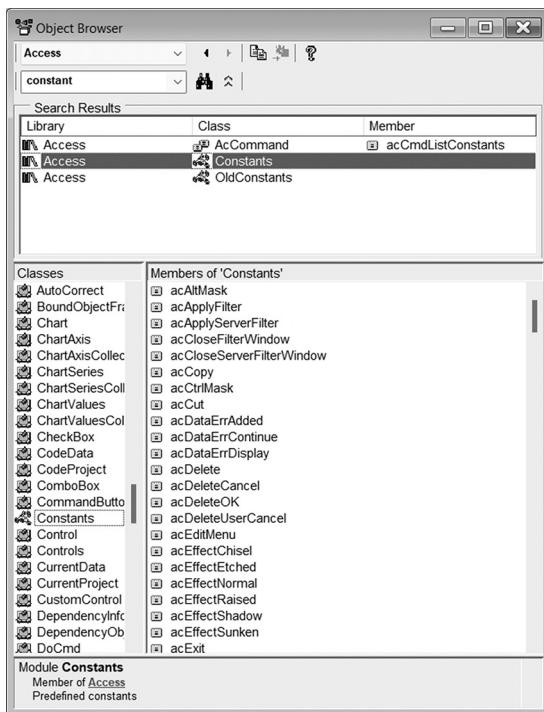


FIGURE 3.2 Use the Object Browser to look up any intrinsic constant.

## SUMMARY

This chapter has introduced you to several important VBA concepts such as data types, variables, and constants. You learned how to declare various types of variables and define their types. You also saw the difference between a variable and a constant.

In the next chapter, you will expand your knowledge of Visual Basic for Applications by writing procedures and functions with arguments. In addition, you will learn about built-in functions that allow your VBA procedures to interact with users.



# Chapter 4

# ACCESS VBA BUILT-IN AND CUSTOM FUNCTIONS

**A**s you already know from Chapter 1, VBA subroutines and function procedures often require arguments to perform certain tasks. In this chapter, you learn various methods of passing arguments to procedures and functions.

## WRITING FUNCTION PROCEDURES

---

Function procedures can perform calculations based on data received through arguments. When you declare a function procedure, you list the names of arguments inside a set of parentheses, as shown in Hands-On 4.1.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 4.1 Writing a Function Procedure with Arguments

1. Start Access and create a new database named **Chap04.accdb** in your **C:\VBAAccess2021\_ByExample\_Primer** folder.
2. Once your new database is opened, press **Alt+F11** to switch to the Visual Basic Editor window.

3. Choose **Insert | Module** to add a new standard module and notice that **Module1** appears under the **Modules** folder in the Project Explorer window.
4. In the **Module1 (Code)** window, enter the code of the **JoinText** function procedure as shown here.

```
Function JoinText(k, o)
    JoinText = k + " " + o
End Function
```

Note that there is a space character in quotation marks concatenated between the two arguments of the **JoinText** function's result: `JoinText = k + " " + o`.

A better way of adding a space is by using one of the following built-in functions:

```
JoinText = k + Space(1) + o
```

or:

```
JoinText = k + Chr(32) + o
```

The `Space` function returns a string of spaces as indicated by the number in the parentheses. The `Chr` function returns a string containing the character associated with the specified character code.

Other control characters you may need to use when writing your VBA procedures include:

Tab	<code>Chr(9)</code>
Linefeed	<code>Chr(10)</code>
Carriage Return	<code>Chr(13)</code>
Space	<code>Chr(32)</code>

## RUNNING FUNCTION PROCEDURES

---

You can execute a function procedure from the Immediate window, or you can write a subroutine procedure to call the function. See Hands-On 4.2 and 4.3 for instructions on how to run the **JoinText** function procedure using these two methods.



### Hands-On 4.2 Executing a Function Procedure from the Immediate Window

This hands-on exercise requires prior completion of Hands-On 4.1.

1. Choose **View | Immediate Window** or press **Ctrl+G**, and enter the following statement:

```
?JoinText («function», «procedure»)
```

Notice that as soon as you type the opening parenthesis, Visual Basic displays the arguments that the function expects. Type the value of the first argument, enter the comma, and supply the value of the second argument. Finish by entering the closing parenthesis.

2. Press **Enter** to execute this statement from the Immediate window. When you press Enter, the string “function procedure” appears in the Immediate window.



### **Hands-On 4.3 Executing a Function Procedure from a Subroutine**

This hands-on exercise requires prior completion of Hands-On 4.1.

1. In the same module where you entered the JoinText function procedure, enter the following **EnterText** subroutine:

```
Sub EnterText()
    Dim strFirst As String, strLast As String, strFull As String
    strFirst = InputBox("Enter your first name:")
    strLast = InputBox("Enter your last name:")
    strFull = JoinText(strFirst, strLast)

    MsgBox strFull
End Sub
```

2. Place the cursor anywhere inside the code of the EnterText procedure and press **F5** to run it.

As Visual Basic executes the statements of the EnterText procedure, it uses the **InputBox** function to collect the data from the user, and then stores the data (the values of the first and last names) in the variables **strFirst** and **strLast**. These values are then passed to the JoinText function. Visual Basic substitutes the variables’ contents for the arguments of the JoinText function and assigns the result to the name of the function (JoinText). When Visual Basic returns to the EnterText procedure, it stores the function’s value in the **strFull** variable. The **MsgBox** function then displays the contents of the **strFull** variable in a message box. The result is the full name of the user (first and last name separated by a space).

**SIDE BAR** *More about Arguments*

Argument names are like variables. Each argument name refers to whatever value you provide at the time the function is called. You write a subroutine to call a function procedure. When a subroutine calls a function procedure, the required arguments are passed to the procedure as variables. Once the function does something, the result is assigned to the function name. Notice that the function procedure's name is used as if it were a variable.

## DATA TYPES AND FUNCTIONS

Like variables, functions can have types. The data type of your function's result can be a String, Integer, Long, and so forth. To specify the data type for your function's result, add the `As` keyword and the name of the desired data type to the end of the function declaration line—for example:

```
Function MultiplyIt(num1, num2) As Integer
```

If you don't specify the data type, Visual Basic assigns the default type (Variant) to your function's result. When you specify the data type for your function's result, you get the same advantages as when you specify the data type for your variables—your procedure uses memory more efficiently, and therefore runs faster.

Let's look at an example of a function that returns an integer, even though the arguments passed to it are declared as Single in a calling subroutine.



### Hands-On 4.4 Calling a Function from a Procedure

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new module.
2. Enter the following **HowMuch** subroutine in the Code window:

```
Sub HowMuch()  
    Dim num1 As Single  
    Dim num2 As Single  
    Dim result As Single  
  
    num1 = 45.33  
    num2 = 19.24  
    result = MultiplyIt(num1, num2)
```

```
    MsgBox result  
End Sub
```

3. Enter the following **MultiplyIt** function procedure in the Code window below the **HowMuch** subroutine:

```
Function MultiplyIt(num1, num2) As Integer  
    MultiplyIt = num1 * num2  
End Function
```

4. Click anywhere within the **HowMuch** procedure and press **F5** to run it.

Because the values stored in the variables `num1` and `num2` are not whole numbers, you may want to assign the `Integer` type to the result of the function to ensure that the result of the multiplication is a whole number. If you don't assign the data type to the `MultiplyIt` function's result, the `HowMuch` procedure will display the result in the data type specified in the declaration line of the `result` variable. Instead of 872, the result of the multiplication will be 872.1492.

To make the `MultiplyIt` function more useful, instead of hard-coding the values to be used in the multiplication, you can pass different values each time you run the procedure by using the `InputBox` function.

5. Take a few minutes to modify the `HowMuch` procedure on your own, following the example of the `EnterText` subroutine that was created in Hands-On 4.3.  
6. To pass a specific value from a function to a subroutine, assign the value to the function name. For example, the `NumOfDays` function shown here passes the value of 7 to the subroutine `DaysInAWeek`.

```
Function NumOfDays()  
    NumOfDays = 7  
End Function  
  
Sub DaysInAWeek()  
    MsgBox "There are " & NumOfDays & " days in a week."  
End Sub
```

---

**SIDE BAR** ***Subroutines or Functions: Which Should You Use?***

Create a subroutine when you:

- Want to perform some actions
- Want to get input from the user
- Want to display a message on the screen

Create a function when you:

- Want to perform a simple calculation more than once
  - Must perform complex computations
  - Must call the same block of instructions more than once
  - Want to check whether a certain expression is true or false
- 

## PASSING ARGUMENTS BY REFERENCE AND BY VALUE

In some procedures, when you pass arguments as variables, Visual Basic can suddenly change the value of the variables. To ensure that the called function procedure does not alter the value of the passed arguments, you should precede the name of the argument in the function's declaration line with the `ByVal` keyword. Let's practice this in the following example.



### Hands-On 4.5 Passing Arguments to Subroutines and Functions

- In the Visual Basic Editor window, choose **Insert | Module** to add a new module.
- In the Code window, type the following **ThreeNumbers** subroutine and the **MyAverage** function procedure:

```
Sub ThreeNumbers()
    Dim num1 As Integer, num2 As Integer, num3 As Integer
    num1 = 10
    num2 = 20
    num3 = 30

    MsgBox MyAverage(num1, num2, num3)
    MsgBox num1
    MsgBox num2
    MsgBox num3
End Sub

Function MyAverage(ByVal num1, ByVal num2, ByVal num3)
    num1 = num1 + 1
    MyAverage = (num1 + num2 + num3) / 3
End Function
```

- Click anywhere within the **ThreeNumbers** procedure and press **F5** to run it. The **ThreeNumbers** procedure assigns values to three variables, and then calls

the MyAverage function to calculate and return the average of the numbers stored in these variables. The function's arguments are the names of the variables: `num1`, `num2`, and `num3`. Notice that all variable names are preceded with the `ByVal` keyword. Also, notice that prior to the calculation of the average, the MyAverage function changes the value of the `num1` variable. Inside the function procedure, the `num1` variable equals 11 ( $10 + 1$ ). Therefore, when the function passes the calculated average to the ThreeNumbers procedure, the `MsgBox` function displays the result as 20.333333333333 and not 20, as expected. The next three functions show the contents of each of the variables. The values stored in these variables are the same as the original values assigned to them: 10, 20, and 30.

What will happen if you omit the `ByVal` keyword in front of the `num1` argument in the MyAverage function's declaration line? The function's result will still be the same, but the content of the `num1` variable displayed by the `MsgBox` is now 11. The MyAverage function has not only returned an unexpected result (20.333333333333 instead of 20), but also modified the original data stored in the `num1` variable. To prevent Visual Basic from permanently changing the values supplied to the function, use the `ByVal` keyword.

#### SIDE BAR *Know Your Keywords: ByRef and ByVal*

Because any of the variables passed to a function procedure (or a subroutine) can be changed by the receiving procedure, it is important to know how to protect the original value of a variable. Visual Basic has two keywords that give or deny the permission to change the contents of a variable: `ByRef` and `ByVal`.

By default, Visual Basic passes information to a function procedure (or a subroutine) by reference (`ByRef` keyword), referring to the original data specified in the function's argument at the time the function is called. So, if the function alters the value of the argument, the original value is changed. You will get this result if you omit the `ByVal` keyword in front of the `num1` argument in the MyAverage function's declaration line. If you want the function procedure to change the original value, you don't need to explicitly insert the `ByRef` keyword because passed variables default to `ByRef`.

When you use the `ByVal` keyword in front of an argument name, Visual Basic passes the argument by value, which means that Visual Basic makes a copy of the original data. This copy is then passed to a function. If the function changes the value of an argument passed by value, the original data does not change—only the copy changes. That's why when the MyAverage function changed the value of the `num1` argument, the original value of the `num1` variable remained the same.

## USING OPTIONAL ARGUMENTS

---

At times, you may want to supply an additional value to a function. Let's say you have a function that calculates the price of a meal per person. Sometimes, however, you'd like the function to perform the same calculation for a group of two or more people. To indicate that a procedure argument isn't always required, precede the name of the argument with the `Optional` keyword. Arguments that are optional come at the end of the argument list, following the names of all the required arguments. Optional arguments must always be the Variant data type. This means that you can't specify the optional argument's type by using the `As` keyword.

In the preceding section, you created a function to calculate the average of three numbers. Suppose that sometimes you would like to use this function to calculate the average of two numbers. You could define the third argument of the `MyAverage` function as optional. To preserve the original `MyAverage` function, let's create the `Avg` function to calculate the average for two or three numbers.



### Hands-On 4.6 Using Optional Arguments

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new module.
2. Type the following `Avg` function procedure in the Code window:

```
Function Avg(num1, num2, Optional num3)
    Dim totalNums As Integer

    totalNums = 3
    If IsMissing(num3) Then
        num3 = 0
        totalNums = totalNums - 1
    End If
    Avg = (num1 + num2 + num3) / totalNums
End Function
```

3. Call this function from the Immediate window by entering the following instruction and pressing **Enter**:

```
?Avg(2, 3)
```

As soon as you press Enter, Visual Basic displays the result: 2.5.

4. Now, type the following instruction that passes the optional argument and press **Enter**:

```
?Avg(2, 3, 5)
```

This time the result is: 3.333333333333.

As you've seen, the Avg function is used to calculate the average of two or three numbers. You decide what values and how many values (two or three) you want to average. When you start typing the values for the function's arguments in the Immediate window, Visual Basic displays the name of the optional argument enclosed in square brackets.

Let's take a few minutes to analyze the Avg function. This function can take up to three arguments. Arguments `num1` and `num2` are required. Argument `num3` is optional. Notice that the name of the optional argument is preceded by the `Optional` keyword. The optional argument is listed at the end of the argument list. Because the types of the `num1`, `num2`, and `num3` arguments are not declared, Visual Basic treats all three arguments as Variants.

Inside the function procedure, the `totalNums` variable is declared as an Integer and then assigned a beginning value of 3. Because the function has to be capable of calculating an average of two or three numbers, the handy built-in function `IsMissing` checks for the number of supplied arguments. If the third (optional) argument is not supplied, the `IsMissing` function puts the value of zero (0) in its place and deducts the value of 1 from the value stored in the `totalNums` variable. Hence, if the optional argument is missing, `totalNums` is 2. The next statement calculates the average based on the supplied data, and the result is assigned to the name of the function.

## USING THE ISMISSING FUNCTION

---

The `IsMissing` function called from within Hands-On 4.6 allows you to determine whether the optional argument was supplied. This function returns the logical value of True if the third argument is not supplied and returns False when the third argument is given. The `IsMissing` function is used here with the decision-making statement `If...Then` (discussed in Chapter 5). If the `num3` argument is missing (`IsMissing`), then Visual Basic supplies a zero (0) for the value of the third argument (`num3 = 0`), and reduces the value stored in the argument `totalNums` by 1 (`totalNums = totalNums - 1`).

## VBA BUILT-IN FUNCTIONS FOR USER INTERACTION

---

VBA comes with numerous built-in functions that can be looked up in the Visual Basic online help. To access an alphabetical listing of all VBA functions in Access use these links:

<https://docs.microsoft.com/en-us/office/vba/access/concepts/criteria-expressions/functions-alphabetical-list>

<https://docs.microsoft.com/en-us/office/vba/Language/Reference/functions-visual-basic-for-applications>

Each function is described in detail and is often illustrated with a code fragment or a complete function procedure that shows how to use it in a specific context. After completing this chapter, be sure to browse through the built-in functions to familiarize yourself with their names and usage.

The following link will bring up the Office VBA language reference:

<http://msdn.microsoft.com/en-us/library/office/gg264383.aspx>

One of the features of a good program is its interaction with the user. When you work with Access, you interact with the application by using various dialog boxes, such as message boxes and input boxes. When you write your own procedures, you can use the `MsgBox` function to inform users about an unexpected error or the result of a specific calculation. So far you have seen a simple implementation of this function.

In the next section, you will find out how to control the appearance of your message. Then you will learn how to get information from the user with the `InputBox` function.

### Using the `MsgBox` Function

---

The `MsgBox` function you have used thus far was limited to displaying a message to the user in a simple, one-button dialog box. You closed the message box by clicking the OK button or pressing the Enter key. You can create a simple message box by following the `MsgBox` function name with the text of the message enclosed in quotation marks. In other words, to display the message “The procedure is complete.” you use the following statement:

```
MsgBox "The procedure is complete."
```

You can try this instruction by entering it in the Immediate window. When you press Enter, Visual Basic displays the message box shown in Figure 4.1.

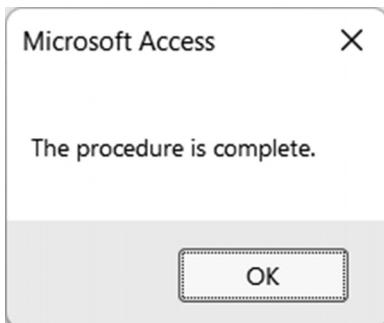


FIGURE 4.1 To display a message to the user, place the text as the argument of the `MsgBox` function.

The `MsgBox` function allows you to use other arguments that make it possible to determine the number of buttons that should be available in the message box or to change the title of the message box from the default. You can also assign your own help topic. The syntax of the `MsgBox` function is shown here.

```
MsgBox (prompt [, buttons] [, title], [, helpfile, context])
```

Notice that while the `MsgBox` function has five arguments, only the first one, `prompt`, is required. The arguments listed in square brackets are optional.

When you enter a long text string for the `prompt` argument, Visual Basic decides how to break the text so it fits the message box. Let's do some exercises in the Immediate window to learn various text formatting techniques.



### Hands-On 4.7 Formatting the Message Box

1. In the Visual Basic Editor window, activate the Immediate window and enter the following instruction. Be sure to enter the entire text string on one line, and then press **Enter**.

```
MsgBox "All done. Now open the File Explorer and locate ""Test.doc"" document in your working folder."
```

As soon as you press **Enter**, Visual Basic shows the resulting dialog box (see Figure 4.2). If you get a compile error, click **OK**. Then make sure that the name of the file is surrounded by double quotation marks (""Test.doc"").

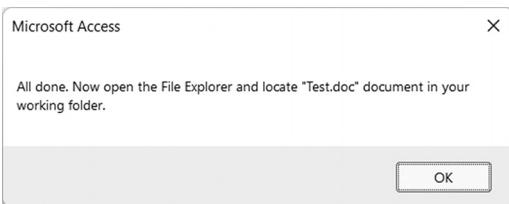


FIGURE 4.2 This long message will look more appealing to the user when you take the text formatting into your own hands.

When the text of your message is particularly long, you can break it into several lines using the VBA `Chr` function. The `Chr` function's argument is a number from 0 to 255, which returns a character represented by this number. For example, `Chr(13)` returns a carriage return character (this is the same as pressing the Enter key), and `Chr(10)` returns a linefeed character (this is useful for adding spacing between the text lines).

2. Modify the instruction entered in the previous step in the following way and make sure it stays on the same line in the Immediate window:

```
MsgBox "All done." & Chr(13) & "Now open the File Explorer" &  
Chr(13) & "and locate ""Test.doc"" document" & Chr(13) & "in  
your working folder."
```

Your result should look like Figure 4.3.

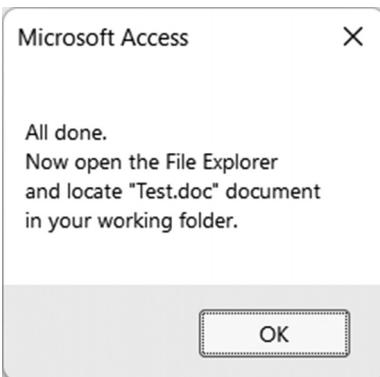


FIGURE 4.3 You can break a long text string into several lines by using the `Chr(13)` function.

You must surround each text fragment with quotation marks. Quoted text embedded in a text string requires an additional set of quotation marks, as in `""Test.doc""`. The `Chr(13)` function indicates a place where you'd like to start a new line. The concatenate character (&) is used to combine strings. When you enter exceptionally long text messages on one line, it's easy to make

a mistake. An underscore (\_) is a special line continuation character in VBA that allows you to break a long VBA statement into several lines. Unfortunately, the line continuation character cannot be used in the Immediate window. A better place to try out various formatting of your long strings for the MsgBox function is within a VBA procedure.

3. Add a new module by choosing **Insert | Module**.
4. In the Code window, enter the following **MyMessage** subroutine. Be sure to precede each line continuation character (\_) with a space.

```
Sub MyMessage()

    MsgBox "All done." & Chr(13) _
        & "Now open the File Explorer" & Chr(13) _
        & "and locate ""Test.doc"" document" & Chr(13) _
        & "in your working folder."

End Sub
```

5. Position the insertion point within the code of the MyMessage procedure and press **F5** to run it. Remember you can also run a procedure by choosing **Run | Run Sub\UserForm**.

When you run the MyMessage procedure, Visual Basic displays the same message as the one illustrated earlier in Figure 4.3.

As you can see, the text entered on several lines is more readable, and the code is easier to maintain. To improve the readability of your message, you may want to add more spacing between the text lines by including blank lines. To do this, use two `Chr(13)` functions, as shown in the following step.

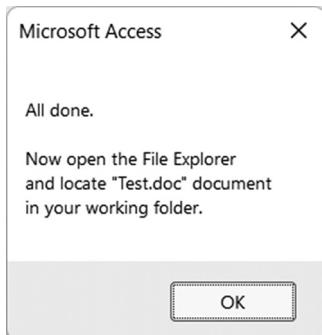
6. Enter the following **MyMessage2** procedure:

```
Sub MyMessage2()

    MsgBox "All done." & Chr(13) & Chr(13) _
        & "Now open the File Explorer" & Chr(13) _
        & "and locate ""Test.doc"" document" & Chr(13) _
        & "in your working folder."

End Sub
```

7. Position the insertion point within the code of the MyMessage2 procedure and press F5 to run it. The result should look like Figure 4.4.



**FIGURE 4.4** You can increase the readability of your message by increasing spacing between selected text lines.

Now that you have mastered the text formatting techniques, let's take a closer look at the next argument of the `MsgBox` function. Although the `buttons` argument is optional, it is frequently used. The `buttons` argument specifies how many and what types of buttons you want to appear in the message box. This argument can be a constant or a number (see Table 4.1). If you omit this argument, the resulting message box contains only the OK button, as you've seen in the preceding examples.

**TABLE 4.1** The `MsgBox` `buttons` argument settings.

Constant	Value	Description
<b>Button settings</b>		
<code>vbOKOnly</code>	0	Displays only an OK button. This is the default.
<code>vbOKCancel</code>	1	OK and Cancel buttons
<code>vbAbortRetryIgnore</code>	2	Abort, Retry, and Ignore buttons
<code>vbYesNoCancel</code>	3	Yes, No, and Cancel buttons
<code>vbYesNo</code>	4	Yes and No buttons
<code>vbRetryCancel</code>	5	Retry and Cancel buttons
<b>Icon settings</b>		
<code>vbCritical</code>	16	Displays the Critical Message icon
<code>vbQuestion</code>	32	Displays the Question Message icon
<code>vbExclamation</code>	48	Displays the Warning Message icon
<code>vbInformation</code>	64	Displays the Information Message icon

Constant	Value	Description
<b>Default button settings</b>		
vbDefaultButton1	0	The first button is default.
vbDefaultButton2	256	The second button is default.
vbDefaultButton3	512	The third button is default.
vbDefaultButton4	768	The fourth button is default.
<b>Message box modality</b>		
vbApplicationModal	0	The user must respond to the message before continuing to work in the current application.
vbSystemModal	4096	On Win16 systems, this constant is used to prevent the user from interacting with any other window until he or she dismisses the message box. On Win32 systems, this constant works like the vbApplicationModal constant with the following exception: The message box always remains on top of any other programs you may have running.
<b>Other MsgBox display settings</b>		
vbMsgBoxHelpButton	16384	Adds the Help button to the message box
vbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window
vbMsgBoxRight	524288	Text is right-aligned.
vbMsgBoxRtlReading	1048576	Text appears as right-to-left reading on Hebrew and Arabic systems.

When should you use the `buttons` argument? Suppose you want the user of your procedure to respond to a question with Yes or No. Your message box will then require two buttons. If a message box includes more than one button, one of them is considered a default button. When the user presses Enter, the default button is selected automatically.

Because you can display various types of messages (critical, warning, information), you can visually indicate the importance of the message by including the graphical representation (icon). In addition to the type of message, the `buttons` argument can include a setting to determine whether the message box must be closed before the user switches to another application. It's quite possible that the user may want to switch to another program or perform another task before he responds to the question posed in your message box. If the message box is application modal (`vbApplicationModal`), then the user must close the message box before continuing to use your application.

For example, consider the following message box:

```
MsgBox "How are you?", vbOKOnly + vbApplicationModal, "Please Close Me"
```

If you type the preceding statement in the Immediate window and press Enter, a message box will pop up and you won't be able to work with your currently open Access database application until you respond to the message box.

On the other hand, if you want to keep the message box visible while the user works with other open applications, you must include the `vbSystemModal` setting in the `buttons` argument, like this:

```
MsgBox "How are you?", vbOKOnly + vbSystemModal, "System Modal"
```

**NOTE**

*Use the `vbSystemModal` constant when you want to ensure that your message box is always visible (not hidden behind other windows).*

The `buttons` argument settings are divided into five groups: button settings, icon settings, default button settings, message box modality, and other `MsgBox` display settings (see Table 4.1). Only one setting from each group can be included in the `buttons` argument. To create a `buttons` argument, you can add up the values for each setting you want to include. For example, to display a message box with two buttons (Yes and No), the question mark icon, and the No button as the default button, look up the corresponding values in Table 4.1, and add them up. You should arrive at 292 (4 + 32 + 256).

To see the message box using the calculated message box argument, enter the following statement in the Immediate window:

```
MsgBox "Do you want to proceed?", 292
```

The resulting message box is shown in Figure 4.5.

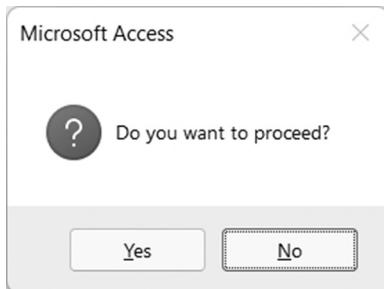


FIGURE 4.5 You can specify the number of buttons to include, their text, and an icon in the message box by using the optional `buttons` argument.

When you derive the `buttons` argument by adding up the constant values, your procedure becomes less readable. There's no reference table where you can check the hidden meaning of 292. To improve the readability of your `MsgBox` function, it's better to use the constants instead of their values. For example, enter the following revised statement in the Immediate window:

```
MsgBox "Do you want to proceed?", vbYesNo + vbQuestion + vbDefaultButton2
```

The preceding statement produces the result shown in Figure 4.5. The following example shows how to use the `buttons` argument inside a Visual Basic procedure.



### Hands-On 4.8 Using the `MsgBox` Function with Arguments

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new module.
2. In the Code window, enter the `MsgYesNo` subroutine shown here:

```
Sub MsgYesNo()
    Dim question As String
    Dim myButtons As Integer

    question = "Do you want to open a new report?"
    myButtons = vbYesNo + vbQuestion + vbDefaultButton2
    MsgBox question, myButtons
End Sub
```

3. Run the `MsgYesNo` procedure by pressing F5.

In this subroutine, the `question` variable stores the text of your message. The settings for the `buttons` argument are placed in the `myButtons` variable. Instead of using the names of constants, you can use their values, as in the following:

```
myButtons = 4 + 32 + 256
```

The `question` and `myButtons` variables are used as arguments for the `MsgBox` function. When you run the procedure, you see a result similar to the one shown in Figure 4.5. Note that the No button is selected, indicating that it's the default button for this dialog box. If you press Enter, Visual Basic removes the message box from the screen. Nothing happens because your procedure does not have any instructions following the `MsgBox` function. To change the default button, use the `vbDefaultButton1` setting instead.

The third argument of the `MsgBox` function is `title`. While this is also an optional argument, it's very handy because it allows you to create procedures that don't provide visual clues to the fact that you programmed them with Access. Using this argument, you can set the titlebar of your message box to any text you want.

Suppose you want the `MsgYesNo` procedure to display the text "New report" in its title. The following `MsgYesNo2` procedure demonstrates the use of the `title` argument.

```
Sub MsgYesNo2()
    Dim question As String
    Dim myButtons As Integer
    Dim myTitle As String

    question = "Do you want to open a new report?"
    myButtons = vbYesNo + vbQuestion + vbDefaultButton2
    myTitle = "New report"
    MsgBox question, myButtons, myTitle
End Sub
```

The text for the `title` argument is stored in the `myTitle` variable. If you don't specify the value for the `title` argument, Visual Basic displays the default text "Microsoft Access." Notice that the arguments are listed in the order determined by the `MsgBox` function.

If you would like to list the arguments in any order, you must precede the value of each argument with its name, as shown here:

```
MsgBox title:=myTitle, prompt:=question, buttons:=myButtons
```

The last two `MsgBox` arguments, `helpfile` and `context`, are used by more advanced programmers who are experienced with using help files in the Windows environment. The `helpfile` argument indicates the name of a special help file that contains additional information you may want to display to your VBA application user. When you specify this argument, the Help button will be added to your message box. When you use the `helpfile` argument, you must also use the `context` argument. This argument indicates which help subject in the specified help file you want to display. Suppose `HelpX.hlp` is the help file you created and 55 is the context topic you want to use. To include this information in your `MsgBox` function, you would use the following instruction:

```
MsgBox title:=myTitle, _
    prompt:=question, _
    buttons:=myButtons, _
```

```
helpfile:= "HelpX.hlp", _
context:=55
```

The preceding is a single VBA statement broken down into several lines using the line continuation character.

### ***Returning Values from the MsgBox Function***

When you display a simple message box dialog with one button, clicking the OK button or pressing the Enter key removes the message box from the screen. However, when the message box has more than one button, your procedure should detect which button was pressed. To do this, you must save the result of the message box in a variable. Table 4.2 lists values that the `MsgBox` function returns.

**TABLE 4.2** Values returned by the `MsgBox` function.

Button Selected	Constant	Value
OK	<code>vbOK</code>	1
Cancel	<code>vbCancel</code>	2
Abort	<code>vbAbort</code>	3
Retry	<code>vbRetry</code>	4
Ignore	<code>vbIgnore</code>	5
Yes	<code>vbYes</code>	6
No	<code>vbNo</code>	7

The `MsgYesNo3` procedure in Hands-On 4.9 is a revised version of `MsgYesNo2`. It demonstrates how to store the user's response in a variable.



### **Hands-On 4.9 Returning Values from the MsgBox Function**

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new module.
2. In the Code window, enter the following code of the `MsgYesNo3` procedure:

```
Sub MsgYesNo3()
    Dim question As String
    Dim myButtons As Integer
    Dim myTitle As String
    Dim myChoice As Integer

    question = "Do you want to open a new report?"
    myButtons = vbYesNo + vbQuestion + vbDefaultButton2
```

```
myTitle = "New report"  
myChoice = MsgBox(question, myButtons, myTitle)  
MsgBox myChoice  
End Sub
```

3. Position the insertion point within the `MsgYesNo3` procedure and press F5 to run it.

In this procedure, you assigned the result of the `MsgBox` function to the variable `myChoice`. Notice that the arguments of the `MsgBox` function are now listed in parentheses:

```
myChoice = MsgBox(question, myButtons, myTitle)
```

When you run the `MsgYesNo3` procedure, a two-button message box is displayed. By clicking on the Yes button, the statement `MsgBox myChoice` displays the number 6. When you click the No button, the number 7 is displayed.

---

**SIDE BAR** ***MsgBox Function—With or without Parentheses?***

Use parentheses around the `MsgBox` function argument list when you want to use the result returned by the `MsgBox` function. By listing the function's arguments without parentheses, you tell Visual Basic that you want to ignore the function's result. Most likely, you will want to use the function's result when the message box contains more than one button.

---

## Using the `InputBox` Function

---

The `InputBox` function displays a dialog box with a message that prompts the user to enter data. This dialog box has two buttons: OK and Cancel. When you click OK, the `InputBox` function returns the information entered in the text box. When you select Cancel, the function returns the empty string (""). The syntax of the `InputBox` function is as follows:

```
InputBox(prompt [, title] [, default] [, xpos] [, ypos]  
[, helpfile, context])
```

The first argument, `prompt`, is the text message you want to display in the dialog box. Long text strings can be entered on several lines by using the `Chr(13)` or `Chr(10)` functions. (See examples of using the `MsgBox` function earlier in this chapter.) All the remaining `InputBox` arguments are optional.

The second argument, `title`, allows you to change the default title of the dialog box. The default value is “Microsoft Access.”

The third argument of the `InputBox` function, `default`, allows the display of a default value in the text box. If you omit this argument, the empty text box is displayed.

The following two arguments, `xpos` and `ypos`, let you specify the exact position where the dialog box should appear on the screen. If you omit these arguments, the input box appears in the middle of the current window. The `xpos` argument determines the horizontal position of the dialog box from the left edge of the screen. When omitted, the dialog box is centered horizontally. The `ypos` argument determines the vertical position from the top of the screen. If you omit this argument, the dialog box is positioned vertically approximately one-third of the way down the screen. Both `xpos` and `ypos` are measured in special units called *twips*. One twip is the equivalent of approximately 0.0007 inches.

The last two arguments, `helpfile` and `context`, are used in the same way as the corresponding arguments of the `MsgBox` function discussed earlier in this chapter.

Now that you know the meaning of the `InputBox` arguments, let's see some examples of using this function.



### Hands-On 4.10 Using the InputBox Function

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new module.
2. In the Code window, type the following **Informant** subroutine procedure:

```
Sub Informant()
    InputBox prompt:="Enter your place of birth:" & Chr(13) _
        & " (e.g., Boston, Great Falls, etc.)"
End Sub
```

3. Position the insertion point within the `Informant` procedure and press **F5** to run it.

This procedure displays a dialog box with two buttons. The input prompt is displayed on two lines (see Figure 4.6). Similar to using the `MsgBox` function you may want to store the result of the `InputBox` function in a variable.



FIGURE 4.6 A dialog box generated by the `Informant` procedure.

4. Now, in the same module, enter the following code of the **Informant2** procedure:

```
Sub Informant2()
    Dim myPrompt As String
    Dim town As String

    Const myTitle = "Enter data"
    myPrompt = "Enter your place of birth:" & Chr(13) _
    & "(e.g., Boston, Great Falls, etc.)"
    town = InputBox(myPrompt, myTitle)

    MsgBox "You were born in " & town & ".", , "Your response"
End Sub
```

5. Position the insertion point within the **Informant2** procedure and press F5 to run it.

Notice that the **Informant2** procedure assigns the result of the `InputBox` function to the `town` variable.

This time, the arguments of the `InputBox` function are listed in parentheses. Parentheses are required if you want to use the result of the `InputBox` function later in your procedure. The **Informant2** subroutine uses a constant to specify the text to appear in the titlebar of the dialog box. Because the constant value remains the same throughout the execution of your procedure, you can declare the input box title as a constant. However, if you'd rather use a variable, you still can.

When you run a procedure using the `InputBox` function, the dialog box generated by this function always appears in the same area of the screen. To change the location of the dialog box, you must supply the `xpos` and `ypos` arguments, which were explained earlier.

6. To display the dialog box in the top left-hand corner of the screen, modify the `InputBox` function in the **Informant2** procedure as follows:

```
town = InputBox(myPrompt, myTitle, , 1, 200)
```

Notice that the argument `myTitle` is followed by two commas. The second comma marks the position of the omitted `default` argument. The next two arguments determine the horizontal and vertical position of the dialog box. If you omit the second comma after the `myTitle` argument, Visual Basic will use the number 1 as the value of the `default` argument. If you precede the values of arguments by their names (e.g., `prompt:=myPrompt, title:=myTitle, xpos:=1, ypos:=200`), you won't have to remember to insert a comma in the place of each omitted argument.

What will happen if, instead of the name of a town, you enter a number? Because users often supply incorrect data in the input box, your procedure must verify that the data the user entered can be used in further data calculations or manipulations. The `InputBox` function itself does not provide a facility for data validation. To validate user input, you must use other VBA instructions, which are discussed in Chapter 5, “Adding Decisions to Your Access VBA Programs.”

## CONVERTING DATA TYPES

---

The result of the `InputBox` function is always a string. So, if a user enters a number, its *string* value must be converted to a *numeric* value before your procedure can use the number in mathematical computations. Visual Basic can automatically convert many values from one data type to another.



### Hands-On 4.11 Converting Data Types

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new module.
2. In the Code window, enter the following **AddTwoNums** procedure:

```
Sub AddTwoNums()
    Dim myPrompt As String
    Dim value1 As String
    Dim mySum As Single

    Const myTitle = "Enter data"

    myPrompt = "Enter a number:"
    value1 = InputBox(myPrompt, myTitle, 0)
    If value1 = "" Then value1 = 0
    mySum = value1 + 2

    MsgBox mySum & " (" & value1 & " + 2)"
End Sub
```

3. Place the cursor anywhere inside the code of the **AddTwoNums** procedure and press **F5** to run it.

This procedure displays the dialog box shown in Figure 4.7. Notice that this dialog box has two special features that are obtained by using the `InputBox` function’s optional arguments: `title` and `default`. Instead of the default

title “Microsoft Access,” the dialog box displays a text string as defined by the contents of the `myTitle` constant. The zero (0) entered as the default value in the edit box suggests that the user enter a number instead of text. Once the user provides the data and clicks OK, the input is assigned to the variable `value1`.

```
value1 = InputBox(myPrompt, myTitle, 0)
```



FIGURE 4.7 To suggest that the user enter a specific type of data, you may want to provide a default value in the edit box.

The data type of the variable `value1` is String. You can check the data type easily if you follow the preceding instruction with this statement:

```
MsgBox varType(value1)
```

When Visual Basic runs this line, it will display a message box with the number 8. Recall that this number represents the String data type. The next line,

```
mySum = value1 + 2
```

adds 2 to the user’s input and assigns the result of the calculation to the variable `mySum`. Because the `value1` variable’s data type is String, Visual Basic goes to work behind the scenes to perform the data type conversion. Visual Basic has the brains to understand the need for conversion. Without it, the two incompatible data types (text and number) would generate a Type Mismatch error. Also, if the user clicks Cancel without inputting a value, the same Type Mismatch error will occur. To prevent this from happening, the procedure checks for the contents of the `value1` variable using the following decision statement:

```
If value1 = "" Then value1 = 0
```

The above statement will assign zero (0) to the `value1` variable if it is empty. We will discuss decision making statements in the next chapter.

The procedure ends with the `MsgBox` function displaying the result of the calculation and showing the user how the total was derived.

---

**SIDE BAR** *Define a Constant*

---

To ensure that all the title bars in a VBA procedure display the same text, assign the title text to a constant. By doing so, you will save yourself the time of typing the title text in more than one place.

---

---

## SUMMARY

---

In this chapter, you learned the difference between subroutine procedures that perform actions and function procedures that return values. You saw examples of function procedures called from another Visual Basic procedure. You learned how to pass arguments to functions and how to determine the data type of a function's result. You increased your repertoire of VBA keywords with the `ByVal`, `ByRef`, and `Optional` keywords.

After working through this chapter, you should be able to create some custom functions of your own that are suited to your specific needs. You should also be able to interact easily with your users by employing the `MsgBox` and `InputBox` functions.

In the next chapter, you learn how to make decisions in your VBA programs.



# Chapter 5

# ADDING DECISIONS TO YOUR ACCESS VBA PROGRAMS

Visual Basic for Applications offers special statements called conditional statements, or “control structures,” which allow you to include decision points in your procedures. In a conditional expression, a relational operator (see Table 5.1), a logical operator (see Table 5.2), or a combination of both evaluates the expression to determine whether it is true or false. If the answer is true, the procedure executes a specified block of instructions. If the answer is false, the procedure either executes a different block of instructions or simply doesn’t do anything. In this chapter, you will learn how to use these VBA conditional statements to alter the flow of your program.

## **RELATIONAL AND LOGICAL OPERATORS**

---

You can make decisions in your VBA procedures by using conditional expressions inside the special control structures. A *conditional expression* is an expression that uses a relational operator (see Table 5.1), a logical operator (see Table 5.2), or a combination of both. When Visual Basic encounters a conditional expression in your program, it evaluates the expression to determine whether it is true or false.

**TABLE 5.1** Relational operators in VBA.

Operator	Description
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

**TABLE 5.2** Logical operators in VBA.

Operator	Description
AND	All conditions must be true before an action can be taken.
OR	At least one of the conditions must be true before an action can be taken.
NOT	If a condition is true, NOT makes it false. If a condition is false, NOT makes it true.

### SIDE BAR Boolean Expressions

Conditional expressions and logical operators are also known as *Boolean*. George Boole was a nineteenth-century British mathematician who made significant contributions to the evolution of computer programming. Boolean expressions can be evaluated as true or false.

For example,

One meter equals 10 inches.	False
Two is less than three.	True

## IF...THEN STATEMENT

The simplest way to get some decision making into your VBA procedure is by using the If...Then statement. Suppose you want to choose an action depending on a condition. You can use the following structure:

`If condition Then statement`

For example, a quiz procedure might ask the user to guess the number of weeks in a year. If the user's response is other than 52, the procedure should display the message "Try Again."

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 5.1 Using the If...Then Statement

1. Start Access and create a new database named **Chap05.accdb** in your **C:\VBAAccess2021\_ByExample\_Primer** folder.
2. Once your new database is opened, choose **Database Tools | Visual Basic** or press **Alt+F11** to switch to the Visual Basic Editor window.
3. Choose **Insert | Module** to add a new standard module.
4. In the **Module1** Code window, enter the following **SimpleIfThen** procedure:

```
Sub SimpleIfThen()
    Dim weeks As String

    weeks = InputBox("How many weeks are in a year:", "Quiz")
    If weeks <> 52 Then MsgBox "Try Again"
End Sub
```

The **SimpleIfThen** procedure stores the user's answer in the **weeks** variable. The variable's value is then compared with the number 52. If the result of the comparison is true (i.e., if the value stored in the variable **weeks** is not equal to 52), Visual Basic will display the message "Try Again."

5. Run the **SimpleIfThen** procedure and enter a number other than 52.
6. Rerun the **SimpleIfThen** procedure and enter the number **52**. When you enter the correct number of weeks, Visual Basic does nothing. The procedure ends. It would be nice to also display a message when the user guesses right.
7. Enter the following instruction on a separate line before the **End Sub** keywords:

```
If weeks = 52 Then MsgBox "Congratulations!"
```

8. Run the **SimpleIfThen** procedure again and enter the number **52**. When you enter the correct answer, Visual Basic does not execute the "Try Again" statement. When the procedure is executed, the statement to the right of the **Then** keyword is ignored if the result from evaluating the supplied condition is false. As you recall, a VBA procedure can call another procedure. Let's see if it can also call itself.
9. Modify the first **If** statement in the **SimpleIfThen** procedure as follows:

```
If weeks <> 52 Then MsgBox "Try Again" : SimpleIfThen
```

We added a colon and the name of the SimpleIfThen procedure to the end of the existing If...Then statement. If you enter the incorrect answer, you'll see a message. After clicking the OK button in the message box, you'll get another chance to supply the correct answer. You'll be able to keep on guessing for a long time. In fact, you won't be able to exit the procedure gracefully until you've supplied the correct answer. After clicking Cancel, you'll have to deal with the unfriendly "Type Mismatch" error message. For now (until you learn other ways of handling errors in VBA), let's create a revised procedure as follows:

```
Sub SimpleIfThen_Revised()
    Dim weeks As String

    On Error GoTo VeryEnd

    weeks = InputBox("How many weeks are in a year:", "Quiz")
    If weeks <> 52 Then MsgBox "Try Again" : SimpleIfThen_Revised
    If weeks = 52 Then MsgBox "Congratulations!"

    VeryEnd:
End Sub
```

If Visual Basic encounters an error, it will jump to the VeryEnd label placed at the end of the procedure. The statements placed between On Error GoTo VeryEnd and the VeryEnd labels are ignored. Later in this chapter you will find other examples of trapping errors in your VBA procedures.

10. Run the SimpleIfThen\_Revised procedure a few times by supplying incorrect answers. The error trap that you added to your procedure will allow you to quit guessing without having to deal with the ugly error message.

## MULTILINE IF...THEN STATEMENT

---

Sometimes you may want to perform several actions when the condition is true. Although you could add other statements on the same line by separating them with colons, your code will look clearer if you use the multiline version of the If...Then statement, as shown here:

```
If condition Then
    statement1
    statement2
    statementN
End If
```

For example, let's create another version of SimpleIfThen\_Revised procedure to include additional statements.



## Hands-On 5.2 Using the Multiline If...Then Statement

1. Insert a new module and enter the following SimpleIfThen2 procedure:

```
Sub SimpleIfThen_Revised2()
    Dim weeks As String
    Dim response As String

    On Error GoTo VeryEnd
    weeks = InputBox("How many weeks are in a year?", "Quiz")
    If weeks <> 52 Then
        response = MsgBox("This is incorrect. Would you like " _
            & " to try again?", vbYesNo + vbInformation _
            + vbDefaultButton1, _
            "Continue Quiz?")
        If response = vbYes Then
            Call SimpleIfThen_Revised2
        End If
    End If

VeryEnd:
End Sub
```

2. Run the SimpleIfThen\_Revised2 procedure and enter any number other than 52.

In this example, the statements between the first `Then` and the first `End If` keywords don't get executed if the variable `weeks` is equal to 52. Notice that the multiline `If...Then` statement must end with the keywords `End If`. How does Visual Basic decides? Simply put, it evaluates the condition it finds between the `If...Then` keywords.

### SIDE BAR *Two Formats of the If...Then Statement*

The `If...Then` statement has two formats: a single-line format and a multiline format. The short format is good for statements that fit on one line, like:

```
If secretCode <> "01W01" Then MsgBox "Access denied"
```

Or

```
If secretCode = "01W01" Then alpha = True : beta = False
```

Notice that there is no `End If` clause in the above statements. In these examples, `secretCode`, `alpha`, and `beta` are the names of variables. In the first example, Visual Basic displays the message “Access denied” if the value of the `secretCode` variable is not equal to 01W01. In the second example, Visual Basic will set the value of the variable `alpha` to True and the value of the variable `beta` to False when the `secretCode` value is equal to 01W01. Notice that the second statement to be executed is separated from the first one by a colon. The multiline `If...Then` statement is clearer when there are more statements to be executed when the condition is true, or when the statement to be executed is extremely long.

---

## DECISIONS BASED ON MORE THAN ONE CONDITION

---

The SimpleIfThen procedure you worked with earlier evaluated only a single condition in the `If...Then` statement. This statement, however, can take more than one condition. To specify multiple conditions in an `If...Then` statement, you use the logical operators AND and OR (see Table 5.2 at the beginning of the chapter). Here is the syntax of the `If...Then` statement with the AND operator:

```
If condition1 AND condition2 Then statement
```

In this syntax, both `condition1` and `condition2` must be true for Visual Basic to execute the statement to the right of the `Then` keyword—for example:

```
If sales = 10000 AND salary < 45000 Then slsCom = sales * 0.07
```

In this example, `condition1` is `sales = 10000`, and `condition2` is `salary < 45000`.

When AND is used in the conditional expression, both conditions must be true before Visual Basic can calculate the sales commission (`slsCom`). If any of these conditions is false or both are false, Visual Basic ignores the statement after `Then`. When it's good enough to meet only one of the conditions, you should use the OR operator. Here is the syntax:

```
If condition1 OR condition2 Then statement
```

The OR operator is more flexible. Only one of the conditions must be true before Visual Basic can execute the statement following the `Then` keyword. Let's look at this example:

```
If dept = "S" OR dept = "M" Then bonus = 500
```

In this example, if at least one condition is true, Visual Basic assigns 500 to the bonus variable. If both conditions are false, Visual Basic ignores the rest of the line.

Now, let's look at a complete procedure example. Suppose you can get a 10% discount if you purchase 50 units of a product priced at \$7.00. The **IfThenAnd** procedure demonstrates the use of the AND operator.



### Hands-On 5.3 Using the If...Then...AND Statement

1. Insert a new module and enter the following **IfThenAnd** procedure in the module's Code window:

```
Sub IfThenAnd()
    Dim price As Single
    Dim units As Integer
    Dim rebate As Single

    Const strMsg1 = "To get a rebate, buy an additional "
    Const strMsg2 = "Price must equal $7.00"

    units = 234
    price = 7

    If price = 7 And units >= 50 Then
        rebate = (price * units) * 0.1
        MsgBox "The rebate is: $" & rebate
    End If

    If price = 7 And units < 50 Then
        MsgBox strMsg1 & "50 - units."
    End If

    If price <> 7 And units >= 50 Then
        MsgBox strMsg2
    End If

    If price <> 7 And units < 50 Then
        MsgBox "You didn't meet the criteria."
    End If
End Sub
```

2. Run the **IfThenAnd** procedure.

The **IfThenAnd** procedure has four **If...Then** statements that are used to evaluate the contents of two variables: **price** and **units**. The **AND** operator

between the keywords `If...Then` allows more than one condition to be tested. With the AND operator, all conditions must be true for Visual Basic to run the statements between the `Then...End If` keywords.

**SIDE BAR** *Indenting If Block Instructions*

To make the `If` blocks easier to read and understand, use indentation. Compare the following:

<code>If condition Then</code>	<code>If condition Then</code>
<code>action</code>	<code>action</code>
<code>End If</code>	<code>End If</code>

Looking at the block statement on the right side, you can easily see where the block begins and where it ends.

## IF...THEN...ELSE STATEMENT

Now you know how to display a message or take an action when one or more conditions are true or false. What should you do, however, if your procedure needs to take one action when the condition is true and another action when the condition is false? By adding the `Else` clause to the simple `If...Then` statement, you can direct your procedure to the appropriate statement depending on the result of the test.

The `If...Then...Else` statement has two formats: single-line and multiline. The single-line format is as follows:

```
If condition Then statement1 Else statement2
```

The statement following the `Then` keyword is executed if the condition is true, and the statement following the `Else` clause is executed if the condition is false—for example:

```
If sales > 5000 Then Bonus = sales * 0.05 Else MsgBox "No Bonus"
```

If the value stored in the variable `sales` is greater than 5000, Visual Basic will calculate the bonus using the following formula: `sales * 0.05`. However, if the variable `sales` is not greater than 5000, Visual Basic will display the message “No Bonus.”

The `If...Then...Else` statement should be used to decide which of two actions to perform. When you need to execute more statements when the condition is true or false, it's better to use the multiline format of the `If...Then...Else` statement:

```
If condition Then  
    statements to be executed if condition is True  
Else  
    statements to be executed if condition is False  
End If
```

Notice that the multiline (block) `If...Then...Else` statement ends with the `End If` keywords. Use the indentation as shown to make this block structure easier to read.

```
If Me.Dirty Then  
    Me!btnUndo.Enabled = True  
Else  
    Me!btnUndo.Enabled = False  
End If
```

In this example, if the condition (`Me.Dirty`) is true, Visual Basic will execute the statements between `Then` and `Else`, and will ignore the statement between `Else` and `End If`. If the condition is false, Visual Basic will omit the statements between `Then` and `Else` and will execute the statement between `Else` and `End If`. The purpose of this procedure fragment is to enable the Undo button when the data on the form has changed and keep the Undo button disabled if the data has not changed. Let's look at a procedure example.



#### Hands-On 5.4 Using the If...Then...Else Statement

1. Insert a new module and enter the following `WhatTypeOfDay` procedure in the module's Code window:

```
Sub WhatTypeOfDay()  
    Dim response As String  
    Dim question As String  
    Dim strMsg1 As String  
    Dim strMsg2 As String  
    Dim myDate As Date  
    Dim strDay As String  
  
    question = "Enter any date in the format mm/dd/yyyy:" _  
        & Chr(13) & " (e.g., 07/06/2021)"  
    strMsg1 = "weekday"  
    strMsg2 = "weekend"  
    response = InputBox(question)  
    myDate = Weekday(CDate(response))  
  
    If myDate = 1 Then strDay = "Sunday"
```

```

If myDate = 2 Then strDay = "Monday"
If myDate = 3 Then strDay = "Tuesday"
If myDate = 4 Then strDay = "Wednesday"
If myDate = 5 Then strDay = "Thursday"
If myDate = 6 Then strDay = "Friday"
If myDate = 7 Then strDay = "Saturday"

If myDate >= 2 And myDate <= 6 Then
    MsgBox strMsg1 & ":" & strDay
Else
    MsgBox strMsg2 & ":" & strDay
End If
End Sub

```

**2.** Run the `WhatTypeOfDay` procedure.

This procedure asks the user to enter any date. The user-supplied string is then converted to the Date data type with the built-in `CDate` function. Finally, the `Weekday` function converts the date into an integer that indicates the day of the week (see Table 5.3). The integer is stored in the variable `myDate`. Several conditional tests are performed to check the value of the variable `myDate` and determine what day of the week it is. To determine if it is a weekday or a weekend, we check if the variable value is greater than or equal to 2 (`>=2`) and less than or equal to 6 (`<=6`). If the result of the test is true, the user is told that the supplied date is a weekday; otherwise, the program announces that it's a weekend.

**3.** Run the procedure a few more times, each time supplying a different date. Check the Visual Basic answers against your desktop or wall calendar.

**TABLE 5.3** The `Weekday` function values.

Constant	Value
<code>vbSunday</code>	1
<code>vbMonday</code>	2
<code>vbTuesday</code>	3
<code>vbWednesday</code>	4
<code>vbThursday</code>	5
<code>vbFriday</code>	6
<code>vbSaturday</code>	7

## IF...THEN...ELSEIF STATEMENT

---

Quite often you will need to check the results of several different conditions. To join a set of `If` conditions together, you can use the `ElseIf` clause. Using the `If...Then...ElseIf` statement, you can evaluate more conditions than is possible with the `If...Then...Else` statement that was the subject of the preceding section. Here is the syntax of the `If...Then...ElseIf` statement:

```
If condition1 Then
    statements to be executed if condition1 is True
ElseIf condition2 Then
    statements to be executed if condition2 is True
ElseIf condition3 Then
    statements to be executed if condition3 is True
ElseIf conditionN Then
    statements to be executed if conditionN is True
Else
    statements to be executed if all conditions are False
End If
```

The `Else` clause is optional; you can omit it if there are no actions to be executed when all conditions are false.

### SIDE BAR *ElseIf Clause*

Your procedure can include any number of `ElseIf` statements and conditions. The `ElseIf` clause always comes before the `Else` clause. The statements in the `ElseIf` clause are executed only if the condition in this clause is true.

---

Let's look at the following procedure fragment:

```
If myNumber = 0 Then
    MsgBox "You entered zero."
ElseIf myNumber > 0 Then
    MsgBox "You entered a positive number."
ElseIf myNumber < 0 Then
    MsgBox "You entered a negative number."
End If
```

This example checks the value of the number entered by the user and stored in the variable `myNumber`. Depending on the number entered, an appropriate message (zero, positive, negative) is displayed. Notice that the `Else` clause is not used. If the result of the first condition (`myNumber = 0`) is false, Visual Basic

jumps to the next `ElseIf` statement and evaluates its condition (`myNumber > 0`). If the value is not greater than zero, Visual Basic skips to the next `ElseIf` and the condition `myNumber < 0` is evaluated.

## NESTED IF...THEN STATEMENTS

---

You can make more complex decisions in your VBA procedures by placing an `If...Then` or `If...Then...Else` statement inside another `If...Then` or `If...Then...Else` statement. Structures in which an `If` statement is contained inside another `If` block are referred to as *nested If* statements. To understand how nested `If...Then` statements work, it's time for another hands-on exercise.



### Hands-On 5.5 Using Nested If...Then Statements

1. In the database **Chap05.accdb**, create a blank form by choosing **Blank form** in the Forms section of the Create tab. When Access opens the new form in Layout view, switch to Design view.
2. Use the text box control in the Controls section of the Form Design tab to add two text boxes to the form (see Figure 5.1).

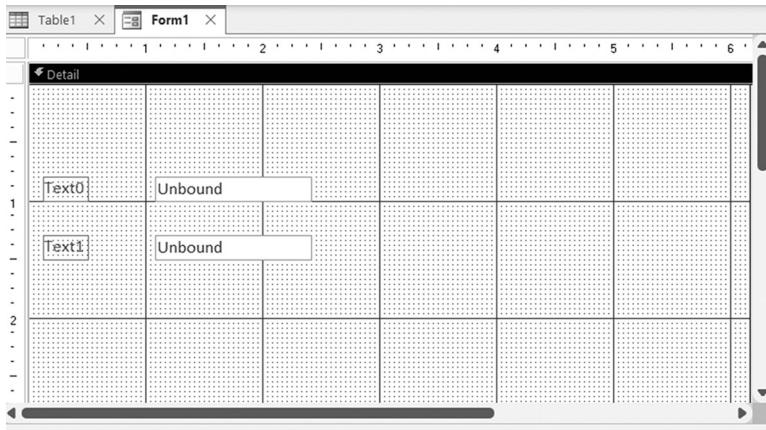


FIGURE 5.1 Placing text box controls on an Access form for Hands-On 5.5.

3. Select the **Text0** label and click the **Property Sheet** button in the **Tools** section of the Form Design tab.
4. In the property sheet, change the **Caption** property for the label to **User**.

5. In the property sheet, choose the second label from the dropdown and set its **Caption** property to **Password**.
6. Click the Unbound text box to the right of the User label. In the property sheet on the Other tab, set the **Name** property of this control to **txtUser**.
7. Click the Unbound text box to the right of the Password label. In the property sheet on the Other tab, set the **Name** property of this text box to **txtPwd** (see Figure 5.2).
8. In the property sheet on the Data tab, type **Password** next to the **Input Mask** property of the txtPwd text box control.
9. Click the **Button** (Form Control) in the **Controls** section of the Form Design tab and add a button to the form. When the Command Button Wizard dialog box appears, click **Cancel**. With the Command button selected, set the **Caption** and **Name** properties of this button by typing the following values in the property sheet next to the shown property name (see Figure 5.3):  
Name property: **cmdOK**  
Caption property: **OK**
10. Right-click the OK button and choose **Build Event** from the shortcut menu. In the Choose Builder dialog box, select **Code Builder** and click **OK**.
11. Enter the following code for the **cmdOK\_Click** event procedure. To make the procedure easier to understand, the conditional statements are shown with different formatting (bold and underlined).

```
Private Sub cmdOK_Click()
    If txtPwd = "FOX" Then
        MsgBox "You're not authorized to run this report."
    ElseIf txtPwd = "DOG" Then
        If txtUser = "John" Then
            MsgBox "You're logged on with restricted privileges."
        ElseIf txtUser = "Mark" Then
            MsgBox "Contact the Admin now."
        ElseIf txtUser = "Anne" Then
            MsgBox "Go home."
        Else
            MsgBox "Incorrect user name."
        End If
    Else
        MsgBox "Incorrect password or user name"
    End If
    Me.txtUser.SetFocus
End Sub
```

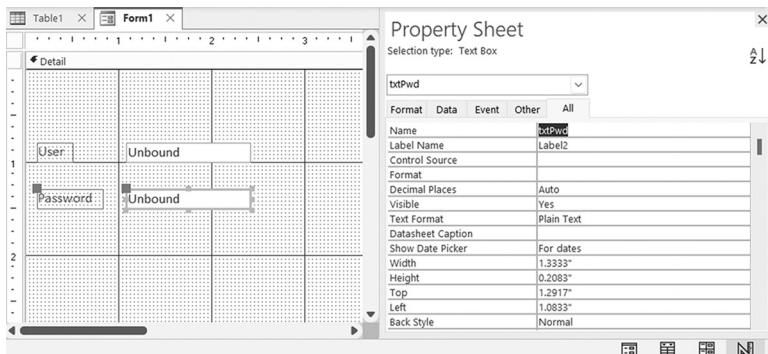


FIGURE 5.2 Setting the Name property of the text box control for Hands-On 5.5.

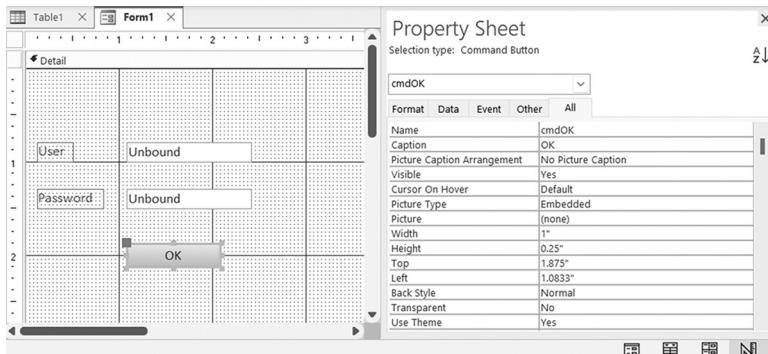


FIGURE 5.3 Setting the Command button properties for Hands-On 5.5.

12. Choose **File | Close and Return to Microsoft Access**. Save your form as **frmTestNesting**. When prompted to save standard modules you created in earlier exercises, save these objects with default names.
13. In the main Access window, switch to **Form** view. Enter any data in the User and Password text boxes, and then click **OK**.

The procedure first checks if the `txtPwd` text box on the form holds the text string “FOX.” If this is true, the message is displayed, and Visual Basic skips over the `ElseIf` and `Else` clauses until it finds the matching `End If` (see the bolded conditional statement).

If the `txtPwd` text box holds the string “DOG,” we use a nested `If...Then...` `Else` statement (underlined) to check if the content of the `txtUser` text box is set to John, Mark, or Anne, and then display the appropriate message. If the username is not one of the specified names, then the condition is false, and we jump to the underlined `Else` to display a message stating that the user entered an incorrect user name.

The first `If` block (in bold) is called the *outer If* statement. This outer statement contains one *inner If* statement (underlined).

#### SIDE BAR **Nesting Statements**

Nesting means placing one type of control structure inside another control structure. You will see more nesting examples with the looping structures discussed in Chapter 6, “Adding Repeating Actions to Your Access VBA Programs.”

Close the form.

---

## SELECT CASE STATEMENT

---

To avoid complex nested `If` statements that are difficult to follow, you can use the `Select Case` statement instead. The syntax of this statement is as follows:

```
Select Case testExpression
    Case expressionList1
        statements to be executed
        if expressionList1 matches testExpression
    Case expressionList2
        statements to be executed
        if expressionList2 matches testExpression
    Case expressionListN
        statements to be executed
        if expressionListN matches testExpression
    Case Else
        statements to be executed
        if no values match testExpression
End Select
```

You can place any number of cases to test between the keywords `Select Case` and `End Select`. The `Case Else` clause is optional. Use it when you expect that there may be conditional expressions that return `False`. In the `Select Case` statement, Visual Basic compares each `expressionList` with the value of `testExpression`.

Here’s the logic behind the `Select Case` statement. When Visual Basic encounters the `Select Case` clause, it makes note of the value of `testExpression`. Then it proceeds to test the expression following the first `Case` clause. If the value of this expression (`expressionList1`) matches the value stored

in `testExpression`, Visual Basic executes the statements until another `Case` clause is encountered, and then jumps to the `End Select` statement. If, however, the expression tested in the first `Case` clause does not match `testExpression`, Visual Basic checks the value of each `Case` clause until it finds a match. If none of the `Case` clauses contain the expression that matches the value stored in `testExpression`, Visual Basic jumps to the `Case Else` clause and executes the statements until it encounters the `End Select` keywords. Notice that the `Case Else` clause is optional. If your procedure does not use `Case Else`, and none of the `Case` clauses contain a value matching the value of `testExpression`, Visual Basic jumps to the statements following `End Select` and continues executing your procedure.

Let's look at an example of a procedure that uses the `Select Case` statement. As you already know, the `MsgBox` function allows you to display a message with one or more buttons. You also know that the result of the `MsgBox` function can be assigned to a variable. Using the `Select Case` statement, you can decide which action to take based on the button the user pressed in the message box.



### Hands-On 5.6 Using the Select Case Statement

1. Press **Alt+F11** to switch from the Access application window to the Visual Basic Editor window.
2. Insert a new module and enter the following **TestButtons** procedure in the module's Code window:

```
Sub TestButtons()
    Dim question As String
    Dim bts As Integer
    Dim myTitle As String
    Dim myButton As Integer

    question = "Do you want to preview the report now?"
    bts = vbYesNoCancel + vbQuestion + vbDefaultButton1
    myTitle = "Report"
    myButton = MsgBox(prompt:=question, buttons:=bts, _
                      Title:=myTitle)

    Select Case myButton
        Case 6
            DoCmd.OpenReport "Sales by Year", acPreview
        Case 7
            MsgBox "You can review the report later."
        Case Else
```

```
    MsgBox "You pressed Cancel."
End Select
End Sub
```

3. Run the TestButtons procedure three times, each time selecting a different button. (Because there is no Sales by Year report in the current database, an error message will pop up when you select Yes. Click **End** to exit the error message.)

The first part of the TestButtons procedure displays a message with three buttons: Yes, No, and Cancel. The value of the button selected by the user is assigned to the variable `myButton`.

If the user clicks Yes, the variable `myButton` is assigned the `vbYes` constant or its corresponding value 6. If the user selects No, the variable `myButton` is assigned the constant `vbNo` or its corresponding value 7. Lastly, if Cancel is pressed, the content of the variable `myButton` equals `vbCancel`, or 2.

The `Select Case` statement checks the values supplied after the `Case` clause against the value stored in the variable `myButton`. When there is a match, the appropriate `Case` statement is executed.

The TestButtons procedure will work the same if you use constants instead of button values:

```
Select Case myButton
Case vbYes
    DoCmd.OpenReport "Sales by Year", acPreview
Case vbNo
    MsgBox "You can review the report later."
Case Else
    MsgBox "You pressed Cancel."
End Select
```

You can omit the `Else` clause. Simply revise the `Select Case` statement as follows:

```
Select Case myButton
Case vbYes
    DoCmd.OpenReport "Sales by Year", acPreview
Case vbNo
    MsgBox "You can review the report later."
Case vbCancel
    MsgBox "You pressed Cancel."
End Select
```

---

**SIDE BAR** *Capture Errors with Case Else*

Although using `Case Else` in the `Select Case` statement isn't required, it's always a good idea to include one just in case the variable you are testing has an unexpected value. The `Case Else` clause is a good place to put an error message.

---

**Using `Is` with the Case Clause**

Sometimes a decision is made based on whether the test expression uses the greater than, less than, equal to, or some other relational operator (see Table 5.1). The `Is` keyword lets you use a conditional expression in a `Case` clause. The syntax for the `Select Case` clause using the `Is` keyword is as follows:

```
Select Case testExpression
    Case Is condition1
        statements if condition1 is true
    Case Is condition2
        statements if condition2 is true
    Case Is conditionN
        statements if conditionN is true
End Select
```

Let's look at an example:

```
Select Case myNumber
    Case Is <= 10
        MsgBox "The number is less than or equal to 10."
    Case 11
        MsgBox "You entered 11."
    Case Is >= 100
        MsgBox "The number is greater than or equal to 100."
    Case Else
        MsgBox "The number is between 12 and 99."
End Select
```

If the variable `myNumber` holds 120, the third `case` clause is true, and the only statement executed is the one between `Case Is >= 100` and the `Case Else` clause.

**Specifying a Range of Values in a Case Clause**

In the preceding example, you saw a simple `Select Case` statement that uses one expression in each `Case` clause. Many times, however, you may want to

specify a range of values in a `Case` clause. You do this by using the `To` keyword between the values of expressions, as in the following example:

```
Select Case unitsSold
    Case 1 To 100
        Discount = 0.05
    Case Is <= 500
        Discount = 0.1
    Case 501 To 1000
        Discount = 0.15
    Case Is >1000
        Discount = 0.2
End Select
```

Let's analyze this `Select Case` block with the assumption that the variable `unitsSold` currently has a value of 99. Visual Basic compares the value of the variable `unitsSold` with the conditional expression in the `Case` clauses. The first and third `Case` clauses illustrate how to use a range of values in a conditional expression by using the `To` keyword.

Because `unitsSold` equals 99, the condition in the first `Case` clause is true; thus, Visual Basic assigns the value 0.05 to the variable `Discount`. Well, how about the second `Case` clause, which is also true? Although it's obvious that 99 is less than or equal to 500, Visual Basic does not execute the associated statement `Discount = 0.1`. The reason for this is that once Visual Basic locates a `Case` clause with a true condition, it doesn't bother to look at the remaining `Case` clauses. It jumps over them and continues to execute the procedure with the instructions that may follow the `End Select` statement.

For more practice with the `Select Case` statement, let's use it in a function procedure. As you recall from Chapter 4, function procedures allow you to return a result to a subroutine. Suppose a subroutine must display a discount based on the number of units sold. You can get the number of units from the user and then run a function to figure out which discount applies.



### Hands-On 5.7 Using the Select Case Statement in a Function

1. Insert a new module and enter the following `DisplayDiscount` procedure in the Code window:

```
Sub DisplayDiscount()
    Dim unitsSold As Integer
    Dim myDiscount As Single
```

```

unitsSold = InputBox("Units Sold:")
myDiscount = GetDiscount(unitsSold)
MsgBox myDiscount
End Sub

```

2. In the same module, enter the following **GetDiscount** function procedure:

```

Function GetDiscount(unitsSold As Integer)
    Select Case unitsSold
        Case 1 To 200
            GetDiscount = 0.05
        Case 201 To 500
            GetDiscount = 0.1
        Case 501 To 1000
            GetDiscount = 0.15
        Case Is > 1000
            GetDiscount = 0.2
    End Select
End Function

```

3. Place the insertion point anywhere within the code of the **DisplayDiscount** procedure and press **F5** to run it.

The **DisplayDiscount** procedure passes the value stored in the variable `unitsSold` to the **GetDiscount** function. When Visual Basic encounters the `Select Case` statement, it checks whether the value of the first `Case` clause expression matches the value stored in the `unitsSold` parameter. If there is a match, Visual Basic assigns a 5% discount (0.05) to the function name, and then jumps to the `End Select` keywords. Because there are no more statements to execute inside the function procedure, Visual Basic returns to the calling procedure, **DisplayDiscount**. Here it assigns the function's result to the variable `myDiscount`. The last statement displays the value of the retrieved discount in a message box.

4. Choose **File | Save Chap05** and click OK when prompted to save the changes to the modules you created during the hands-on exercises.
5. Choose **File | Close and Return to Microsoft Access**.
6. Close the **Chap05.accdb** database and exit Access.

### **Specifying Multiple Expressions in a Case Clause**

---

You may specify multiple conditions within a single `Case` clause by separating each condition with a comma:

```

Select Case myMonth
    Case "January", "February", "March"
        Debug.Print myMonth & ": 1st Qtr."

```

```
Case "April", "May", "June"  
    Debug.Print myMonth & ": 2nd Qtr."  
Case "July", "August", "September"  
    Debug.Print myMonth & ": 3rd Qtr."  
Case "October", "November", "December"  
    Debug.Print myMonth & ": 4th Qtr."  
End Select
```

**NOTE***Multiple Conditions within a Case Clause*

*The commas used to separate conditions within a Case clause have the same meaning as the OR operator used in the If statement. The Case clause is true if at least one of the conditions is true.*

## SUMMARY

---

The conditional statements introduced in this chapter allow you to control the flow of your VBA procedure. By testing the truth of a condition, you can decide which statements should be run and which should be skipped over. In other words, instead of running your procedure from top to bottom, line by line, you can execute only certain lines. Here are a few guidelines to help you determine which conditional statement you should use:

- If you want to supply only one condition, the simple If...Then statement is the best choice.
- If you need to decide which of two actions to perform, use the If...Then...Else statement.
- If your procedure requires two or more conditions, use the If...Then...ElseIf or Select Case statements.
- If your procedure has many conditions, use the Select Case statement. This statement is more flexible and easier to comprehend than the If...Then...ElseIf statement.

Sometimes decisions must be repeated. The next chapter teaches you how your procedures can perform the same actions repeatedly.



# Chapter 6

# ADDING REPEATING ACTIONS TO YOUR ACCESS VBA PROGRAMS

**N**ow that you've learned how conditional statements can give your VBA procedures decision-making capabilities, it's time to get more involved. Not all decisions are easy. Sometimes you will need to perform a number of statements several times to arrive at a certain condition. On other occasions, however, after you've reached the decision, you may need to run the specified statements as long as a condition is true or until a condition becomes true. In programming, performing repetitive tasks is called *looping*. VBA has various looping structures that allow you to repeat a sequence of statements several times. In this chapter, you learn how to loop through your code.

---

#### SIDE BAR *What Is a Loop?*

---

A *loop* is a programming structure that causes a section of program code to execute repeatedly. VBA provides several structures to implement loops in your procedures: `Do...While`, `Do...Until`, `For...Next`, and `For Each...Next`.

---

## USING THE DO...WHILE STATEMENT

---

Visual Basic has two types of `Do` loop statements that repeat a sequence of statements either as long as or until a certain condition is true: `Do...While` and `Do...Until`.

The `Do...While` statement lets you repeat an action as long as a condition is true. This statement has the following syntax:

```
Do While condition  
    statement1  
    statement2  
    statementN  
Loop
```

When Visual Basic encounters this loop, it first checks the truth value of the condition. If the condition is false, the statements inside the loop are not executed, and Visual Basic will continue to execute the program with the first statement after the `Loop` keyword or will exit the program if there are no more statements to execute. If the condition is true, the statements inside the loop are run one by one until the `Loop` statement is encountered. The `Loop` statement tells Visual Basic to repeat the entire process again as long as the testing of the condition in the `Do...While` statement is true.

Let's see how you can put the `Do...While` loop to good use in Access. You will find out how to continuously display an input box until the user enters the correct password. The following hands-on exercise demonstrates this.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 6.1 Using the Do...While Statement

1. Start Access and create a new database named **Chap06.accdb** in your **C:\VBAAccess2021\_ByExample\_Primer** folder.
2. Once your new database is opened, press **Alt+F11** to switch to the Visual Basic Editor window.
3. Choose **Insert | Module** to add a new standard module.
4. In the **Module1** Code window, enter the following **AskForPassword** procedure:

```
Sub AskForPassword()  
    Dim pWord As String
```

```
pWord = ""  
Do While pWord <> "DADA"  
    pWord = InputBox("What is the report password?")  
Loop  
MsgBox "You entered the correct report password."  
End Sub
```

## 5. Run the AskForPassword procedure.

In this procedure, the statement inside the `Do...While` loop is executed as long as the variable `pWord` is not equal to the string “DADA.” If the user enters the correct password (“DADA”), Visual Basic leaves the loop and executes the `MsgBox` statement after the `Loop` keyword.

To allow the user to exit the procedure gracefully and cancel out of the input box if he does not know the correct password, add the following statement on an empty line before the `Loop` keyword:

```
If pWord = "" Then Exit Do
```

The `Exit Do` statement tells Visual Basic to exit the `Do` loop if the variable `pWord` does not hold any value (please see the section titled “Exiting Loops Early” later in this chapter). Therefore, when the input box appears, the user can leave the text field empty and click OK or Cancel to stop the procedure. Without the `Exit Do` statement, the procedure will keep on asking the user to enter the password until the correct value is supplied.

To forgo displaying the informational message when the user has not provided the correct password, you may want to use the conditional statement `If...Then` that you learned in the previous chapter. Here is the revised `AskForPassword` procedure:

```
Sub AskForPassword_Revised() ' revised procedure  
    Dim pWord As String  
  
    pWord = ""  
    Do While pWord <> "DADA"  
        pWord = InputBox("What is the report password?")  
        If pWord = "" Then  
            MsgBox "You did not enter a password."  
            Exit Do  
        End If  
    Loop  
    If pWord <> "" Then  
        MsgBox "You entered the correct report password."  
    End If  
End Sub
```

### **Another Approach to the Do...While Statement**

---

The `Do...While` statement has another syntax that lets you test the condition at the bottom of the loop:

```
Do
    statement1
    statement2
    statementN
Loop While condition
```

When you test the condition at the bottom of the loop, the statements inside the loop are executed at least once. Let's try this in the next hands-on exercise.



### **Hands-On 6.2 Using the Do...While Statement with a Condition at the Bottom of the Loop**

1. In the Visual Basic Editor window, insert a new module and enter the following `SignIn` procedure:

```
Sub SignIn()
    Dim secretCode As String

    Do
        secretCode = InputBox("Enter your secret code:")
        If secretCode = "sp1045" Then Exit Do
    Loop While secretCode <> "sp1045"
End Sub
```

2. Run the `SignIn` procedure.

Notice that by the time the condition is evaluated, Visual Basic has already executed the statements one time. In addition to placing the condition at the end of the loop, the `SignIn` procedure shows again how to exit the loop when a condition is reached. When the `Exit Do` statement is encountered, the loop ends immediately.

To exit the loop in the `SignIn` procedure without entering the password, you may revise it as follows:

```
Sub SignIn_Revised() 'revised procedure
    Dim secretCode As String

    Do
        secretCode = InputBox("Enter your secret code:")
        If secretCode = "sp1045" Or secretCode = "" Then
            Exit Do
    Loop
```

```
    End If
Loop While secretCode <> "sp1045"
End Sub
```

### SIDE BAR **Avoid Infinite Loops**

If you don't design your loop correctly, you can get an *infinite loop*—a loop that never ends. You will not be able to stop the procedure by using the Esc key. The following procedure causes the loop to execute endlessly because the programmer forgot to include the test condition:

```
Sub SayHello()
Do
    MsgBox "Hello."
Loop
End Sub
```

To stop the execution of the infinite loop, you must press Ctrl+Break. When Visual Basic displays the message box “Code execution has been interrupted,” click End to end the procedure.

## USING THE DO...UNTIL STATEMENT

Another handy loop is `Do...Until`, which allows you to repeat one or more statements until a condition becomes true. In other words, `Do...Until` repeats a block of code as long as something is false. Here is the syntax:

```
Do Until condition
    statement1
    statement2
    statementN
Loop
```

Using the preceding syntax, you can now rewrite the `AskForPassword` procedure (written in Hands-On 6.1) as shown in the following hands-on exercise.



### **Hands-On 6.3 Using the Do...Until Statement**

1. In the Visual Basic Editor window, insert a new module and type the `AskForPassword2` procedure:

```
Sub AskForPassword2()
    Dim pWord As String
```

```
pWord = ""  
Do Until pWord = "DADA"  
    pWord = InputBox("What is the report password?")  
Loop  
End Sub
```

## 2. Run the AskForPassword2 procedure.

The first line of this procedure says: Perform the following statements until the variable `pWord` holds the value “DADA.” As a result, until the correct password is supplied, Visual Basic executes the `InputBox` statement inside the loop. This process continues as long as the condition `pWord = "DADA"` evaluates to false. You could modify this procedure to allow the user to cancel the input box without supplying the password, as follows:

```
Sub AskForPassword_Revised() 'revised procedure  
    Dim pWord As String  
  
    pWord = ""  
    Do Until pWord = "DADA"  
        pWord = InputBox("What is the report password?")  
        If pWord = "" Then Exit Do  
    Loop  
End Sub
```

---

**SIDE BAR** *Variables and Loops*

All variables that appear in a loop should be assigned default values before the loop is entered.

---

## **Another Approach to the Do...Until Statement**

---

Similar to the `Do...While` statement, the `Do...Until` statement has a second syntax that lets you test the condition at the bottom of the loop:

```
Do  
    statement1  
    statement2  
    statementN  
Loop Until condition
```

If you want the statements to execute at least once, no matter what the value of the condition, place the condition on the line with the `Loop` statement. Let’s try out the following example that prints 27 numbers to the Immediate window.



### Hands-On 6.4 Using the Do...Until Statement with a Condition at the Bottom of the Loop

1. In the Visual Basic Editor window, insert a new module and type the **PrintNumbers** procedure shown here:

```
Sub PrintNumbers()
    Dim num As Integer

    num = 0
    Do
        num = num + 1
        Debug.Print num
    Loop Until num = 27
End Sub
```

2. Make sure the Immediate window is open in the Visual Basic Editor window (choose **View | Immediate Window** or press **Ctrl+G**).

3. Run the **PrintNumbers** procedure.

The variable `num` is initialized at the beginning of the procedure to zero (0). When Visual Basic enters the loop, the content of the variable `num` is increased by one, and the value is written to the Immediate window with the `Debug.Print` statement. Next, the condition tells Visual Basic that it should execute the statements inside the loop until the variable `num` equals 27.

4. Return to the main Access application window by choosing **File | Close and Return to Microsoft Access**. When prompted, save the changes to all the modules.

#### **SIDE BAR** *Counters*

A *counter* is a numeric variable that keeps track of the number of items that have been processed. The preceding **PrintNumbers** procedure declares the variable `num` to keep track of numbers that were printed. A counter variable should be initialized (assigned a value) at the beginning of the program. This ensures that you always know the exact value of the counter before you begin using it. A counter can be incremented or decremented by a specified value.

## USING THE FOR...NEXT STATEMENT

The `For...Next` statement is used when you know how many times you want to repeat a group of statements. The syntax of a `For...Next` statement looks like this:

```
For counter = start To end [Step increment]
    statement1
    statement2
    statementN
Next [counter]
```

The code in the brackets is optional. `Counter` is a numeric variable that stores the number of iterations. `Start` is the number at which you want to begin counting. `End` indicates how many times the loop should be executed. For example, if you want to repeat the statements inside the loop five times, use the following `For` statement:

```
For counter = 1 To 5
    statements
Next
```

When Visual Basic encounters the `Next` statement, it will go back to the beginning of the loop and execute the statements inside the loop again, as long as the counter hasn't reached the `end` value. As soon as the value of `counter` is greater than the number entered after the `To` keyword, Visual Basic exits the loop. Because the variable `counter` automatically changes after each execution of the loop, sooner or later the value stored in the counter exceeds the value specified in `end`.

By default, every time Visual Basic executes the statements inside the loop, the value of the variable `counter` is increased by one. You can change this default setting by using the `Step` clause. For example, to increase the variable `counter` by three, use the following statement:

```
For counter = 1 To 5 Step 3
    statements
Next counter
```

When Visual Basic encounters this statement, it executes the statements inside the loop twice. The first time the loop runs, the counter equals 1. The second time the loop runs, the counter equals 4 (1+3). The loop does not run a third time, because now the counter equals 7 (4+3), causing Visual Basic to exit the loop.

Note that the `Step` increment is optional. Optional statements are always shown in square brackets (see the syntax at the beginning of this section). The `Step` increment isn't specified unless it's a value other than 1. You can place a negative number after `Step` in order to decrement this value from the counter each time it encounters the `Next` statement. The name of the variable (`counter`) after the `Next` statement is also optional; however, it's good programming practice to make your `Next` statements explicit by including the `counter` variable's name.

How can you use the `For...Next` loop in Microsoft Access? Suppose you want to retrieve the names of the text boxes located on an active form. The procedure in the next hands-on exercise demonstrates how to determine whether a control is a text box and how to display its name if a text box is found.



### Hands-On 6.5 Using the For...Next Statement

1. Delete the **Table1** that Access created automatically when you created the **Chap6.accdb** database.
2. Make sure you have a copy of the **Northwind 2007.accdb** database from the companion files in your **VBAAccess2021\_ByExample\_Primer** folder.
3. Import the **Customers** table from the **Northwind 2007.accdb** database. To do this, click **New Data Source** | **From Database** | **Access** in the Import & Link section of the External Data tab. In the File name text box of the Get External Data dialog box, enter **C:\VBAAccess2021\_ByExample\_Primer\Northwind 2007.accdb** and click **OK**. In the Import Objects dialog box, select the **Customers** table and click **OK**. Click **Close** to exit the Get External Data dialog box.
4. Now, create a simple **Customers** form based on the Customers table. To do this, select the Customers table in the navigation pane by clicking on its name. Next, use the **Form Wizard** button in the Forms section of the Create tab. Add all fields from the **Customers** table and click **Finish**. Access creates a form as shown in Figure 6.1.

The screenshot shows the Microsoft Access 'Customers' form in Layout View. The form has a title bar 'Customers'. It contains a grid of text boxes for inputting customer information. The data entered is:

ID	Company	Notes
	Company A	
Customer ID	Bedecs	
First Name	Anna	
E-mail Address		
Job Title	Owner	
Business Phone	(123)555-0100	
Home Phone		
Mobile Phone		
Fax Number	(123)555-0101	
Address	123 1st Street	
City	Seattle	
State/Province	WA	
ZIP/Postal Code	99999	
Country/Region	USA	
Web Page		

At the bottom, there is a status bar showing 'Record: 1 of 29' and other navigation controls.

FIGURE 6.1 Automatic data entry form created by Microsoft Access shown in the Layout View.

5. Press **Alt+F11** to switch to the Visual Basic Editor window and insert a new module.
6. In the module's Code window, enter the following **GetTextBoxNames** procedure:

```
Sub GetTextBoxNames()
    Dim myForm As Form
    Dim myControl As Control
    Dim c As Integer

    Set myForm = Screen.ActiveForm
    Set myControl = Screen.ActiveControl

    For c = 0 To myForm.Count - 1
        If TypeOf myForm(c) Is TextBox Then
            Debug.Print myForm(c).Name
        End If
    Next c
End Sub
```

The conditional statement (**If...Then**) nested inside the **For...Next** loop tells Visual Basic to display the name of the active control only if it is a text box.

7. Run the **GetTextBoxNames** procedure.

---

**SIDE BAR** *Paired Statements*

**For** and **Next** must be paired. If one is missing, Visual Basic generates the following error message: "For without Next."

---

## **USING THE FOR EACH...NEXT STATEMENT**

---

When your procedure needs to loop through all the objects of a collection or all of the elements in an array (arrays are the subject of the next chapter), the **For Each...Next** statement should be used. This loop does not require a counter variable. Visual Basic can figure out on its own how many times the loop should execute. The **For Each...Next** statement looks like this:

```
For Each element In Group
    statement1
    statement2
    statementN
Next [element]
```

**Element** is a variable to which all the elements of an array or collection will be assigned. This variable must be of the Variant data type for an array and of the

Object data type for a collection. `Group` is the name of a collection or an array. Let's now see how to use the `For Each...Next` statement to print the names of the controls in the Customers form to the Immediate window.



### Hands-On 6.6 Using the For Each...Next Statement

This hands-on exercise requires the completion of Steps 1 and 2 of Hands-On 6.5.

1. Ensure that the Customers form you created in Hands-On 6.5 is still open in Form view.
2. Switch to the Visual Basic Editor window and insert a new module.
3. In the Code window, enter the `GetControls` procedure shown here:

```
Sub GetControls()
    Dim myControl As Control
    Dim myForm As Form

    DoCmd.OpenForm "Customers"
    Set myForm = Screen.ActiveForm

    For Each myControl In myForm
        Debug.Print myControl.Name
    Next
End Sub
```

4. Run the `GetControls` procedure.
5. The results of the procedure you just executed will be displayed in the Immediate window. If the window is not visible, press **Ctrl+G** in the Visual Basic Editor window to open the Immediate window or choose **View | Immediate Window**.

## EXITING LOOPS EARLY

---

Sometimes you might not want to wait until the loop ends on its own. It's possible that a user will enter the wrong data, a procedure will encounter an error, or perhaps the task will complete and there's no need to do additional looping. You can leave the loop early without reaching the condition that normally terminates it. Visual Basic has two types of `Exit` statements:

- The `Exit For` statement is used to end either a `For...Next` or a `For Each...Next` loop early.

- The `Exit Do` statement immediately exits any of the VBA `Do` loops.

The following hands-on exercise demonstrates how to use the `Exit For` statement to leave the `For Each...Next` loop early.



### Hands-On 6.7 Early Exit from a Loop

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, enter the following **GetControls2** procedure:

```
Sub GetControls2()
    Dim myControl As Control
    Dim myForm As Form

    DoCmd.OpenForm "Customers"
    Set myForm = Screen.ActiveForm

    For Each myControl In myForm
        Debug.Print myControl.Name
        If myControl.Name = "Address" Then
            Exit For
        End If
    Next
End Sub
```

3. Run the `GetControls2` procedure.

The `GetControls2` procedure examines the names of the controls in the open `Customers` form. If Visual Basic encounters the control named “Address,” it exits the loop.

4. Return to the main Access application window by choosing **File | Close and Return to Microsoft Access**.

#### SIDE BAR *Exiting Procedures*

If you want to exit a subroutine earlier than normal, use the `Exit Sub` statement. If the procedure is a function, use the `Exit Function` statement instead.

## NESTED LOOPS

So far in this chapter you have tried out various loops. Each procedure demonstrated the use of an individual looping structure. In programming practice, however, one loop is often placed inside another. Visual Basic allows you to “nest” various types of loops (`For` and `Do` loops) within the same procedure. When writing nested loops, you must make sure that each inner loop is com-

pletely contained inside the outer loop. Also, each loop must have a unique counter variable. When you use nesting loops, you can often execute specific tasks more effectively.

The GetFormsAndControls procedure shown in the following hands-on exercise illustrates how one `For Each...Next` loop is nested within another `For Each...Next` loop.



### Hands-On 6.8 Using Nested Loops

1. Import the **Employees** table from the **Northwind 2007.accdb** database located in your **VBAAccess2021\_ByExample\_Primer** folder (see Hands-On 6.5).
2. Use the **Form Wizard** to create a simple **Employees** form based on the **Employees** table.
3. Leave the **Employees** form in Form view and press **Alt+F11** to switch to the Visual Basic Editor window.
4. Choose **Insert | Module** to add a new module. In the module's Code window, enter the **GetFormsAndControls** procedure shown here:

```
Sub GetFormsAndControls()
    Dim accObj As AccessObject
    Dim myControl As Control

    For Each accObj In CurrentProject.AllForms
        Debug.Print accObj.Name & " Form"
        If Not accObj.IsLoaded Then
            DoCmd.OpenForm accObj.Name
        End If
        For Each myControl In Forms(accObj.Name).Controls
            Debug.Print Chr(9) & myControl.Name
        Next
        DoCmd.Close , , acSaveYes
    Next
End Sub
```

5. Run the **GetFormsAndControls** procedure.

The **GetFormsAndControls** procedure uses two `For Each...Next` loops to print the name of each currently open form and its controls to the Immediate window. To enumerate through the form's controls, the form must be open. Notice the use of the Access built-in function `IsLoaded`. The procedure will open the form only if it is not yet loaded. The control names are indented in the Immediate window using the `Chr(9)` function. This is like pressing the Tab key once. To get the same result, you can replace `Chr(9)` with a VBA constant: `vbTab`.

After reading the names of the controls, the form is closed, and the next form is processed in the same manner. The procedure ends when no more forms are found in the AllForms collection of CurrentProject.

6. Choose **File | Save Chap06** to save changes to the modules.
7. Choose **File | Close and Return to Microsoft Access**.
8. Close the **Chap06.accdb** database and click **Yes** when prompted to save changes.
9. Exit Access.

## SUMMARY

---

In this chapter, you learned how to repeat certain groups of statements in VBA procedures by using loops. While working with several types of loops, you saw how each loop performs repetitions in a slightly different way. As you gain experience, you'll find it easier to choose the appropriate flow control structure for your task.

The next chapter shows you how to write procedures that require a large number of variables.

# Chapter 7

## *KEEPING TRACK OF MULTIPLE VALUES USING ARRAYS*

In previous chapters, you worked with many VBA procedures that used variables to hold specific information about an object, property, or value. For each single value you wanted your procedure to manipulate, you declared a variable. But what if you have a series of values? If you had to write a VBA procedure to deal with larger amounts of data, you would have to create enough variables to handle all the data. Can you imagine the nightmare of storing currency exchange rates for all the countries in the world in your program? To create a table to hold the necessary data, you'd need at least three variables for each country: country name, currency name, and exchange rate. Fortunately, Visual Basic has a way to get around this problem. By clustering the related variables together, your VBA procedures can manage a large amount of data with ease. In this chapter, you'll learn how to manipulate lists and tables of data with arrays.

## UNDERSTANDING ARRAYS

---

In Visual Basic, an *array* is a special type of variable that represents a group of similar values that are of the same data type (String, Integer, Currency, Date, etc.). The two most common types of arrays are one-dimensional arrays (lists) and two-dimensional arrays (tables).

A one-dimensional array is sometimes referred to as a *list*. A shopping list, a list of the days of the week, and an employee list are examples of one-dimensional arrays or, simply, numbered lists. Each element in the list has an index value that allows you to access that element. For example, in the following illustration we have a one-dimensional array of six elements indexed from 0 to 5:

(0)	(1)	(2)	(3)	(4)	(5)
-----	-----	-----	-----	-----	-----

You can access the third element of this array by specifying index (2). By default, the first element of an array is indexed zero (0). You can change this behavior by using the `Option Base 1` statement or by explicitly coding the lower bound of your array as explained later in this chapter.

All elements of the array should be of the same data type. In other words, if you declare an array to hold textual data you cannot store in it both strings and integers. If you want to store values of *different* data types in the same array, you must declare the array as Variant as discussed later. Following are two examples of one-dimensional arrays: an array named `cities` that is populated with text (String data type—\$) and an array named `lotto` that contains six lottery numbers stored as integers (Integer data type—%).

A one-dimensional array: cities\$		A one-dimensional array: lotto%	
cities(0)	Baltimore	lotto(0)	25
cities(1)	Atlanta	lotto(1)	4
cities(2)	Boston	lotto(2)	31
cities(3)	Washington	lotto(3)	22
cities(4)	New York	lotto(4)	11
cities(5)	Trenton	lotto(5)	5

As you can see, the contents assigned to each array element match the array type. Storing values of different data types in the same array requires that you declare the array as Variant. You will learn how to declare arrays in the next section.

A two-dimensional array may be thought of as a table or matrix. The position of each element in a table is determined by its row and column numbers.

For example, an array that holds the yearly sales data for each product your company sells has two dimensions: the product name and the year. The following is a diagram of an empty two-dimensional array.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)

You can access the first element in the second row of this two-dimensional array by specifying indices (1, 0). Following are two examples of two-dimensional arrays: an array named `yearlyProductSales` that stores yearly product sales using the `Currency` data type (@) and an array named `exchange` (of `Variant` data type) that stores the name of the country, its currency, and the U.S. dollar exchange rate.

A Two-Dimensional array: `yearlyProductSales`@

Walking Cane (0,0)	\$25,023 (0,1)
Pill Crusher (1,0)	\$64,085 (1,1)
Electric Wheelchair (2,0)	\$345,016 (2,1)
Folding Walker (3,0)	\$85,244 (3,1)

A Two-dimensional array: `exchange` (not actual rates)

Japan (0,0)	Japanese Yen (0,1)	122.856 (0,2)
Australia (1,0)	Australian Dollar (1,1)	1,38220 (1,2)
Canada (2,0)	Canadian Dollar (2,1)	1.33512 (2,2)
Norway (3,0)	Norwegian Krone (3,1)	8.63744 (3,2)
Europe (4,0)	Euro (4,1)	0.939350 (4,2)

In these examples, the `yearlyProductSales` array can hold a maximum of 8 elements (4 rows \* 2 columns = 8) and the `exchange` array will allow a maximum of 15 elements (5 rows \* 3 columns = 15).

Although VBA arrays can have up to 60 dimensions, most people find it difficult to picture dimensions beyond 3D. A three-dimensional array is an array of two-dimensional arrays (tables) where each table has the same number of rows and columns. A three-dimensional array is identified by three indices: table, row, and column. The first element of a three-dimensional array is indexed (0, 0, 0).

## Declaring Arrays

---

Because an array is a variable, you must declare it in a similar way that you declare other variables (by using the keywords `Dim`, `Private`, or `Public`). For fixed-length arrays, the array bounds are listed in parentheses following the variable name. The *bounds* of an array are its lowest and highest indices. If a variable-length, or dynamic, array is being declared, the variable name is followed by an empty pair of parentheses.

The last part of the array declaration is the definition of the data type that the array will hold. An array can hold any of the following data types: Integer, Long, Single, Double, Variant, Currency, String, Boolean, Byte, or Date. Let's look at some examples:

Array Declaration (one-dimensional)	Description
<code>Dim cities(5) as String</code>	Declares a 6-element array, indexed 0 to 5
<code>Dim lotto(1 To 6) as String</code>	Declares a 6-element array, indexed 1 to 6
<code>Dim supplies(2 To 11)</code>	Declares a 10-element array, indexed 2 to 11
<code>Dim myIntegers(-3 To 6)</code>	Declares a 10-element array, indexed -3 to 6
<code>Dim dynArray() as Integer</code>	Declares a variable-length array whose bounds will be determined at runtime (see examples later in this chapter)

Array Declaration (two-dimensional)	Description
<code>Dim exchange(4, 2) as Variant</code>	Declares a two-dimensional array (five rows by three columns)
<code>Dim yearlyProductSales(3, 1) as Currency</code>	Declares a two-dimensional array (four rows by two columns)
<code>Dim my2Darray(1 To 3, 1 To 7) as Single</code>	Declares a two-dimensional array (three rows indexed 1 to 3 by seven columns indexed 1 to 7)

When you declare an array, Visual Basic automatically reserves enough memory space for it. The amount of memory allocated depends on the array's size and data type. For a one-dimensional array with six elements, Visual Basic sets aside 12 bytes—2 bytes for each element of the array (recall that the size of the Integer data type is 2 bytes—hence  $2 * 6 = 12$ ). The larger the array, the more memory space is required to store the data. Because arrays can eat up a lot of memory and impact your computer's performance, it's recommended that you declare arrays with only as many elements as you think you'll use.

---

**SIDE BAR** *What Is an Array Variable?*

An *array* is a group of variables that have a common name. While a typical variable can hold only one value, an *array variable* can store many individual values. You refer to a specific value in the array by using the array name and an index number.

---

**SIDE BAR** *Subscripted Variables*

The numbers inside the parentheses of the array variables are called *subscripts*, and each individual variable is called a subscripted variable or element. For example, `cities(5)` is the sixth subscripted variable (element) of the array `cities()`.

---

**Array Upper and Lower Bounds**

By default, VBA assigns zero (0) to the first element of the array. Therefore, number 1 represents the second element of the array, number 2 represents the third, and so on. With numeric indexing starting at 0, the one-dimensional array `cities(5)` contains six elements numbered from 0 to 5. If you'd rather start counting your array's elements at 1, you can explicitly specify a lower bound of the array by using an `Option Base 1` statement. This instruction must be placed in the declaration section at the top of a VBA module before any `Sub` statements. If you don't specify `Option Base 1` in a procedure that uses arrays, VBA assumes that the statement `Option Base 0` is to be used and begins indexing your array's elements at 0. If you'd rather not use the `Option Base 1` statement and still have the array indexing start at a number other than 0, you must specify the bounds of an array when declaring the array variable. As mentioned in the

previous section, the *bounds* of an array are its lowest and highest indices. Let's look at the following example:

```
Dim cities(3 To 6) As Integer
```

This statement declares a one-dimensional array with four elements. The numbers enclosed in parentheses after the array name specify the lower (3) and upper (6) bounds of the array. The index of the first element of this array is 3, the second 4, the third 5, and the fourth 6. Notice the keyword `To` between the lower and upper indices.

## **Initializing and Filling an Array**

---

After you declare an array, you must assign values to its elements. This is often referred to as "initializing an array," "filling an array," or "populating an array." The three methods you can use to load data into an array are discussed in this section.

### ***Filling an Array Using Individual Assignment Statements***

---

Assume you want to store the names of your six favorite cities in a one-dimensional array named `cities`. After declaring the array with the `Dim` statement:

```
Dim cities(5) as String
```

or

```
Dim cities$(5)
```

you can assign values to the array variable like this:

```
cities(0) = "Baltimore"
cities(1) = "Atlanta"
cities(2) = "Boston"
cities(3) = "San Diego"
cities(4) = "New York"
cities(5) = "Denver"
```

### ***Filling an Array Using the Array Function***

---

VBA has a built-in `Array` function that returns an array of `Variants`. Because `Variant` is the default data type, the `As Variant` clause is optional in the array variable declaration:

```
Dim cities() as Variant
```

or

```
Dim cities()
```

Notice that you don't specify the number of elements between the parentheses.

Using the `Array` function as shown here, you can easily assign values to your `cities` array:

```
cities = Array("Baltimore", "Atlanta", "Boston", _  
    "San Diego", "New York", "Denver")
```

When using the `Array` function to populate a six-element array like `cities`, the lower bound of the array is 0 or 1 and the upper bound is 5 or 6, depending on the setting of `Option Base` statement at the top of the module (see the previous section titled "Array Upper and Lower Bounds").

### ***Filling an Array Using the For...Next Loop***

---

The easiest way to learn how to use loops to populate an array is by writing a procedure that fills an array with a specific number of integer values. Let's look at the following example procedure:

```
Sub LoadArrayWithIntegers()  
    Dim myIntArray(1 To 10) As Integer  
    Dim i As Integer  
  
    ' Initialize random number generator  
    Randomize  
  
    ' Fill the array with 10 random numbers between 1 and 100  
    For i = 1 To 10  
        myIntArray(i) = Int((100 * Rnd) + 1)  
    Next  
  
    ' Print array values to the Immediate window  
    For i = 1 To 10  
        Debug.Print myIntArray(i)  
    Next  
End Sub
```

This procedure uses a `For...Next` loop to fill `myIntArray` with 10 random numbers between 1 and 100. The second loop is used to print out the values from the array. Notice that the procedure uses the `Rnd` function to generate a random number. This function returns a value less than 1 but greater than or equal to 0. You can try it out in the Immediate window by entering:

```
x=rnd  
?x
```

Before calling the `Rnd` function, the `LoadArrayWithIntegers` procedure uses the `Randomize` statement to initialize the random number generator. To become more familiar with the `Randomize` statement and `Rnd` function, be sure to follow up with the Access online help. For an additional example of using loops, `Randomize` and `Rnd`, see Hands-On 7.4.

## USING A ONE-DIMENSIONAL ARRAY

---

Having learned the basics of array variables, let's write a couple of VBA procedures to make arrays a part of your new skill set. The procedure in Hands-On 7.1 uses a one-dimensional array to programmatically display a list of six North American cities.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 7.1 Using a One-Dimensional Array

1. Start Access and create a new database named **Chap07.accdb** in your **C:\VBAAccess2021\_ByExample\_Primer** folder.
2. Once your new database is opened, press **Alt+F11** to switch to the Visual Basic Editor window.
3. Choose **Insert | Module** to add a new standard module.
4. In the Module1 Code window, enter the following **FavoriteCities** procedure. Be sure to enter the `Option Base 1` statement at the top of the module so that the array numbering will begin from 1.

```
Option Base 1
```

```
Sub FavoriteCities()
    ' declare the array
    Dim cities(6) As String

    ' assign the values to array elements
    cities(1) = "Baltimore"
    cities(2) = "Atlanta"
    cities(3) = "Boston"
    cities(4) = "San Diego"
    cities(5) = "New York"
    cities(6) = "Denver"
```

```
' display the list of cities
MsgBox cities(1) & Chr(13) & cities(2) & Chr(13) -
& cities(3) & Chr(13) & cities(4) & Chr(13) -
& cities(5) & Chr(13) & cities(6)
End Sub
```

**5.** Choose **Run | Run Sub/UserForm** to execute the FavoriteCities procedure.

Before the FavoriteCities procedure begins, the default indexing for an array is changed. Notice the `Option Base 1` statement at the top of the module window before the `Sub` statement. This statement tells Visual Basic to assign the number 1 instead of the default 0 to the first element of the array. The array `cities()` is declared with six elements of the String data type. Each element of the array is then assigned a value. The last statement in this procedure uses the `MsgBox` function to display the list of cities in a message box. When you run this procedure, each city name will appear on a separate line (see Figure 7.1). You can change the order of the displayed data by switching the index values.

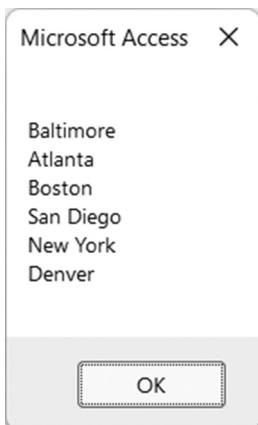


FIGURE 7.1 You can display the elements of a one-dimensional array with the `MsgBox` function.

- 6.** Click **OK** to close the message box.
- 7.** On your own, modify the FavoriteCities procedure so that it displays the names of the cities in reverse order (from 6 to 1).

**SIDE BAR** *The Range of the Array*

The spread of the elements specified by the `Dim` statement is called the *range* of the array—for example: `Dim mktgCodes(5 To 15)`.

## ARRAYS AND LOOPING STATEMENTS

---

Several of the looping statements you learned about in Chapter 6 (For...Next and For Each...Next) will come in handy now that you're ready to perform such tasks as populating an array and displaying the elements of an array. It's time to combine the skills you've learned so far.

How can you rewrite the FavoriteCities procedure, so each city name is shown in a separate message box? To answer this question, notice how in the FavoriteCities2 procedure in Hands-On 7.2 we are replacing the last statement of the original procedure with the For Each...Next loop.



### Hands-On 7.2 Using the For Each...Next Statement to List the Array Elements

1. In the Visual Basic Editor window, insert a new module.
2. Enter the **FavoriteCities2** procedure in the Code window. Be sure to enter the Option Base 1 statement at the top of the module.

```
Option Base 1
```

```
Sub FavoriteCities2()
    ' declare the array
    Dim cities(6) As String
    Dim city As Variant

    ' assign the values to array elements
    cities(1) = "Baltimore"
    cities(2) = "Atlanta"
    cities(3) = "Boston"
    cities(4) = "San Diego"
    cities(5) = "New York"
    cities(6) = "Denver"

    ' display the list of cities in separate messages
    For Each city In cities
        MsgBox city
    Next
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the FavoriteCities2 procedure. Notice that the For Each...Next loop uses the variable `city` of the Variant data type. As you recall from the previous chapter, the For Each...Next loop allows you to loop through all the objects in a collection or all the elements of an

array and perform the same action on each object or element. When you run the FavoriteCities2 procedure, the loop will execute as many times as there are elements in the array.

In Chapter 4, you practiced passing arguments as variables to subroutines and functions. The CityOperator procedure in Hands-On 7.3 demonstrates how you can pass elements of an array to another procedure.



### Hands-On 7.3 Passing Elements of an Array to Another Procedure

1. In the Visual Basic Editor window, insert a new module.
2. Enter the following two procedures (**CityOperator** and **Hello**) in the module's Code window. Be sure to enter the `Option Base 1` statement at the top of the module.

```
Option Base 1

Sub CityOperator()
    ' declare the array
    Dim cities(6) As String

    ' assign the values to array elements
    cities(1) = "Baltimore"
    cities(2) = "Atlanta"
    cities(3) = "Boston"
    cities(4) = "San Diego"
    cities(5) = "New York"
    cities(6) = "Denver"

    ' call another procedure and pass
    ' the array as argument
    Hello cities()
End Sub

Sub Hello(cities() As String)
    Dim counter As Integer

    For counter = 1 To 6
        MsgBox "Hello, " & cities(counter) & "!"
    Next
End Sub
```

Notice that the last statement in the CityOperator procedure calls the Hello procedure and passes to it the array `cities()` that holds the names of our favorite cities. Also notice that the declaration of the Hello procedure includes

an array type argument—`cities()`—passed to this procedure as String. In order to iterate through the elements of an array, you need to know how many elements are included in the passed array. You can easily retrieve this information via two array functions—`LBound` and `UBound`. These functions are discussed later in this chapter. In this procedure example, `LBound(cities())` will return 1 as the first element of the array, and `UBound(cities())` will return 6 as the last element of the `cities()` array. Therefore, the statement `For counter = LBound(cities()) To UBound(cities())` will boil down to `For counter = 1 To 6`.

### 3. Execute the CityOperator procedure (choose Run | Run Sub/UserForm).

Passing array elements from a subroutine to a subroutine or function procedure allows you to reuse the same array in many procedures without unnecessary duplication of the program code.

Here's how you can put to work your newly acquired knowledge about arrays and loops in real life. If you're an avid lotto player who is getting tired of picking your own lucky numbers, have Visual Basic do the picking. The **Lotto** procedure in Hands-On 7.4 populates an array with six numbers from 1 to 54. You can adjust this procedure to pick numbers from any range.



#### Hands-On 7.4 Using Arrays and Loops in Real Life

1. In the Visual Basic Editor window, insert a new module.
2. Enter the following **Lotto** procedure in the module's Code window:

```
Sub Lotto()
    Const spins = 6
    Const minNum = 1
    Const maxNum = 54
    Dim t As Integer ' looping variable in outer loop
    Dim i As Integer ' looping variable in inner loop
    Dim myNumbers As String ' string to hold all picks
    Dim lucky(spins) As String ' array to hold generated picks

    myNumbers = ""
    For t = 1 To spins
        Randomize
        lucky(t) = Int((maxNum - minNum + 1) * Rnd) + minNum

        ' check if this number was picked before
        For i = 1 To (t - 1)
            If lucky(t) = lucky(i) Then
                lucky(t) = Int((maxNum - minNum + 1) * Rnd) + minNum
                i = 0
            End If
        Next i
    Next t
End Sub
```

```
    End If
Next i
MsgBox "Lucky number is " & lucky(t), , "Lucky number " & t
myNumbers = myNumbers & " -" & lucky(t)
Next t
MsgBox "Lucky numbers are " & myNumbers, , "6 Lucky Numbers"
End Sub
```

The `Randomize` statement initializes the random number generator. The instruction `Int((maxNum - minNum + 1) * Rnd + minNum)` uses the `Rnd` function to generate a random value from the specified `minNum` to `maxNum`. The `Int` function converts the resulting random number to an integer. Instead of assigning constant values for `minNum` and `maxNum`, you can use the `InputBox` function to get these values from the user.

The inner `For...Next` loop ensures that each picked number is unique—it may not be any one of the previously picked numbers. If you omit the inner loop and run this procedure multiple times, you'll likely see some occurrences of duplicate numbers.

3. Execute the Lotto procedure (choose **Run | Run Sub/UserForm**) to get your very own computer-generated lottery numbers.

---

**SIDE BAR** *Initial Value of an Array Element*

Until a value is assigned to an element of an array, the element retains its default value. Numeric variables have a default value of zero (0), and string variables have a default value of empty string ("").

---

**SIDE BAR** *Passing Arrays between Procedures*

When an array is declared in a procedure, it is local to this procedure and unknown to other procedures. However, you can pass the local array to another procedure by using the array's name followed by an empty set of parentheses as an argument in the calling statement. For example, the statement `Hello cities()` calls the procedure named `Hello` and passes to it the array `cities`.

---

## USING A TWO-DIMENSIONAL ARRAY

---

Now that you know how to programmatically produce a list (a one-dimensional array), let's take a closer look at how you can work with tables of data. The following procedure creates a two-dimensional array that will hold country name, currency name, and exchange rate for three countries.



### Hands-On 7.5 Using a Two-Dimensional Array

1. In the Visual Basic Editor window, insert a new module.
2. Enter the **Exchange** procedure in the module's Code window:

```
Sub Exchange()
    Dim t As String
    Dim r As String
    Dim Ex(3, 3) As Variant

    t = Chr(9) & Chr(9) ' 2 Tabs
    r = Chr(13) ' Enter

    Ex(1, 1) = «Japan»
    Ex(1, 2) = «Yen»
    Ex(1, 3) = 122.856
    Ex(2, 1) = «Europe»
    Ex(2, 2) = «Euro»
    Ex(2, 3) = 0.939350
    Ex(3, 1) = «Canada»
    Ex(3, 2) = «Dollar»
    Ex(3, 3) = 1.33512

    MsgBox «Country « & t & «Currency» & t & _
        «1 USD» & r & r _
        & Ex(1, 1) & t & Ex(1, 2) & t & Ex(1, 3) & r _
        & Ex(2, 1) & t & Ex(2, 2) & t & Ex(2, 3) & r _
        & Ex(3, 1) & t & Ex(3, 2) & t & Ex(3, 3), , _
        "Exchange Rates"
End Sub
```

3. Execute the Exchange procedure (choose **Run | Run Sub/UserForm**).

When you run the Exchange procedure, you will see a message box with the information presented in three columns, as shown in Figure 7.2.

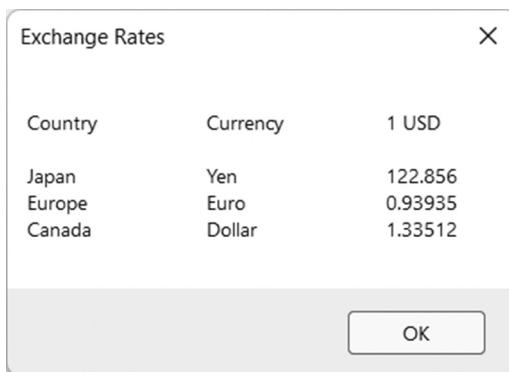


FIGURE 7.2 The text displayed in the message box can be custom formatted. (Note that these are fictitious exchange rates for demonstration only.)

4. Click OK to close the message box.

## STATIC AND DYNAMIC ARRAYS

---

The arrays introduced thus far are static. A *static array* is an array of a specific size. You use a static array when you know in advance how big the array should be. The size of the static array is specified in the array's declaration statement. For example, the statement `Dim Fruits(10) As String` declares a static array called `Fruits` that is made up of 10 elements.

But what if you're not sure how many elements your array will contain? If your procedure depends on user input, the number of user-supplied elements might vary every time the procedure is executed. How can you ensure that the array you declare is not wasting memory?

You may recall that after you declare an array, VBA sets aside enough memory to accommodate the array. If you declare an array to hold more elements than what you need, you'll end up wasting valuable computer resources. The solution to this problem is making your arrays dynamic. A *dynamic array* is an array whose size can change. You use a dynamic array when the array size will be determined each time the procedure is run.

---

**SIDE BAR** *Fixed-Dimension Arrays*

A static array contains a fixed number of elements. The number of elements in a static array will not change once it has been declared.

---

A dynamic array is declared by placing empty parentheses after the array name—for example:

```
Dim Fruits() As String
```

Before you use a dynamic array in your procedure, you must use the `ReDim` statement to dynamically set the lower and upper bounds of the array.

The `ReDim` statement redimensions arrays as the procedure code executes. The `ReDim` statement informs Visual Basic about the new size of the array. This statement can be used several times in the same procedure. Now let's write a procedure that demonstrates the use of a dynamic array.



### Hands-On 7.6 Using a Dynamic Array

1. Insert a new module and enter the following `DynArray` procedure in the module's Code window:

```
Sub DynArray()
    Dim counter As Integer
    Dim myArray() As Integer ' declare a dynamic array
    ReDim myArray(5) ' specify the initial size of the array
    Dim myValues As String

    ' populate myArray with values
    For counter = 1 To 5
        myArray(counter) = counter + 1
        myValues = myValues & myArray(counter) & Chr(13)
    Next

    ' change the size of myArray to hold 10 elements
    ReDim Preserve myArray(10)

    ' add new values to myArray
    For counter = 6 To 10
        myArray(counter) = counter * counter
        myValues = myValues & myArray(counter) & Chr(13)
    Next counter

    MsgBox myValues
    For counter = 1 To 10
        Debug.Print myArray(counter)
    Next counter
End Sub
```

In the DynArray procedure, the statement `Dim myArray() As Integer` declares a dynamic array called `myArray`. Although this statement declares the array, it does not allocate any memory to the array. The first `ReDim` statement specifies the initial size of `myArray` and reserves for it 10 bytes of memory to hold its five elements. As you know, every `Integer` value requires 2 bytes of memory. The `For...Next` loop populates `myArray` with data and writes the array's elements to the variable `myValues`. The value of the variable `counter` equals 1 at the beginning of the loop.

The first statement in the loop (`myArray(counter) = counter +1`) assigns the value 2 to the first element of `myArray`. The second statement (`myValues = myValues & myArray(counter) & Chr(13)`) enters the current value of `myArray`'s element followed by a carriage return (`Chr(13)`) into the variable `myValues`. The statements inside the loop are executed five times. Visual Basic places each new value in the variable `myValues` and proceeds to the next statement: `ReDim Preserve myArray(10)`.

Normally, when you change the size of the array, you lose all the values that were in that array. When used alone, the `ReDim` statement reinitializes the array. However, you can append new elements to an existing array by following the `ReDim` statement with the `Preserve` keyword. In other words, the `Preserve` keyword guarantees that the redimensioned array will not lose its existing data.

The second `For...Next` loop assigns values to the 6th through 10th elements of `myArray`. This time the values of the array's elements are obtained by multiplication: `counter * counter`.

2. Execute the `DynArray` procedure (choose **Run | Run Sub/UserForm**).

---

**SIDE BAR** *Dimensioning Arrays*

You can't assign a value to an array element until you have declared the array with the `Dim` or `ReDim` statement. (An exception to this is if you use the `Array` function discussed in the next section.)

---

## ARRAY FUNCTIONS

---

You can manipulate arrays with five built-in VBA functions: `Array`, `IsArray`, `Erase`, `LBound`, and `UBound`. The following sections demonstrate the use of each of these functions in VBA procedures.

## The Array Function

The `Array` function allows you to create an array during code execution without having to first dimension it. This function always returns an array of Variants. You can quickly place a series of values in a list by using the `Array` function.

The `CarInfo` procedure in the following hands-on exercise creates a fixed-size, one-dimensional, three-element array called `auto`.



### Hands-On 7.7 Using the Array Function

1. Insert a new module and enter the following `CarInfo` procedure in the module's Code window:

```
Option Base 1

Sub CarInfo()
    Dim auto As Variant

    auto = Array("Ford", "Black", "2021")
    MsgBox auto(2) & " " & auto(1) & ", " & auto(3)

    auto(2) = "4-door"
    MsgBox auto(2) & " " & auto(1) & ", " & auto(3)
End Sub
```

2. Run the `CarInfo` procedure and examine the results.

When you run this procedure, you get two message boxes. The first one displays the following text: “Black Ford, 2021.” After changing the value of the second array element, the second message box will say: “4-door Ford, 2021”

**NOTE**

*Be sure to enter Option Base 1 at the top of the module before running the CarInfo procedure. If this statement is missing in your module, Visual Basic will display runtime error 9—“Subscript out of range.”*

## The `IsArray` Function

The `IsArray` function lets you test whether a variable is an array. The `IsArray` function returns True if the variable is an array or False if it is not an array. Let's do another hands-on exercise.



## Hands-On 7.8 Using the IsArray Function

1. Insert a new module and enter the code of the **IsThisArray** procedure in the module's Code window:

```
Sub IsThisArray()
    ' declare a dynamic array
    Dim tblNames() As String
    Dim totalTables As Integer
    Dim counter As Integer
    Dim db As Database

    Set db = CurrentDb

    ' count the tables in the open database
    totalTables = db.TableDefs.Count

    ' specify the size of the array
    ReDim tblNames(1 To totalTables)

    ' enter and show the names of tables
    For counter = 1 To totalTables - 1
        tblNames(counter) = db.TableDefs(counter).Name
        Debug.Print tblNames(counter)
    Next counter

    ' check if this is indeed an array
    If IsArray(tblNames) Then
        MsgBox "The tblNames is an array."
    End If
End Sub
```

2. Run the **IsThisArray** procedure to examine its results.

When you run this procedure, the list of tables in the current database is written to the Immediate window. A message box displays whether the `tblNames` array is indeed an array.

## The Erase Function

When you want to remove all data from an array, you should use the `Erase` function. This function deletes all the data held by static or dynamic arrays. In addition, the `Erase` function reallocates all the memory assigned to a dynamic array. If a procedure must use the dynamic array again, you must use the `ReDim` statement to specify the size of the array. The next hands-on exercise demonstrates how to erase the data from the array `cities`.



### Hands-On 7.9 Removing Data from an Array

1. Insert a new module and enter the code of the **FunCities** procedure in the module's Code window:

```
' start indexing array elements at 1
Option Base 1

Sub FunCities()
    ' declare the array
    Dim cities(1 To 5) As String

    ' assign the values to array elements
    cities(1) = "Las Vegas"
    cities(2) = "Orlando"
    cities(3) = "Atlantic City"
    cities(4) = "New York"
    cities(5) = "San Francisco"

    ' display the list of cities
    MsgBox cities(1) & Chr(13) & cities(2) & Chr(13) _
        & cities(3) & Chr(13) & cities(4) & Chr(13) _
        & cities(5)

    Erase cities

    ' check if contents of array were erased
    MsgBox cities(1) & Chr(13) & cities(2) & Chr(13) _
        & cities(3) & Chr(13) & cities(4) & Chr(13) _
        & cities(5)
End Sub
```

2. Run the **FunCities** procedure to examine its results.

3. Click **OK** to close the message box.

Visual Basic should now display an empty message box because all values were deleted from the array by the **Erase** function.

4. Click **OK** to close the empty message box.

You may be wondering if there is a quicker and easier way to determine if the array is empty or contains data.

Instead of looping through the array elements, you can use the **Join** function to concatenate all elements of an array into a single string. Recall that you used

this function earlier in the book to concatenate first and last name. The syntax of the `Join` function is shown below:

```
Join (<source array>, [<Delimiter>])
```

The source array is the name of the array whose elements we need to join.

`Delimiter` is the character you'd like to use between the concatenated elements. This is an optional parameter and if it's not provided, a space will be used. If, instead, you provide a zero-length string (""), all elements will be joined without any extra characters in between. Let's see how to use the `Join` function with the `Cities` array in the `FunCities_withJoinFunction` procedure.

```
Sub FunCities_withJoinFunction()

    ' declare the array
    Dim cities(1 To 5) As String
    Dim arrayContent As String
    Dim itm As Variant

    ' assign the values to array elements
    cities(1) = "Las Vegas"
    cities(2) = "Orlando"
    cities(3) = "Atlantic City"
    cities(4) = "New York"
    cities(5) = "San Francisco"

    ' display the list of cities
    MsgBox cities(1) & Chr(13) & cities(2) & Chr(13) _
        & cities(3) & Chr(13) & cities(4) & Chr(13) _
        & cities(5)

    ' another way to display the list of cities

    For Each itm In cities
        arrayContent = arrayContent & itm & vbCrLf
    Next

    MsgBox arrayContent

    ' print list of cities contained in cities array

    arrayContent = Join(cities, ", ")
    Debug.Print arrayContent

    ' erase all content from the cities array
```

```

Erase cities
arrayContent = Join(cities, "")
Debug.Print arrayContent
' Check if cities array is empty
If Len(arrayContent) = 0 Then
    Debug.Print "Array is Empty."
Else
    Debug.Print "Array is not Empty."
    Debug.Print Len(arrayContent)
End If

End Sub

```

When you run the above procedure, you should see the cities listed in the Immediate window like this:

Las Vegas, Orlando, Atlantic City, New York, San Francisco

The above string is generated via the following code snippet:

```

' print list of cities contained in cities array
arrayContent = Join(cities, ", ")
Debug.Print arrayContent

```

In the above code, the string variable `arrayContent` gets its value from the `Join` function that concatenates the elements of `cities` array with the comma and a space (", ").

After printing the contents of the `arrayContent` variable to the Immediate window, we call the `Erase` function to erase all the cities we entered in that array. To find out if the array is indeed empty after we called the `Erase`, we again fill the string variable `arrayContent` with the value returned from the `Join` function, but instead of a comma and a space, we use an empty-string ("") as a delimiter:

```

arrayContent = Join(cities, "")"
Debug.Print arrayContent

```

The result of the above is an empty line in the Immediate window. To confirm that the array is indeed empty, we can use the `Len` function to determine the size of the array:

```

' Check if cities array is empty
If Len(arrayContent) = 0 Then
    Debug.Print "Array is Empty."
Else
    Debug.Print "Array is not Empty."

```

```
Debug.Print Len(arrayContent)
End If
```

## The LBound and UBound Functions

---

The `LBound` and `UBound` functions return whole numbers that indicate the lower bound and upper bound indices of an array.



### Hands-On 7.10 Finding the Lower and Upper Bounds of an Array

1. Insert a new module and enter the code of the `FunCities2` procedure in the module's Code window:

```
Sub FunCities2()
    ' declare the array
    Dim cities(1 To 5) As String

    ' assign the values to array elements
    cities(1) = "Las Vegas"
    cities(2) = "Orlando"
    cities(3) = "Atlantic City"
    cities(4) = "New York"
    cities(5) = "San Francisco"

    ' display the list of cities
    MsgBox cities(1) & Chr(13) & cities(2) & Chr(13) _
        & cities(3) & Chr(13) & cities(4) & Chr(13) _
        & cities(5)

    ' display the array bounds
    MsgBox "The lower bound: " & LBound(cities) & Chr(13) _
        & "The upper bound: " & UBound(cities)
End Sub
```

2. Run the `FunCities2` procedure.
3. Click **OK** to close the message box that displays the favorite cities.
4. Click **OK** to close the message box that displays the lower and upper bound indices.

To determine the upper and lower indices in a two-dimensional array, you may want to add the following statements at the end of the `Exchange` procedure

that was prepared in Hands-On 7.5 (add these lines just before the `End Sub` keywords):

```
MsgBox "The lower bound (first dimension) is " & LBound(Ex, 1) & "."
MsgBox "The upper bound (first dimension) is " & UBound(Ex, 1) & "."
MsgBox "The lower bound (second dimension) is " & LBound(Ex, 2) & "."
MsgBox "The upper bound (second dimension) is " & UBound(Ex, 2) & "."
```

**NOTE**

*When determining the lower and upper bound indices of a two-dimensional array, you must specify the dimension number: 1 for the first dimension and 2 for the second dimension.*

## ERRORS IN ARRAYS

When working with arrays, it's easy to make a mistake. If you try to assign more values than there are elements in the declared array, Visual Basic will display the error message "Subscript out of range" (see Figure 7.3).

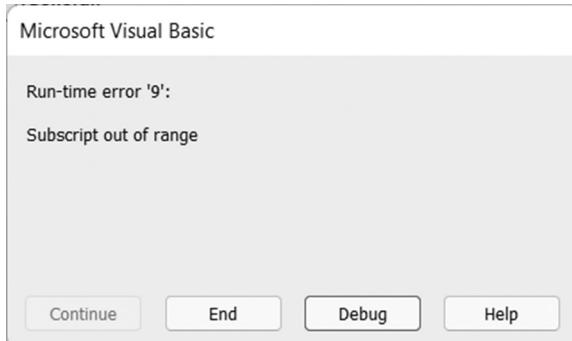


FIGURE 7.3 This error was caused by an attempt to access a nonexistent array element.

Suppose you declared a one-dimensional array that consists of three elements, and you are trying to assign a value to the fourth element. When you run the procedure, Visual Basic can't find the fourth element, so it displays the error message shown in Figure 7.3. If you click the Debug button, Visual Basic will highlight the line of code that caused the error (see Figure 7.4).

(General)	Zoo2
<pre> Option Compare Database Option Explicit  Sub Zoo1()     ' this procedure triggers an error     ' "Subscript out of range"     Dim zoo(3) As String     Dim i As Integer     Dim response As String      i = 0     Do         i = i + 1         response = InputBox("Enter a name of animal:")         zoo(i) = response     Loop <b>i=4</b>il response = "" End Sub </pre>	
<pre> Sub Zoo2()     ' this procedure avoids the error     ' "Subscript out of range"     Dim zoo(3) As String     Dim i As Integer     Dim response As String      i = 1     Do While i &gt;= LBound(zoo) And i &lt;= UBound(zoo)         response = InputBox("Enter a name of animal:")         If response = "" Then Exit Sub         zoo(i) = response         Debug.Print zoo(i)         i = i + 1     Loop End Sub </pre>	Zoo2

**FIGURE 7.4** The statement that triggered the error shown in Figure 7.3. is highlighted.

The error *Subscript out of range* is often triggered in procedures using loops. The procedure **Zoo1** shown in Hands-On 7.11 serves as an example of such a situation.



### Hands-On 7.11 Understanding Errors in Arrays

1. Insert a new module and enter the following **Zoo1** and **Zoo2** procedures in the module's Code window:

```

Sub Zoo1()
    ' this procedure triggers an error
    ' "Subscript out of range"
    Dim zoo(3) As String
    Dim i As Integer

```

```
Dim response As String

i = 0
Do
    i = i + 1
    response = InputBox("Enter a name of animal:")
    zoo(i) = response
Loop Until response = ""
End Sub

Sub Zoo2()
    ' this procedure avoids the error
    ' "Subscript out of range"
    Dim zoo(3) As String
    Dim i As Integer
    Dim response As String

    i = 1
    Do While i >= LBound(zoo) And i <= UBound(zoo)
        response = InputBox("Enter a name of animal:")
        If response = "" Then Exit Sub
        zoo(i) = response
        Debug.Print zoo(i)
        i = i + 1
    Loop
End Sub
```

2. Run the Zoo1 procedure and enter your favorite animal names when prompted. Do not cancel the procedure until you see the error.

While executing this procedure, when the variable *i* equals 4, Visual Basic will not be able to find the fourth element in a three-element array, so the error message will appear.

3. Click the **Debug** button in the error message.

Visual Basic will highlight the code that caused the error.

4. Position the cursor over the variable *i* in the highlighted line of code to view the variable's value.

Visual Basic displays: *i=4*

Notice that at the top of the Zoo1 procedure *zoo* has been declared as an array containing only three elements:

```
Dim zoo(3) As String
```

Because Visual Basic could not find the fourth element, it displayed the “Subscript out of range” error.

The Zoo2 procedure demonstrates how, by using the `LBound` and `UBound` functions introduced in the preceding section, you can avoid errors caused by an attempt to access a nonexistent array element.

5. Choose **Run | Reset** to terminate the debugging session and exit the procedure. You will learn more about debugging procedures in Chapter 9.

Another frequent error you may encounter while working with arrays is a *Type Mismatch* error. To avoid this error, keep in mind that each element of an array must be of the same data type. Therefore, if you attempt to assign to an element of an array a value that conflicts with the data type of the array, you will get a Type Mismatch error during the code execution. If you need to hold values of different data types in an array, declare the array as Variant.

## PARAMETER ARRAYS

---

In Chapter 4, you learned that values can be passed between subroutines or functions as either required or optional arguments. If the passed argument is not absolutely required for the procedure to execute, the argument's name is preceded by the keyword `Optional`. Sometimes, however, you don't know in advance how many arguments you want to pass. A classic example is addition. One time you may want to add 2 numbers together, another time you may want to add 3, 10, or 15 numbers.

Using the keyword `ParamArray`, you can pass an array consisting of any number of elements to your subroutines and functions. The following hands-on exercise uses the `AddMultipleArgs` function to add as many numbers as you may require. This function begins with the declaration of an array `myNumbers`. Notice the use of the `ParamArray` keyword.

The array must be declared as type Variant, and it must be the last argument in the procedure definition.



### Hands-On 7.12 Working with Parameter Arrays

1. Insert a new module and enter the following `AddMultipleArgs` function procedure in the module's Code window:

```
Function AddMultipleArgs(ParamArray myNumbers() As Variant)
    Dim mySum As Single
    Dim myValue As Variant

    For Each myValue In myNumbers
        mySum = mySum + myValue
```

```
    Next
    AddMultipleArgs = mySum
End Function
```

2. Choose **View | Immediate Window** and type the following instruction, and then press **Enter** to execute it:

```
?AddMultipleArgs(1, 23.24, 3, 24, 8, 34)
```

When you press Enter, Visual Basic returns the total of all the numbers in the parentheses: 93.24. You can supply an unlimited number of arguments. To add more values, enter additional values in the parentheses after the function name in the Immediate window, and then press Enter. Notice that each function argument must be separated by a comma.

## PASSING ARRAYS TO FUNCTION PROCEDURES

---

You can pass an array to a function procedure and return an array from a function. For example, let's assume you have a list of countries. You want to convert the country names stored in your array to uppercase and keep the original array intact. You can delegate the conversion process to a function procedure. When the array is passed using the `ByVal` keyword, the function will work with the copy of the original array. Any modifications performed within the function will affect only the copy. Therefore, the array in the calling procedure will not be modified.



### Hands-On 7.13 Passing an Array to a Function Procedure

1. Insert a new module and enter the following procedure and function in the module's Code window:

```
Sub ManipulateArray()
    Dim countries(1 To 6) As Variant
    Dim countriesUCase As Variant
    Dim i As Integer

    ' assign the values to array elements
    countries(1) = "Bulgaria"
    countries(2) = "Argentina"
    countries(3) = "Brazil"
    countries(4) = "Sweden"
    countries(5) = "New Zealand"
    countries(6) = "Denmark"
```

```
countriesUCase = ArrayToUCase(countries)

For i = LBound(countriesUCase) To UBound(countriesUCase)
    Debug.Print countriesUCase(i)
    Debug.Print countries(i) & " (Original Entry)"
Next i
End Sub

Public Function ArrayToUCase(ByVal myValues _
As Variant) As String()
    Dim i As Integer
    Dim Temp() As String
    If IsArray(myValues) Then
        ReDim Temp(LBound(myValues) To UBound(myValues))
        For i = LBound(myValues) To UBound(myValues)
            Temp(i) = CStr(UCase(myValues(i)))
        Next i
        ArrayToUCase = Temp
    End If
End Function
```

2. Run the ManipulateArray procedure and check its results in the Immediate window.

## **SORTING AN ARRAY**

---

We all find it easier to work with sorted data. Some operations on arrays, like finding maximum and minimum values, require that the array is sorted. Once it is sorted, you can find the maximum value by assigning the upper bound index to the sorted array, as in the following:

```
y = myIntArray(UBound(myIntArray))
```

The minimum value can be obtained by reading the first value of the sorted array:

```
x = myIntArray(1)
```

So how can you sort an array? Hands-On 7.14 demonstrates how to delegate the sorting task to a classic bubble sort routine. A *bubble sort* is a comparison sort. To create a sorted set, you step through the list to be sorted, compare each pair of adjacent items, and swap them if they are in the wrong order. As a result of this sorting algorithm, the smaller values “bubble” to the top of the list. In the next procedure, we will sort the list of countries alphabetically in ascending order.

## Hands-On 7.14 Sorting an Array

This hands-on exercise requires prior completion of Hands-On 7.13.

1. In the same module where you entered the `ArrayToUCase` function procedure, enter the following `BubbleSort` function procedure:

```
Sub BubbleSort(myArray As Variant)
    Dim i As Integer
    Dim j As Integer
    Dim uBnd As Integer
    Dim Temp As Variant
    uBnd = UBound(myArray)
    For i = LBound(myArray) To uBnd - 1
        For j = i + 1 To uBnd
            If UCASE(myArray(i)) > UCASE(myArray(j)) Then
                Temp = myArray(j)
                myArray(j) = myArray(i)
                myArray(i) = Temp
            End If
        Next j
    Next i
End Sub
```

2. Add the following statements to the `ManipulateArray` procedure, placing them just above the `For...Next` statement block (see Figure 7.5):

```
' call function to sort the array
BubbleSort countriesUCase
```

The screenshot shows the Microsoft Access VBA editor. On the left, the code for the `ManipulateArray` procedure is displayed. It includes code to assign values to an array, convert the array to uppercase using `ArrayToUCase`, and then sort it using `BubbleSort`. It also includes a `For...Next` loop to print the sorted array. On the right, the `Immediate` window is open, showing the original entries and the sorted entries. The original entries are: ARGENTINA, Bulgaria (Original Entry), BRAZIL, Argentina (Original Entry), SWEDEN, Brazil (Original Entry), DENMARK, Sweden (Original Entry), NEW ZEALAND, Norway (Original Entry), SWEDEN, Denmark (Original Entry). The sorted entries are: ARGENTINA, BRAZIL, SWEDEN, SWEDEN, SWEDEN, SWEDEN, SWEDEN, SWEDEN, SWEDEN, SWEDEN.

```
[General] [ManipulateArray]
' assign the values to array elements
countries(1) = "Bulgaria"
countries(2) = "Argentina"
countries(3) = "Brazil"
countries(4) = "Sweden"
countries(5) = "New Zealand"
countries(6) = "Denmark"

countriesUCase = ArrayToUCase(countries)
' call function to sort the array
BubbleSort countriesUCase

For i = LBound(countriesUCase) To UBound(countriesUCase)
    Debug.Print countriesUCase(i)
    Debug.Print countries(i) & " (Original Entry)"
    Next i
End Sub

Public Function ArrayToUCase(ByVal myValues _ 
    As Variant) As String()
    Dim i As Integer
    Dim Temp() As String

    If IsArray(myValues) Then
        ReDim Temp(LBound(myValues) To UBound(myValues))
        For i = LBound(myValues) To UBound(myValues)
            Temp(i) = CStr(UCASE(myValues(i)))
        Next i
        ArrayToUCase = Temp
    End If
End Function

Sub BubbleSort(myArray As Variant)
    Dim i As Integer
    Dim j As Integer
```

**FIGURE 7.5** Calling the `BubbleSort` function procedure from the `ManipulateArray` procedure.

3. Run the ManipulateArray procedure and check its results in the Immediate window. Notice that the countries that appear in uppercase letters are shown in alphabetic order.
4. Choose **File | Save Chap07** and save changes to the modules when prompted.
5. Choose **File | Close and Return to Microsoft Access**.
6. Close the **Chap07.accdb** database and exit Access.

## SUMMARY

---

In this chapter, you learned how, by creating an array, you can write procedures that require a large number of variables. You worked with examples of procedures that demonstrated how to declare and use a one-dimensional array (list) and a two-dimensional array (table). You learned the difference between static and dynamic arrays. This chapter introduced you to five built-in VBA functions that are frequently used with arrays (`Array`, `IsArray`, `Erase`, `LBound`, and `UBound`), as well as the `ParamArray` keyword. In addition, you learned how to use the `Join` and `Len` functions to determine whether the array contains any data. You also learned how to pass one array and return another array from a function procedure. Finally, you saw how to sort an array. You now know all the VBA control structures that can make your code more intelligent: conditional statements, loops, and arrays.

In the next chapter, you will learn how to use collections instead of arrays to manipulate large amounts of data.



# Chapter 8

## *KEEPING TRACK OF MULTIPLE VALUES USING COLLECTIONS*

In the previous chapter, you learned how arrays are used to quickly and easily manipulate a large number of items. Instead of creating multiple variables to keep track of your data, you only needed to declare one variable. Using arrays instead of defining individual variables saves you from writing many lines of repetitive code. As you have seen so far, in programming there are many ways of performing the same task. The method you use depends on your needs. So it is with storing multiple values. In addition to arrays, you can maintain your items of data while your program is running by using a special type of object – the collection. Like arrays, collections are used for grouping variables.

Because collections have built-in properties and methods that allow you to add, remove, and count their elements, they make working with multiple data items much easier than arrays. Collections can also be used to hold objects. This is a more advanced feature of object-oriented programming that we'll focus on in Chapter 24. For now, let's learn the basic skills of using collections for tracking and maintaining data in your VBA procedures.

## CREATING YOUR OWN COLLECTION

---

Collection is an object; it is a data type in VBA. To create a user-defined collection, begin by declaring an object variable of the Collection type. This variable is declared with the `New` keyword in the `Dim` statement:

```
Dim CollectionName As New Collection
```

Notice the `New` keyword. VBA uses this keyword to create a new instance of an object. The `CollectionName` is the name of your collection. You can use any name if it is not one of the reserved words Access uses for its own collections or other internal operations. You can define more than one collection, but each collection you define should have a distinct name so you can easily reference it in your code. Collections can be defined at the top of the standard module or within a procedure. They can also be defined in Class modules, as covered in detail in Chapter 24. Notice that unlike arrays, collections do not require you to predefine their size. Once you define the object variable of the Collection type, you are ready to begin adding items to it.

### **Adding Items to Your Collection**

---

You can insert new items into the collection by using the `Add` method. The items with which you populate your collection do not have to be of the same data type. The `Add` method looks as follows:

```
object.Add item[, key, before, after]
```

The `object` is the name you defined for your collection when you declared it. For example, if the name of your collection is `colFruits`, the following statement adds several new items to it:

```
colFruits.Add "Apple"  
colFruits.Add "Pear"  
colFruits.Add "Strawberry"  
colFruits.Add "Blueberry"  
colFruits.Add "Orange"  
colFruits.Add "Peach"
```

Although the other arguments are optional, they are quite useful. It's important to understand that the items in a user-defined collection are automatically assigned numbers starting with 1. However, they can also be assigned a unique key value. Instead of accessing a specific item with an index (1, 2, 3, and so on) at the time an object is added to a collection, you can assign a key for that object. For instance, to identify an individual in a collection of students or employees,

you could use their EmployeeID or StudentID as a key. For example, here's how you can add *James Allen* to `colPeople` collection, using his EmployeeID as a key:

```
Dim colPeople as New Collection  
colPeople.Add "James Allen", Key:="123456"
```

If you want to specify the position of the object in the collection, you can use either the `Before` or `After` argument (but not both). The `Before` argument is the object before which the new object is added. The `After` argument is the object after which the new object is added. For example, to add *Kiwi* to the `colFruits` collection so that it appears after the second item, use the following statement:

```
colFruits.Add «Kiwi», , , 2
```

or

```
colFruits.Add "Kiwi", After:=2
```

Notice that if you are not using the named argument `After`, you must place a comma for each of the preceding optional arguments that you are not specifying.

To enter *Cherry* in the first position, use the named `Before` argument:

```
colFruits.Add "Cherry", Before:=1
```

By using the optional `Before` or `After` arguments, you can easily add elements in any position.

Each element of a collection can be a different data type. Please note that arrays can support different data types only if they are defined as Variants. To store a date item in your collection, use the following statement:

```
colFruits.Add #12/10/2021#, Key:="InvoiceDate"
```

To store a number in your collection, the following statement can be used:

```
colFruits.Add 100.99, Key:="InvoiceTotal"
```

## Determine the Number of Items in Your Collection

---

Use the `Count` property to find the number of items in your collection. To determine the current number of fruit items in `colFruits`, write the following statement:

```
Debug.Print colFruits.Count
```

## Accessing Items in a Collection

To refer to a specific item in your collection, use its index or key value.

For example, to find out the names of the first collection item, use this statement:

```
Debug.Print colFruits.Item(1)
```

Because the `Item` method is a default method of the collection, you may omit it from the statement, as shown here:

```
Debug.Print colFruits(1)
```

If you know the item key, then you can quickly retrieve it like this:

```
Debug.Print colPeople("123456")
```

By using the Key to access a collection item, you can go to it directly without the need to iterate through all the items. Access VBA does not provide a built-in method to check if the Key exists, but you can write your own function to return a Boolean value of True if the key exists. Here's how you would do it:

```
Function KeyExists(colName As Collection, _  
    key As String) As Boolean  
    On Error GoTo EndHere  
    IsObject (colName.Item(key))  
    KeyExists = True  
  
EndHere:  
End Function
```

See Hands-On 8.1 on how to call the `KeyExists` function from your procedure code. The `IsObject` is a built-in VBA function that returns True if the passed to it expression (i.e., key name) represents an object variable. The statement `OnError GoTo EndHere` tells Access to jump to the line `EndHere:` if the result of the `IsObject` function is False. The error handling statements introduced here are covered in detail in Chapter 9.

When using keys in your collection, keep in mind another caveat: You cannot update the value stored in the Key unless the collection item was defined as an object. We focus on using objects in collections in Chapter 24.

## Removing Items from a Collection

---

Removing an item from your custom collection is as easy as adding an item. To remove an item, use the `Remove` method in the following format:

```
object.Remove index
```

object is the name of the custom collection that contains the object you want to remove. index is an expression specifying the position of the object in the collection. To remove the third fruit item from colFruits, you simply write the following statement:

```
colFruits.Remove 3
```

Collections are reindexed automatically when an item is removed. Therefore, to remove all items from a custom collection, you can use 1 for the Index argument, as in the following example:

```
Do While colFruits.Count > 0
  colFruits.Remove Index:=1
Loop
```

Another way to remove all objects from your collection is by using the For Each...Next loop. For example, to remove all objects from colFruits, use the following looping structure:

```
Dim m As Variant
For Each m in colFruits
  colFruits.Remove 1
Next
```

Note that the control variable (m) used in the For Each...Next loop must be declared as Variant or Object data type. Because collections are reindexed, the preceding statement will remove the first item of the collection on each iteration. After the loop completes, colFruits should have zero items. However, to be sure, use the Count property to find out.

```
Debug.Print colFruits.Count
```

You can also remove all items from a collection by setting the collection object variable to a new collection, like this:

```
Set colFruits = New Collection
```

## Updating Items in a Collection

---

When you add an item to your collection that has a basic data type such as string, integer, long, currency, or date, your collection will be read-only. This means you will not be able to change the value of the item. Access will display an error if you try to assign a value to an existing item in your collection:

```
' this statement will produce Run-time error 424 'Object required'
  colFruits(4) = "Cranberry"
```

Therefore, if your procedure requires that the values are updated, you should group your items into an array. If you want to stick with the collection, to change an item, remove the item and add a new one. The only time that the collection is updatable is when it references objects. Using objects with collections is covered in Chapter 24.

Let's proceed to the first Hands-On in this chapter, where you put all your knowledge about collections into a VBA procedure.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 8.1 Creating and Manipulating a Custom Collection

1. Start Access and create a new database named **Chap08.accdb** in your **C:\VBAAccess2021\_ByExample\_Primer** folder.
2. Once your new database is opened, press **Alt+F11** to switch to the Visual Basic Editor window.
3. Choose **Insert | Module** to add a new standard module.
4. In the Module1 Code window, enter the following **WorkWith\_Collection**, **Display\_Items**, and **KeyExists** procedures.

```
Sub WorkWith_Collection()
    Dim colFruits As New Collection
    Dim itm As Variant
    Dim strColItems As String

    colFruits.Add "Apple"
    colFruits.Add "Pear"
    colFruits.Add "Strawberry"
    colFruits.Add "Blueberry"
    colFruits.Add "Orange"
    colFruits.Add "Peach"
    colFruits.Add "Kiwi", , , 2
    colFruits.Add "Mango", , , 5
    colFruits.Add "Cherry", Before:=1
    colFruits.Add 100.99, Key:="InvoiceTotal"
    colFruits.Add #12/10/2021#, Key:="InvoiceDate"

    Debug.Print "Total Items in colFruits: " & colFruits.Count

    'call a procedure to display all items in the collection
    Display_Items colFruits, itm
End Sub
```

```
    colFruits.Remove 3
    Debug.Print "New Total Items in colFruits: " & colFruits.Count

    For Each itm In colFruits
        strColItems = strColItems & ", " & itm
    Next

    ' remove a comma and a space from the beginning of
    ' the strColItems variable
    strColItems = Mid(strColItems, 3, Len(strColItems))
    Debug.Print strColItems

    'Find if keys exist and if not display a message
    'and go to the next line

    If KeyExists(colFruits, "InvoiceDate") And _
       KeyExists(colFruits, "InvoiceTotal") Then
        Debug.Print "Invoiced on: " & colFruits("InvoiceDate") & _
                   vbCrLf & "Total: " & colFruits("InvoiceTotal")
    Else
        MsgBox "Provided key(s) not found."
    End If

    ' Remove all items from collection one by one
    For Each itm In colFruits
        colFruits.Remove 1
    Next
    Debug.Print "Total Items in colFruits: " & colFruits.Count

End Sub

Sub Display_Items(col As Collection, myItm As Variant)

    For Each myItm In col
        Debug.Print myItm
    Next
End Sub

Function KeyExists(colName As Collection, _
                   key As String) As Boolean

    On Error GoTo EndHere

    IsObject (colName.Item(key))
    KeyExists = True

EndFunction
```

```
EndHere:  
End Function
```

5. Choose **Run | Run Sub/UserForm** to execute the **WorkWith\_Collection** procedure.
6. The **WorkWith\_Collection** procedure performs various operations on the declared collection object variable `colFruits`. If you plan on using the same collection in other procedures in the same module, you will need to move its variable declaration statement to the top of the module.

### **Returning a Collection from a Function**

---

Collections, like arrays, can be used as parameters or return values to functions or subroutine procedures. In Hands-On 8.2, you will collect entries from the user via the VBA InputBox function and store them into an array. Next, you will pass that array to a function and return a collection with the same items. Let's see how this is done.



### **Hands-On 8.2 Creating and Manipulating a Custom Collection**

1. Choose **Insert | Module** to add a new standard module to the current VBA project.
2. In the Code window, enter the code as shown below.
3. Notice that the `allItems` variable is declared at the top of the module (above all the procedure code). This placement will make this variable available to all the procedures in this module.

```
Dim allItems As String  
  
Sub ShowCollItems()  
    Dim coll As Collection  
    Dim myArray As Variant  
    Dim itm As Variant  
  
    ' get items from user input  
    If AskForItems <> "" Then  
        Debug.Print allItems  
        ' extract items from the user input string (allItems)  
        ' and place them in an array  
        myArray = Split(allItems, ",")  
        Debug.Print "Array has " & UBound(myArray) + 1 & " items."  
  
        ' call function to create a collection from the array  
        Set coll = CreateCollection(myArray)
```

```

    ' iterate through the collection to display each item
    For Each itm In coll
        Debug.Print itm
    Next
    Debug.Print "Total items in the collection: " & coll.Count
End If
End Sub

Function AskForItems() As String
    allItems = InputBox("Enter your items separated by a comma", _
        "Demo - Get User Input", _
        "item1, item2")

    If allItems = "" Then
        AskForItems = ""
    Else
        AskForItems = allItems
    End If
End Function

Function CreateCollection(arrMyItems As Variant) As Collection
    Dim coll As New Collection
    Dim i As Integer
    For i = 0 To UBound(arrMyItems)
        coll.Add arrMyItems(i)
    Next i
    'Return a collection
    Set CreateCollection = coll
End Function

```

Before running the above code, let's examine what it does. The main procedure `ShowCollItems` declares three variables that we need for working with the array and collection. Notice that you don't need to use the `New` keyword to declare the collection because the collection is created inside the `CreateCollection` function. The declared object variable `coll` will be assigned a collection received from this function. In addition to the `CreateCollection` function, the `ShowCollItem` procedure relies on following functions: `AskForItems` and `Split`.

The custom function procedure `AskForItems` populates the `allItems` string variable (declared on the top of the module) with the values obtained from the user via the VBA `InputBox` function. The user is requested to input values as a comma-delimited string (each item must be separated by a comma). If the user does not enter any values and presses Cancel instead, the function

will return an empty string to the calling procedure (`ShowCollItems`). If items are entered, then the entire string will be returned. If the string returned from the `AskForItems` function is not empty, we continue running the statements within the If block. If the string is empty, the procedure ends.

Within the If block, we print the contents of the `allItems` variable to the Immediate window. Next, we use the built-in VBA `Split` function to extract the values from the `allItems` string using a comma delimiter and return an array (`myArray`). We use the `UBound` function to find the number of items in the array. Because by default arrays are zero-based, we need to add 1 to the count to get the correct number of items. Refer to the previous chapter on arrays if you'd like to add code here to list the items in the `myArray` variable.

The next set of statements focus on creating a collection. To do this, we set the `coll` variable to the result obtained from the `CreateCollection` function. We call the `CreateCollection` function and pass it the `myArray` variable. Notice that `myArray` is a Variant and the `CreateCollection` function was defined to expect the parameter of the Variant type. Inside the `CreateCollection` function, we start by declaring the `coll` variable of the Collection type. We also need a counter (`i`) to loop through the items of the array variable that we passed to this function. Note that the parameter name that the `CreateCollection` function expects can be any name you define.

Using the `For...Next` loop, we loop through the items of the array starting from zero and add each array item to the collection. Once we are done looping, we pass the entire collection to the calling procedure – `ShowCollItems`. `coll` is an object variable so we need to use the `Set` keyword to return it from the function:

```
'Return a collection
Set CreateCollection = coll
```

Now we are back again in the `ShowCollItems` procedure, this time returning a collection. To view the collection items, we use the `For Each...Next` loop to print each item to the Immediate window. Note that the `itm` iterator must be defined as Variant. The procedure ends by printing the total count of collection items.

## COLLECTIONS VS. ARRAYS

---

As you have seen so far, both collections and arrays provide a convenient way for storing and manipulating groups of similar items. Most people find collections easier to use and master than arrays. However, before you decide which

grouping structure you should use for storing items in your program, examine your needs. Arrays are usually faster and more convenient to use if you know ahead of time the number of items you are going to store. If the number of elements varies and you often need to add and remove elements, collections may be more efficient to use.

Let's do some feature comparison: collections versus arrays.

- Custom collections you create use 1 by default as a first element. Arrays by default are zero-based. You need to use the Option Base 1 statement to force the numbering of array items to start at position 1.
- You don't need to specify the size of your collection upfront as collections are dynamically allocated. Arrays, on the other hand, require that you define their size, and if you need to change the array size further in your procedure, you must use the `ReDim` keyword. Each time you redimension the array, Access takes up more resources.
- It's easy to add or remove items from the collection with the `Add` and `Remove` methods. With arrays, before you can add or remove items, you need to find the size of the array by using its upper and lower bounds.
- You can add new items to a collection in any position. To perform the same task using arrays, you must write more code.
- Collections can store items of different data types. Arrays can only store items of different data types when they are declared as a Variant.
- You can use the `For` and `For Each` loop to access items in a collection while with arrays you must first set and verify the upper and lower bounds to iterate through the items.
- Collections allow you to use Keys to access a particular item directly, while arrays don't provide this feature. If you want to access items using the Keys, use the Dictionary object instead of the Collection object (see Chapter 27 for details).

## **WATCHING THE EXECUTION OF YOUR VBA PROCEDURES**

To help you understand what's going on when your code runs and how one procedure passes information to a function and receives back the function's result, let's walk through the `ShowCollItems` procedure you created in Hands-On 8.2.

Treat this exercise as a brief introduction to the debugging techniques that are covered in detail in the next chapter.

### Hands-On 8.3 Code Walkthrough

1. In the Module 2 Code window, locate the `ShowCollItems` procedure.
2. Set a breakpoint by clicking in the left margin next to the following line of code, as shown in Figure 8.1:

```
If AskForItems <> "" Then
```

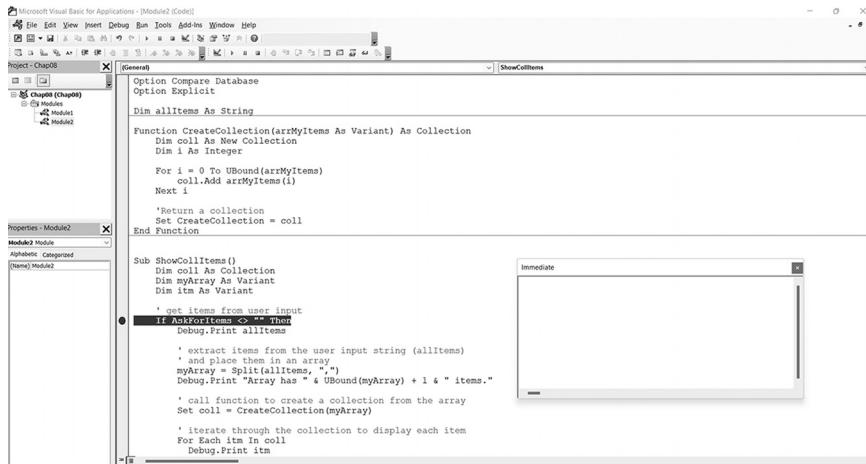


FIGURE 8.1 A red circle in the margin indicates a breakpoint. The statement with a breakpoint is displayed as white text on a red background.

3. Choose **View | Immediate window** and position the window next to the procedure as shown in Figure 8.1.
  4. Click anywhere within the code of `ShowCollItems` procedure and press **F5** or choose **Run | Run Sub/User Form**.
- Visual Basic should now jump to the line where you set a breakpoint (see Figure 8.2).

```

[General] ShowCollItems
Object
Option Compare Database
Option Explicit
Dim allItems As String

Function CreateCollection(arrMyItems As Variant) As Collection
    Dim coll As New Collection
    Dim i As Integer
    For i = 0 To UBound(arrMyItems)
        coll.Add arrMyItems(i)
    Next i
    'Return a collection
    Set CreateCollection = coll
End Function

Sub ShowCollItems()
    Dim coll As Collection
    Dim myArray As Variant
    Dim itm As Variant
    ' get items from user input
    If AskForItems <> "" Then
        Debug.Print allItems
        ' extract items from the user input string (allItems)
        ' and place them in an array
        myArray = Split(allItems, ",")
        Debug.Print "Array has " & UBound(myArray) + 1 & " items."
        ' call function to create a collection from the array
        Set coll = CreateCollection(myArray)
        ' iterate through the collection to display each item
        For Each itm In coll
            Debug.Print itm
        Next
    End If
End Sub

```

**FIGURE 8.2** When Visual Basic encounters a breakpoint while running a procedure, it switches to the Code window and displays a yellow arrow in the margin to the left of the statement at which the procedure is suspended.

5. Step through the code one statement at a time by pressing **F8**. Visual Basic runs the current statement, which in this case, is a call to the `AskForItems` function procedure. The yellow highlight moves to the line with the name of this function, as shown in Figure 8.3.
6. Press **F8** again and the highlight should move to the first line of the function procedure. Press **F8** again to execute this line.

You should now be presented with the Input box, where you need to enter the items you want to include in the array (see Figure 8.4). Notice that Access activates its main application window when you call the `InputBox` function.

7. Enter four names of your best friends or family members or any items you want separated by a comma and click **OK**. Clicking **Cancel** will terminate the procedure.

When you click **OK**, Access executes all the lines of the current function procedure and other code that was put in the `ShowCollItems` procedure and the `CreateCollection` function. You should see the Immediate window filled with the data you entered (see Figure 8.5). All `Debug` statements in your procedures wrote results to the Immediate window while the code was executing.

```

[General] [AskForItems]
Dim coll As Collection
Dim myArray As Variant
Dim itm As Variant

' get items from user input
If AskForItems <> "" Then
    Debug.Print allItems

    ' extract items from the user input string (allItems)
    ' and place them in an array
    myArray = Split(allItems, ",")
    Debug.Print "Array has " & UBound(myArray) + 1 & " items."

    ' call function to create a collection from the array
    Set coll = CreateCollection(myArray)

    ' iterate through the collection to display each item
    For Each itm In coll
        Debug.Print itm
    Next
    Debug.Print "Total items in the collection: " & coll.Count
End If

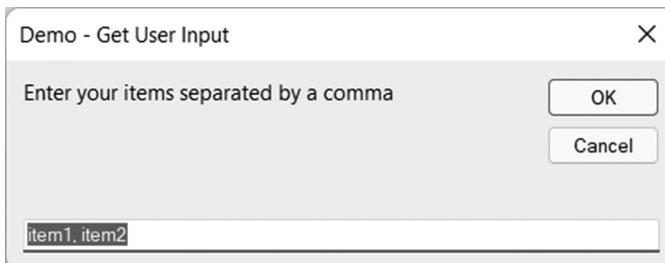
End Sub

Function AskForItems() As String
    allItems = InputBox("Enter your items separated by a comma", _
        "Demo - Get User Input", _
        "item1, item2")

    If allItems = "" Then
        AskForItems = ""
    Else
        AskForItems = allItems
    End If
End Function

```

**FIGURE 8.3** When you press the F8 key, you activate a step mode when each press of the key jumps to the next line of code, allowing you to step through the procedure.



**FIGURE 8.4** The Function procedure asks for items separated by a comma. These items will be used to fill an array variable.

```

Immediate
Mark, Peter, Anita, Yolanda
Array has 4 items.
Mark
Peter
Anita
Yolanda
Total items in the collection: 4

```

**FIGURE 8.5** The Immediate Window is populated with the data generated by the Debug Print statements.

This was a quick run through the VBA procedure. To go slower and gain more understanding of what's happening in the code, you need to put in more breakpoints next to the lines where you would like Visual Basic to temporarily stop the code execution.

8. Erase the data in the Immediate window by clicking anywhere within it, then press Ctrl+A, and then Delete.

Let's add more breakpoints and execute the procedure again.

9. Add the breakpoints as shown in Figure 8.6.

```

(General) CreateCollection
Option Compare Database
Option Explicit

Dim allItems As String

Function CreateCollection(arrMyItems As Variant) As Collection
    Dim coll As New Collection
    Dim i As Integer

    For i = 0 To UBound(arrMyItems)
        coll.Add arrMyItems(i)
    Next i

    'Return a collection
    Set CreateCollection = coll
End Function

Sub ShowCollItems()
    Dim coll As Collection
    Dim myArray As Variant
    Dim itm As Variant

    ' get items from user input
    If AskForItems < "" Then
        Debug.Print allItems
    End If

    ' extract items from the user input string (allItems)
    ' and place them in an array
    myArray = Split(allItems, ",")
    Debug.Print "Array has " & UBound(myArray) + 1 & " items."
    Set coll = CreateCollection(myArray)

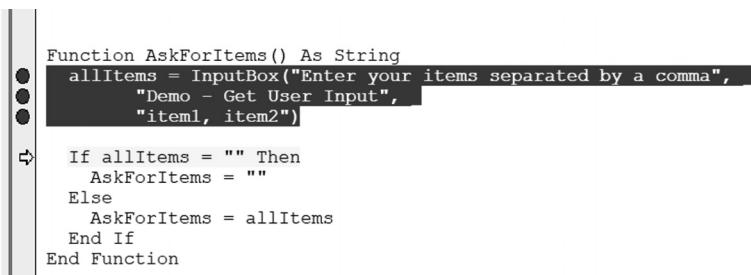
    ' call function to create a collection from the array
    ' iterate through the collection to display each item
    For Each item In coll
        Debug.Print item
    Next
    Debug.Print "Total items in the collection: " & coll.Count
    End If
End Sub

Function AskForItems() As String
    allItems = InputBox("Enter your items separated by a comma", _
        "Demo - Get User Input", _
        "item1, item2")
    If allItems = "" Then
        AskForItems = ""
    Else
        AskForItems = allItems
    End If
End Function

```

**FIGURE 8.6** Adding multiple breakpoints on important lines in VBA procedures allows you to better troubleshoot the code you've written or received from others.

10. Start from Step 4 above to execute the procedure again. After you've entered your items in the Input Box, you should be returned to the Code window with the yellow highlight positioned on the next statement in the AskForItems function procedure, as shown in Figure 8.7.



```
Function AskForItems() As String
    allItems = InputBox("Enter your items separated by a comma",
        "Demo - Get User Input",
        "item1, item2")
    If allItems = "" Then
        AskForItems = ""
    Else
        AskForItems = allItems
    End If
End Function
```

FIGURE 8.7 Because there was a breakpoint on the line that calls the `InputBox` function, when this line executes, the yellow highlight is moved to the next statement in the function, indicating that this is the next statement to be executed.

While your procedure execution is in the break mode with the yellow highlight on some code statement, you can use the Immediate window to find out the contents of your variables or issue other commands that will allow you to check the returned values. Let's find out what's in the `allItems` variable.

11. In the Immediate window type the following and press Enter:

```
?allItems
```

You should see a list of the items as you entered them in the Input Box. Here are my items:

```
Monday, Tuesday, Wednesday, Thursday
```

Now that you know that you have data in the `allItems` variable, it is easy to figure out what happens next.

12. Keep pressing **F8** to execute the function procedure step by step. The yellow highlight should eventually move to the line that tells Visual Basic to return the `allItems` variable from this function:

```
AskForItems = allItems
```

13. Press **F8** until the execution moves to the `ShowAllItems` procedure. Press **F8** and you should see the output of the `Debug.Print allItems` statement in the Immediate window.
14. Press **F8** again when the yellow highlight reaches the line that uses the `Split` function to extract items from the `allItems` string variable into the `myArray` variable:

```
myArray = Split(allItems, ",")
```

15. Press **F8** to fill the `myArray` variable.
16. Find the first element of the `myArray` variable by typing the following in the Immediate window and pressing Enter:

```
?myArray(0)
```

You should see the first item you entered.

17. Continue pressing **F8** key and examining each line of code until the execution moves to the `CreateCollection` function procedure.
18. Press **F8** again to move to the `For Next` loop within the `CreateCollection` procedure. Let's use the Immediate Window to find out what was passed to this function.
19. In the Immediate Window, type `?Ubound(arrMyItems)` and press **Enter**. You should see the count of the total items you entered. Recall that arrays are zero based by default so the actual count is `?Ubound(arrMyItems) + 1`.
20. Press **F8** until you are out of the loop that adds individual array items to the `coll` object variable.
21. When you reach the last statement of the `CreateCollection` procedure (`Set CreateCollection = coll`), type `?coll.count` to return the total number of items in your collection.
22. Press **F8** to pass the collection back to the calling procedure (`ShowAllItems`) and exit the function by pressing **F8** again.

Now you should be positioned on the next statement in the `ShowAllItems` procedure. This statement will loop through the collection as you continue pressing **F8**. All collection items and the total count will be written to the Immediate window.

23. Press **F8** until you reach the end of the procedure. The code execution will stop. You can repeat this entire walkthrough again for more practice before we clear the breakpoints in the next step.
24. Select **Debug | Clear All Breakpoints**.
25. Choose **File | Close and Return to Microsoft Access**.
26. Close the `Chap08.accdb` database and exit Access.

---

**SIDE BAR** *VBA Debugging Tools*

Visual Basic provides many debugging tools to help you analyze how your application operates, as well as to locate the source of errors in your procedures. See the next chapter for details on working with these tools.

---

## SUMMARY

This chapter walked you through the process of building your own custom collection object that can be used to track and manipulate data in your VBA program. You also learned how to build an array and convert it into a collection. After making some comparisons between collections and arrays, you learned how to analyze your VBA procedures by stepping through their programming code. As your procedures become more complex, you will need to start using special tools for tracing errors, which are covered in the next chapter.

# Chapter 9

# GETTING TO KNOW BUILT-IN TOOLS FOR TESTING AND DEBUGGING

You've probably heard that computer programs are "full of bugs." In programming, errors are called bugs, and *debugging* is a process of eliminating errors from your programs. Visual Basic for Applications provides a myriad of tools for tracking down and eliminating bugs. This chapter provides an overview of these tools that are within your reach when you work in the Visual Basic Editor screen.

## **SYNTAX, RUNTIME, AND LOGIC ERRORS**

While writing or editing VBA procedures, no matter how careful you are, you're likely to make some mistakes. For example, you may misspell a variable, misplace a comma or quotation mark, or forget a period or ending parenthesis. These kinds of mistakes are known as *syntax errors*. Fortunately, Visual Basic for Applications is quite helpful in spotting these kinds of errors. To have VBA

automatically check for correct syntax after you enter a line of code, choose Tools | Options in the VBE window. Make sure the Auto Syntax Check setting is selected on the Editor tab, as shown in Figure 9.1.

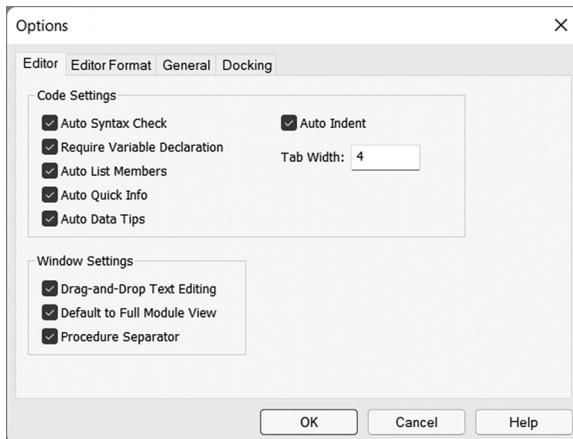


FIGURE 9.1 The Auto Syntax Check setting on the Editor tab of the Options dialog box helps you find typos in your VBA procedures.

When VBA finds a syntax error, it displays an error message box and changes the color of the incorrect line of code to red, or another color as indicated on the Editor Format tab in the Options dialog box.

If the explanation of the error in the error message isn't clear, you can click the Help button for more help. If Visual Basic for Applications cannot point you in the right direction, you must return to your procedure and carefully examine the offending instruction for missed letters, quotation marks, periods, colons, equal signs, and beginning and ending parentheses. Finding syntax errors can be aggravating and time-consuming. Certain syntax errors can be caught only during the execution of the procedure. While attempting to run your procedure, VBA can find errors that were caused by using invalid arguments or omitting instructions that are used in pairs, such as `If...End` statements and looping structures.

The first step in debugging a procedure is to correct all syntax errors. In addition to the syntax errors that we discussed in this chapter's introduction, there are two other types of errors: runtime and logic. *Runtime errors*, which occur while the procedure is running, are often caused by unexpected situations the programmer did not think of while writing the code. For example, the program may be trying to access a drive or a file that does not exist on the user's computer. Or it may be trying to copy a file to a CD-ROM disc or a flash card without first determining whether the user had inserted the required media.

The third type of error, a logic error, often does not generate a specific error message. Even though the procedure has no flaws in its syntax and runs without errors, it produces incorrect results. *Logic errors* happen when your procedure simply does not do what you want it to do. Logic errors are usually very difficult to locate. Those that happen intermittently are sometimes so well concealed that you can spend long hours—even days—trying to locate the source of the error.

## STOPPING A PROCEDURE

---

To understand what the VBA procedure is doing as it is being executed, you need to know how to put it in a state called “break mode.” VBA offers four methods of stopping your procedure and entering into the break mode:

- Pressing Ctrl+Break
- Setting one or more breakpoints
- Inserting the `Stop` statement
- Adding a watch expression

A break occurs when execution of your VBA procedure is suspended. Visual Basic remembers the values of all variables and the statement from which the execution of the procedure should resume when you decide to continue.

You can resume a suspended procedure in one of the following ways:

- Click the Run Sub/UserForm button on the toolbar
- Choose Run | Run Sub/UserForm from the menu bar
- Click the Continue button in the error message box (see Figure 9.2)

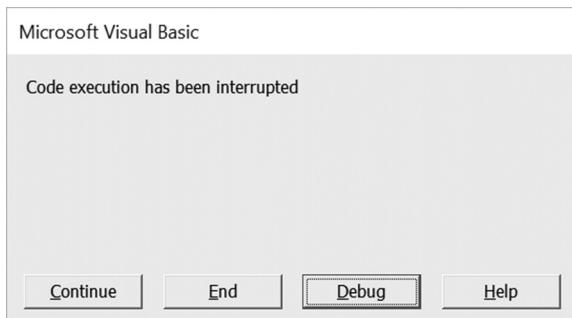


FIGURE 9.2 This message appears when you press Ctrl+Break while your VBA procedure is running.

The error message box shown in Figure 9.2 informs you that the procedure was halted. The description of each button is provided in Table 9.1.

**TABLE 9.1** Error message box buttons.

Button Name	Description
Continue	Click this button to resume code execution. This button will be grayed out if an error was encountered.
End	Click this button if you do not want to troubleshoot the procedure at this moment. VBA will stop code execution.
Debug	Click this button to enter break mode. The Code window will appear, and VBA will highlight the line at which the procedure execution was suspended. You can examine, debug, or step through the code.
Help	Click this button to view the online help that explains the cause of this error message.

## USING BREAKPOINTS

If you know where there may be a problem in your procedure code, you should suspend code execution at that location (on a given line). Set a breakpoint by pressing F9 when the cursor is on the desired line of code. When VBA gets to that line while running your procedure, it will display the Code window immediately. At this point you can step through the procedure code line by line by pressing F8 or choosing Debug | Step Into.

To see how this works, let's look at the following scenario. Assume that during the execution of the ListEndDates function procedure (see Custom Project 9.1) the following line of code could get you into trouble:

```
ListEndDates = Format(((Now() + intOffset) - 35) + 7 * row, _
"MM/DD/YYYY")
```

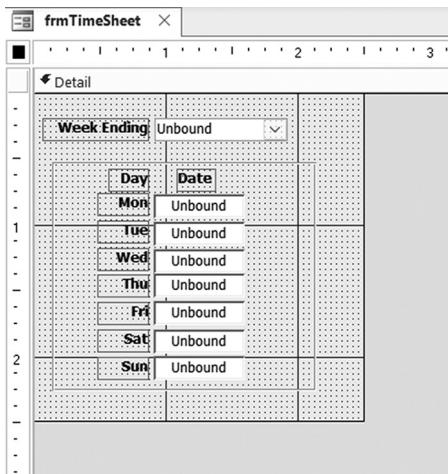
**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Custom Project 9.1 Debugging a Function Procedure

1. Start Access and create a new database named **Chap09.accdb** in your C:\VBAAccess2021\_ByExample\_Primer folder.
2. Create the form shown in Figure 9.3.



**FIGURE 9.3** The combo box control shown on this form will be filled with the result of the ListEndDates function.

3. Use the property sheet to set the following control properties:

Control Name	Property Name	Property Setting
combo box	Name Row Source Type Column Count	cboEndDate ListEndDates 1
text box controls	Name	txt1 txt2 txt3 txt4 txt5 txt6 txt7

4. Save the form as **frmTimeSheet**.
5. In the property sheet, select **Form** from the drop-down listbox. Click the **Event tab**. Choose **[Event Procedure]** from the drop-down list next to the **On Load** property, and then click the **Build** button (...). Complete the following **Form\_Load** procedure when the Code window appears:

```
Private Sub Form_Load()
    With Me.cboEndDate
        .SetFocus
        .ListIndex = 5 ' Select current end date
    End With
End Sub
```

6. Select the combo box control (cboEndDate) on the form. In the property sheet, click the **Event** tab. Choose **[Event Procedure]** from the drop-down list next to the **On Change** property, and then click the **Build** button (...). Enter the following code:

```
Private Sub cboEndDate_Change()
    Dim endDate As Date

    endDate = Me.cboEndDate.Value
    With Me
        .txt1 = Format(endDate - 6, "mm/dd")
        .txt2 = Format(endDate - 5, "mm/dd")
        .txt3 = Format(endDate - 4, "mm/dd")
        .txt4 = Format(endDate - 3, "mm/dd")
        .txt5 = Format(endDate - 2, "mm/dd")
        .txt6 = Format(endDate - 1, "mm/dd")
        .txt7 = Format(endDate - 0, "mm/dd")
    End With
End Sub
```

7. In the Visual Basic Editor window, choose **Insert | Module** to add a new standard module.
8. In the Properties window, change the Name property of Module1 to **TimeSheetProc**.
9. Enter the **ListEndDates** function procedure in the TimeSheetProc module:

```
Function ListEndDates(fld As Control, id As Variant, _
    row As Variant, col As Variant, _
    code As Variant) As Variant

    Dim intOffset As Integer

    Select Case code
        Case acLBInitialize
            ListEndDates = True
        Case acLBOpen
            ListEndDates = Timer
        Case acLBGetRowCount
            ListEndDates = 11
        Case acLBGetColumnCount
            ListEndDates = 1
        Case acLBGetColumnWidth
            ListEndDates = -1
        Case acLBGetValue
            ' days till end date
```

```

        intOffset = Abs((8 - Weekday(Now)) Mod 7)
        ' start 5 weeks prior to current week end date
        ' (7 days * 5 weeks = 35 days before next end date)
        ' and show 11 dates

        ListEndDates = Format(((Now() + intOffset) - 35) _
            + 7 * row, "MM/DD/YYYY")
    End Select
End Function

```

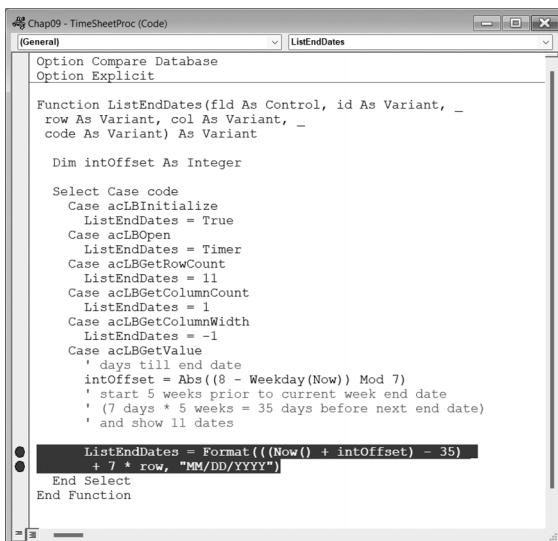
- 10.** In the ListEndDates function procedure, click anywhere on the line containing the following statement:

```
ListEndDates = Format(((Now() + intOffset) - 35) _
    + 7 * row, "MM/DD/YYYY")
```

- 11.** Press **F9** (or choose **Debug | Toggle Breakpoint**) to set a breakpoint on the line where the cursor is located.

When you set the breakpoint, Visual Basic displays a red dot in the margin. At the same time, the line that has the breakpoint will change to white text on a red background (see Figure 9.4). The color of the breakpoint can be changed on the Editor Format tab in the Options dialog box (choose **Tools | Options**).

Another way of setting a breakpoint is to click in the margin indicator to the left of the line on which you want to stop the procedure.



**FIGURE 9.4** The line of code where the breakpoint is set is displayed in the color specified on the Editor Format tab in the Options dialog box.

12. Press Alt+F11 to switch to the main Access application window and open the form **frmTimeSheet** in the Form view.

When the form is opened, Visual Basic for Applications will call the **ListEndDates** function to fill the combo box, executing all the statements until it encounters the breakpoint you set in steps 10–11. Once the breakpoint is reached, the code is suspended and the screen displays the Code window in break mode (notice the word “break” surrounded by square brackets in the Code window’s titlebar), as shown in Figure 9.5. VBA displays a yellow arrow in the margin to the left of the statement at which the procedure was suspended. At the same time, the statement appears inside a box with a yellow background. The arrow and the box indicate the current statement, or the statement that is about to be executed. If the current statement also contains a breakpoint, the margin displays both indicators overlapping one another (the circle and the arrow).

```

Option Compare Database
Option Explicit

Function ListEndDates(fld As Control, id As Variant, _
row As Variant, col As Variant, _
code As Variant) As Variant

Dim intOffset As Integer

Select Case code
    Case acLBInitialize
        ListEndDates = True
    Case acLBOpen
        ListEndDates = Timer
    Case acLBGetRowCount
        ListEndDates = 11
    Case acLBGetColumnCount
        ListEndDates = 1
    Case acLBGetColumnWidth
        ListEndDates = -1
    Case acLBGetValue
        ' days till end date
        intOffset = Abs((8 - Weekday(Now)) Mod 7)
        ' start 5 weeks prior to current week end date
        ' (7 days * 5 weeks = 35 days before next end date)
        ' and show 11 dates
        ListEndDates = Format(((Now() + intOffset) - 35) _
+ 7 * row, "MM/DD/YYYY")
    End Select
End Function

```

**FIGURE 9.5** Code window in break mode. A yellow arrow appears in the margin to the left of the statement at which the procedure was suspended. Because the current statement also contains a breakpoint (indicated by a red circle), the margin displays both indicators overlapping one another (the circle and the arrow).

13. Finish running the **ListEndDates** function procedure by pressing **F5** to continue without stopping or press **F8** to execute the procedure line by line.

When you step through your procedure code line by line by pressing F8, you can use the Immediate window to further test your procedure (see the section titled “Using the Immediate Window in Break Mode”). To learn more about

stepping through a procedure, refer to the section titled “Stepping through VBA Procedures” later in this chapter.

You can set any number of breakpoints in a procedure. This way you can suspend and continue the execution of your procedure as you please. Press F5 to quickly move between the breakpoints. You can analyze the code of your procedure and check the values of variables while code execution is suspended. You can also perform various tests by typing statements in the Immediate window. Consider setting a breakpoint if you suspect that your procedure never executes a certain block of code.

### Removing Breakpoints

---

When you finish running the procedure in which you had set breakpoints, VBA does not automatically remove them. To remove the breakpoint, choose Debug | Clear All Breakpoints or press Ctrl+Shift+F9. All the breakpoints are removed. If you had set several breakpoints in a procedure and would like to remove only some of them, click on the line containing the breakpoint you want to remove and press F9 (or choose Debug | Clear Breakpoint). You should clear the breakpoints when they are no longer needed. The breakpoints are automatically removed when you exit Access.

<b>NOTE</b>	<i>Remove the breakpoint you set in Custom Project 9.1.</i>
-------------	---

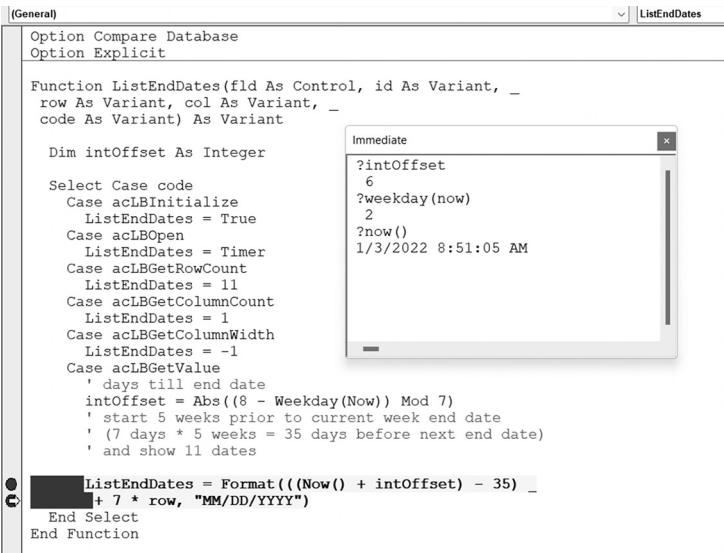
### USING THE IMMEDIATE WINDOW IN BREAK MODE

---

When the procedure execution is suspended, the Code window appears in break mode. This is a good time to activate the Immediate window and type VBA instructions to find out, for instance, the name of the open form or the value of a certain control. You can also use the Immediate window to change the contents of variables to correct values that may be causing errors. By now, you should be an expert when it comes to working in the Immediate window. Figure 9.6 shows the suspended ListEndDates function procedure and the Immediate window with the questions that were asked of Visual Basic for Applications while in break mode.

In break mode, you can also hold the mouse pointer over any variable in a running procedure to see the variable’s value. For example, in the ListEndDates function procedure shown in Figure 9.7, the breakpoint has been set on the statement just before the `End Select` keywords. When Visual Basic for Applications encounters this statement, the Code window appears in break mode.

Because the statement that stores the value of the variable `intOffset` has already been executed, you can quickly find out the value of this variable by resting the mouse pointer over its name. The name of the variable and its current value appear in a floating frame. To show the values of several variables used in a procedure, you should use the Locals window, which is discussed later in this chapter.



The screenshot shows the Microsoft Access VBA editor. In the General tab of the Properties window, there is some initial code. Below it, the code for the `ListEndDates` function is shown:

```

Option Compare Database
Option Explicit

Function ListEndDates(fld As Control, id As Variant, _
    row As Variant, col As Variant, _
    code As Variant) As Variant
    Dim intOffset As Integer
    Select Case code
        Case acLBInitialize
            ListEndDates = True
        Case acLBOpen
            ListEndDates = Timer
        Case acLBGetRowCount
            ListEndDates = 11
        Case acLBGetColumnCount
            ListEndDates = 1
        Case acLBGetColumnWidth
            ListEndDates = -1
        Case acLBGetValue
            ' days till end date
            intOffset = Abs((8 - Weekday(Now)) Mod 7)
            ' start 5 weeks prior to current week end date
            ' (7 days * 5 weeks = 35 days before next end date)
            ' and show 11 dates
    End Select
    End Function

```

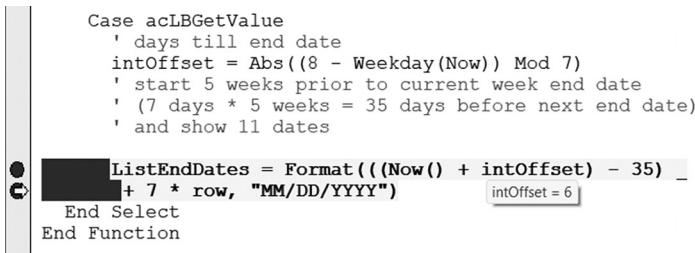
To the right of the code window, the Immediate window is open. It displays the following output:

```

?intOffset
6
?weekday(now)
2
?now()
1/3/2022 8:51:05 AM

```

**FIGURE 9.6** When code execution is suspended, you can check current values of variables and expressions by entering appropriate statements in the Immediate window.



The screenshot shows the Microsoft Access VBA editor with the code for the `ListEndDates` function. A mouse cursor is hovering over the variable `intOffset` in the line of code where it is assigned a value:

```

Case acLBGetValue
    ' days till end date
    intOffset = Abs((8 - Weekday(Now)) Mod 7)
    ' start 5 weeks prior to current week end date
    ' (7 days * 5 weeks = 35 days before next end date)
    ' and show 11 dates

    ListEndDates = Format(((Now() + intOffset) - 35) -
        + 7 * row, "MM/DD/YYYY")
    End Select
End Function

```

A floating frame appears, showing the value `intOffset = 6`.

**FIGURE 9.7** In break mode, you can find out the value of a variable by resting the mouse pointer on that variable.

### SIDE BAR Working in a Code Window in Break Mode

While in break mode, you can change code, add new statements, execute the procedure one line at a time, skip lines, set the next statement, use the

Immediate window, and more. When the procedure is in break mode, all the options on the Debug menu are available. You can enter break mode by pressing Ctrl+Break or F8 or by setting a breakpoint. In break mode, if you change a certain line of code, VBA will prompt you to reset the project by displaying the message “This action will reset your project, proceed anyway?” Click OK to stop the program’s execution and proceed editing your code or click Cancel to delete the new changes and continue running the code from the point where it was suspended. For example, change the variable declaration. As you press F5 to resume code execution, you’ll be prompted to reset your project.

## USING THE STOP STATEMENT

Sometimes you won’t be able to test your procedure right away. If you set up your breakpoints and then close the database file, the breakpoints will be removed; next time, when you are ready to test your procedure, you’ll have to begin by setting up your breakpoints again. If you need to postpone the task of testing your procedure until later, you can take a different approach by inserting a `Stop` statement into your code wherever you want to halt a procedure.

Figure 9.8 shows the `Stop` statement before the `With...End With` construct. VBA will suspend the execution of the `cboEndDate_Change` event procedure when it encounters the `Stop` statement, and the screen will display the Code window in break mode. Although the `Stop` statement has the same effect as setting a breakpoint, it does have one disadvantage: All `Stop` statements stay in the procedure until you remove them. When you no longer need to stop your procedure, you must locate and remove all the `Stop` statements.

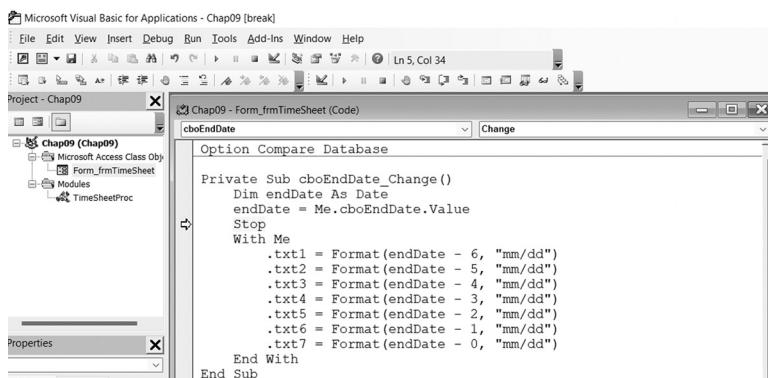


FIGURE 9.8 You can insert a `Stop` statement anywhere in your VBA procedure code. The procedure will halt when it gets to the `Stop` statement, and the Code window will appear with the code line highlighted.

## USING THE ASSERT STATEMENT

A very powerful and easy-to-apply debugging technique is utilizing `Debug.Assert` statements. Assertions allow you to write code that checks itself while running. By including assertions in your programming code, you can verify that a particular condition or assumption is true. Assertions give you immediate feedback when an error occurs. They are great for detecting logic errors early in the development phase instead of hearing about them later from your end users. Just because your procedure ran on your system without generating an error does not mean that there are no bugs in that procedure. Don't assume anything—always test for validity of expressions and variables in your code. The `Debug.Assert` statement takes any expression that evaluates to True or False and activates the break mode when that expression evaluates to False. The syntax for `Debug.Assert` is as follows:

`Debug.Assert condition`

where `condition` is a VBA code or expression that returns True or False. If `condition` evaluates to False or 0 (zero), VBA will enter break mode. For example, when running the following looping structure, the code will stop executing when the variable `i` equals 50:

```
Sub TestDebugAssert()
    Dim i As Integer
    For i = 1 To 100
        Debug.Assert i <> 50
    Next
End Sub
```

Keep in mind that `Debug.Assert` does nothing if the condition is False or zero (0). The execution simply stops on that line of code and the VBE screen opens with the line containing the false statement highlighted so that you can start debugging your code. You may need to write an error handler to handle the identified error. Error-handling procedures are covered later in this chapter. While you can stop the code execution by using the `Stop` statement (see the previous section), `Debug.Assert` differs from the `Stop` statement in its conditional aspect; it will stop your code only under specific conditions. Conditional breakpoints can also be set by using the Watches window (see the next section). After you have debugged and tested your code, comment out or remove the `Debug.Assert` statements from your final code. The easiest way to do this is to use `Edit | Replace` in the VBE editor screen. To comment out the statements, in the Find What box, enter `Debug.Assert`. In the Replace With box, enter an apostrophe followed by `Debug.Assert`.

**NOTE**

*To remove the Debug.Assert statements from your code, enter Debug.Assert in the Find What box. Leave the Replace With box empty but be sure to mark the Use Pattern Matching checkbox.*

## USING THE ADD WATCH WINDOW

Many errors in procedures are caused by variables that assume unexpected values. If a procedure uses a variable whose value changes in various locations, you may want to stop the procedure and check the current value of that variable. VBA offers a special Watches window that allows you to keep an eye on variables or expressions while your procedure is running. To add a watch expression to your procedure, select the variable whose value you want to monitor in the Code window, and then choose Debug | Add Watch. The screen will display the Add Watch dialog box, as shown in Figure 9.9.

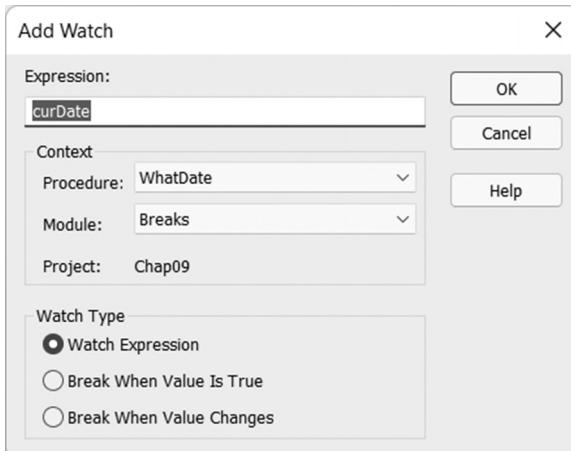


FIGURE 9.9 The Add Watch dialog box allows you to define conditions you want to monitor while a VBA procedure is running.

The Add Watch dialog box contains three sections, which are described in Table 9.2.

**TABLE 9.2** Add Watch dialog box sections.

Section	Description
Expression	Displays the name of a variable you have highlighted in your procedure. If you opened the Add Watch dialog box without selecting a variable name, type the name of the variable you want to monitor in the Expression text box.
Context	In this section, indicate the name of the procedure that contains the variable and the name of the module where this procedure is located.
Watch Type	Specifies how to monitor the variable. If you choose: <ul style="list-style-type: none"> <li>The Watch Expression option button, you can read the value of the variable in the Add Watch window while in break mode.</li> <li>Break When Value Is True, Visual Basic will automatically stop the procedure when the variable evaluates to True (nonzero).</li> <li>Break When Value Changes, Visual Basic will automatically stop the procedure each time the value of the variable or expression changes.</li> </ul>

You can add a watch expression before running a procedure or after suspending the execution of your procedure.

The difference between a breakpoint and a watch expression is that the breakpoint always stops a procedure in a specified location, but the watch stops the procedure only when the specified condition (Break When Value Is True or Break When Value Changes) is met. Watches are extremely useful when you are not sure where the variable is being changed. Instead of stepping through many lines of code to find the location where the variable assumes the specified value, you can put a watch breakpoint on the variable and run your procedure as normal. Let's see how this works.



### Hands-On 9.1 Watching the Values of VBA Expressions

1. In the Visual Basic Editor window, choose **Insert | Module** to insert a new standard module.
2. Use the Properties window to change the name of the module to **Breaks**.
3. In the Breaks Code window, type the following **WhatDate** procedure:

```

Sub WhatDate()
    Dim curDate As Date
    Dim newDate As Date
    Dim x As Integer

    curDate = Date
    For x = 1 To 365
        newDate = Date + x
    Next x
End Sub

```

The WhatDate procedure uses the `For...Next` loop to calculate the date that is `x` days in the future. You won't see any result when you run this procedure unless you insert the following instruction in the procedure code just before the `End Sub` keywords:

```
MsgBox "In " & x & " days, it will be " & NewDate
```

However, you don't want to display the individual dates, day after day. Suppose that you want to stop the program when the value of the variable `x` reaches 211. In other words, you want to know the date 211 days from now. To get the answer, you could insert the following statement into your procedure before the `Next x` statement:

```
If x = 211 Then MsgBox "In " & x & " days it will be "  
& _ NewDate
```

Instead of introducing any new statements into your procedure, simply add watch expressions to the procedure. Visual Basic for Applications will stop the `For...Next` loop when the specified condition is met, and you'll be able to check the values of the desired variables.

4. Choose **Debug | Add Watch**.
5. In the Expression text box, enter the following expression: `x = 211`.
6. In the Context section, choose **WhatDate** from the Procedure combo box and **Breaks** from the Module combo box.
7. In the Watch Type section, select the **Break When Value Is True** option button.
8. Click **OK** to close the Add Watch dialog box. You have now added your first watch expression.
9. In the Code window, position the insertion point anywhere within the name of the `curDate` variable.
10. Choose **Debug | Add Watch** and click **OK** to set up the default watch type with the **Watch Expression** option.
11. In the Code window, position the insertion point anywhere within the name of the `newDate` variable.
12. Choose **Debug | Add Watch** and click **OK** to set up the default watch type with the **Watch Expression** option.

After performing these steps, the `WhatDate` procedure contains the following three watches:

<code>x = 211</code>	Break When Value Is True
<code>curDate</code>	Watch Expression
<code>newDate</code>	Watch Expression

You can view the watches you set by choosing **View | Watch Window**.

13. Position the cursor anywhere inside the code of the WhatDate procedure and press F5.

Visual Basic stops the procedure when  $x = 211$  (see Figure 9.10). Notice that the value of the variable  $x$  in the Watches window is the same as the value you specified in the Add Watch dialog box.

In addition, the Watches window shows the value of the variables `curDate` and `newDate`. The procedure is in break mode. You can press F5 to continue, or you can ask another question: What date will be in 277 days? The next step shows how to do this.

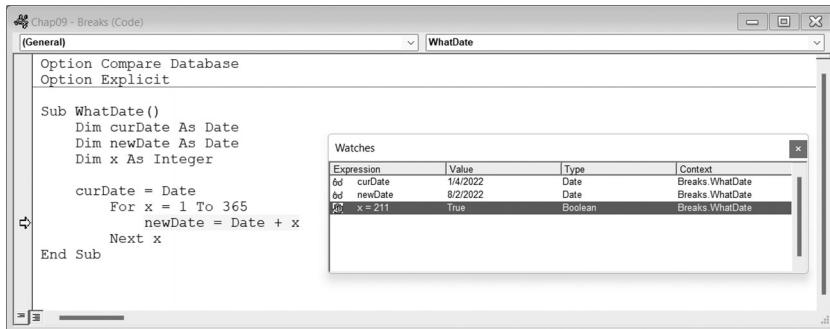


FIGURE 9.10 Using the Watches window.

14. Choose **Debug | Edit Watch** and enter the following expression:  $x = 277$ . You can also display the Edit Watch dialog box by double-clicking the expression in the Watches window.
15. Click **OK** to close the Edit Watch dialog box. Notice that the Watches window now displays a new value of the expression.  $x$  is now false.
16. Press F5. The procedure stops again when the value of  $x = 277$ . The value of `curDate` is the same; however, the `newDate` variable now contains a new value—a date that is 277 days from now. You can change the value of the expression again or finish the procedure.
17. Press F5 to finish the procedure without stopping.  
When your procedure is running and a watch expression has a value, the Watches window displays the value of the Watch expression. If you open the Watches window after the procedure has finished, you will see the error “<out of context>” instead of the variable values. In other words, when the watch expression is out of context, it does not have a value.

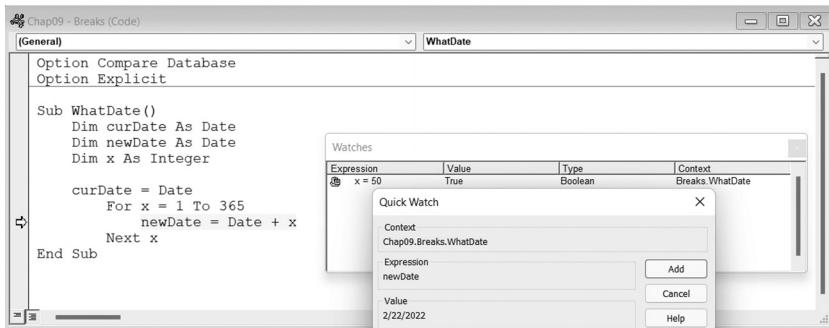
## Removing Watch Expressions

To remove a watch expression, click on the expression you want to remove from the Watches window and press Delete. Remove all the watch expressions you defined in the preceding exercise.

## USING QUICK WATCH

To check the value of an expression not defined in the Watches window, you can use Quick Watch (see Figure 9.11).

To access the Quick Watch dialog box while in break mode, position the insertion point anywhere inside a variable name or an expression you want to watch and choose Debug | Quick Watch, or press Shift+F9.



**FIGURE 9.11** The Quick Watch dialog box shows the value of the selected expression in a VBA procedure.

The Quick Watch dialog box contains an Add button that allows you to add the expression to the Watches window. Let's see how to take advantage of Quick Watch.



### Hands-On 9.2 Using the Quick Watch Dialog Box

#### NOTE

*Remove all the watch expressions you defined in Hands-On 9.1. See the preceding section on how to remove a watch expression from the Watches window.*

1. In the **WhatDate** procedure, position the insertion point on the name of the variable **x**.
2. Choose **Debug | Add Watch**.

3. Enter the expression `x = 50`.
4. Choose the **Break When Value Is True** option button and click **OK**.
5. Run the **WhatDate** procedure.

Visual Basic will suspend procedure execution when `x = 50`. Notice that the Watches window does not contain either the `newDate` or the `curDate` variables. To check the values of these variables, you can position the mouse pointer over the appropriate variable name in the Code window, or you can invoke the Quick Watch dialog box.

6. In the Code window, position the mouse inside the `newDate` variable and press **Shift+F9**, or choose **Debug | Quick Watch**.  
The Quick Watch dialog box shows the name of the expression and its current value.
7. Click **Cancel** to return to the Code window.
8. In the Code window, position the mouse inside the `curDate` variable and press **Shift+F9**, or choose **Debug | Quick Watch**.
9. The Quick Watch dialog box now shows the value of the variable `curDate`.
10. Click **Cancel** to return to the Code window.
11. Press **F5** to continue running the procedure.

## USING THE LOCALS WINDOW

---

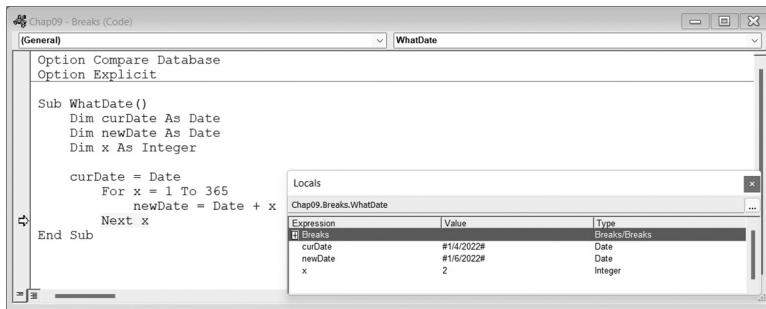
If you need to keep an eye on all the declared variables and their current values during the execution of a VBA procedure, choose **View | Locals Window** before you run your procedure. While in break mode, VBA will display a list of variables and their corresponding values in the Locals window (see Figure 9.12).

The Locals window contains three columns: Expression, Value, and Type.

The Expression column displays the names of variables that are declared in the current procedure. The first row displays the name of the module preceded by the plus sign. When you click the plus sign, you can check if any variables have been declared at the module level. Here the class module will show the system variable `Me`. In the Locals window, global variables and variables used by other projects aren't displayed.

The second column, Value, shows the current variable values. In this column, you can change the value of a variable by clicking on it and typing the new value. After changing the value, press **Enter** to register the change. You can also press **Tab**, **Shift+Tab**, or the up or down arrows, or click anywhere within the Locals window after you've changed the variable value.

Type, the third column, displays the type of each declared variable.



**FIGURE 9.12** The Locals window displays the current values of all the declared variables in the current VBA procedure.

To observe the variable values in the Locals window, let's proceed to the following hands-on exercise.



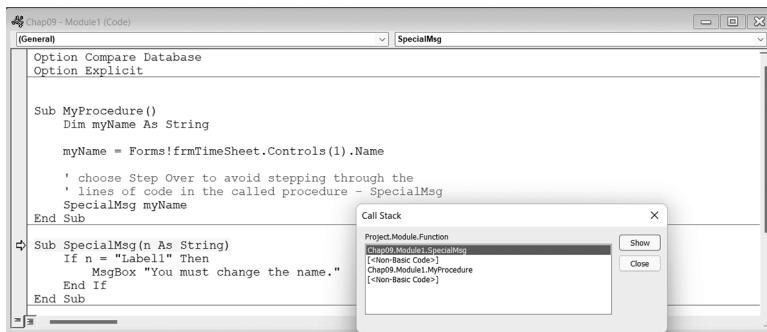
### Hands-On 9.3 Using the Locals Window

1. Choose **View | Locals Window**.
2. Click anywhere inside the **WhatDate** procedure and press **F8**.
3. Pressing F8 places the procedure in break mode. The Locals window displays the name of the current module, the local variables, and their beginning values.
4. Press **F8** a few more times while keeping an eye on the Locals window.
5. Press **F5** to continue running the procedure.

## USING THE CALL STACK DIALOG BOX

The Locals window (see Figure 9.12) contains a button with an ellipsis (...). This button opens the Call Stack dialog box (see Figure 9.13), which displays a list of all active procedure calls. An *active procedure call* is a procedure that is started but not completed. You can also activate the Call Stack dialog box by choosing **View | Call Stack**. This option is available only in break mode.

The Call Stack dialog box is especially helpful for tracing nested procedures. Recall that a nested procedure is a procedure that is being called from within another procedure (see Hands-On 9.5). If a procedure calls another, the name of the called procedure is automatically added to the Calls list in the Call Stack dialog box. When VBA has finished executing the statements of the called procedure, the procedure name is automatically removed from the Call Stack dialog box. You can use the Show button in the Call Stack dialog box to display the statement that calls the next procedure listed in the Call Stack dialog box.



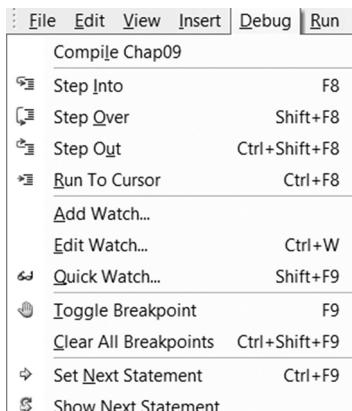
**FIGURE 9.13** The Call Stack dialog box displays a list of procedures that are started but not completed.

When the code must access external libraries to implement some functionality that VBA uses, you will see the [*<Non-Basic-Code>*] lines in the Call Stack dialog. This code cannot be debugged by you.

## STEPPING THROUGH VBA PROCEDURES

---

Stepping through the code means running one statement at a time. This allows you to check every line in every procedure that is encountered. To start stepping through the procedure from the beginning, place the cursor anywhere inside the code of your procedure and choose Debug | Step Into, or press F8. The Debug menu contains several options that allow you to execute a procedure in step mode (see Figure 9.14).



**FIGURE 9.14** The Debug menu offers many commands for stepping through VBA procedures. Certain commands on this menu are available only in break mode.

When you run a procedure one statement at a time, VBA executes each statement until it encounters the `End Sub` keywords. If you don't want to step through every statement, you can press F5 at any time to run the remaining code of the procedure without stepping through it.



### Hands-On 9.4 Stepping Through a Procedure

1. Place the cursor anywhere inside the procedure you want to trace.
2. Press F8 or choose **Debug | Step Into**.

Visual Basic for Applications executes the current statement, then automatically advances to the next statement, and suspends execution. While in break mode, you can activate the Immediate window, the Watches window, or the Locals window to see the effect of a particular statement on the values of variables and expressions. And if the procedure you are stepping through calls other procedures, you can activate the Call Stack dialog box to see which procedures are currently active.

3. Press F8 again to execute the selected statement. After executing this statement, VBA will select the next statement, and again the procedure execution will be halted.
4. Continue stepping through the procedure by pressing F8, or press F5 to continue running the code without stopping.
5. You can also choose **Run | Reset** to stop the procedure at the current statement without executing the remaining statements.

When you step over procedures (**Shift+F8**), VBA executes each procedure as if it were a single statement. This option is quite handy if a procedure contains calls to other procedures you don't want to step into because they have already been tested and debugged, or because you want to focus only on the new code that has not been debugged yet.

### Stepping Over a Procedure

Suppose that the current statement in `MyProcedure` calls the `SpecialMsg` procedure. If you choose **Debug | Step Over** (**Shift+F8**) instead of **Debug | Step Into** (F8), VBA will quickly execute all the statements inside the `SpecialMsg` procedure and select the next statement in the calling procedure, `MyProcedure`. While the `SpecialMsg` procedure is being executed, VBA continues to display the current procedure in the Code window.



### Hands-On 9.5 Stepping Over a Procedure

This hands-on exercise refers to the Access form named frmTimeSheet that you created in Custom Project 9.1 at the beginning of this chapter.

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new standard module.
2. In the module's Code window, enter the **MyProcedure** and **SpecialMsg** procedures as shown here:

```
Sub MyProcedure()
    Dim myName As String
    myName = Forms!frmTimeSheet.Controls(1).Name
    ' choose Step Over to avoid stepping through the
    ' lines of code in the called procedure - SpecialMsg
    SpecialMsg myName
End Sub

Sub SpecialMsg(n As String)
    If n = "Label1" Then
        MsgBox "You must change the name."
    End If
End Sub
```

3. Add a breakpoint within MyProcedure at the following statement:

```
SpecialMsg myName
```

4. Place the insertion point anywhere within the code of **MyProcedure** and press **F5** to run it.

Visual Basic halts execution when it reaches the breakpoint.

5. Press **Shift+F8** or choose **Debug | Step Over**.

Visual Basic runs the SpecialMsg procedure, and then execution advances to the statement immediately after the call to the SpecialMsg procedure.

6. Press **F5** to finish running the procedure without stepping through its code.  
Now suppose you want to execute MyProcedure to the line that calls the SpecialMsg procedure.

7. Click anywhere inside the statement **SpecialMsg myName**.

8. Choose **Debug | Run to Cursor**.

Visual Basic will stop the procedure when it reaches the specified line.

9. Press **Shift+F8** to step over the SpecialMsg procedure.

10. Press **F5** to execute the rest of the procedure without single stepping.

Stepping over a procedure is useful when you don't want to analyze individual statements inside the called procedure (SpecialMsg).

### **Stepping Out of a Procedure**

---

Another command on the Debug menu, Step Out (Ctrl+Shift+F8), is used when you step into a procedure and then decide that you don't want to step all the way through it. When you choose this option, Visual Basic will execute the remaining statements in this procedure in one step and proceed to activate the next statement in the calling procedure.

In the process of stepping through a procedure, you can switch between the Step Into, Step Over, and Step Out options. The option you select depends on which code fragment you wish to analyze at a given moment.

### **Running a Procedure to Cursor**

---

The Debug menu Run to Cursor command (Ctrl+F8) lets you run your procedure until the line you have selected is encountered. This command is quite useful if you want to stop the execution before a large loop or intend to step over a called procedure.

### **Setting the Next Statement**

---

At times, you may want to rerun previous lines of code in the procedure or skip over a section of code that is causing trouble. In each of these situations, you can use the Set Next Statement option on the Debug menu. When you halt execution of a procedure, you can resume the procedure from any statement you want. VBA will skip execution of the statements between the selected statement and the statement where execution was suspended.

---

**SIDE BAR** ***Skipping Lines of Code***

Although skipping lines of code can be very useful in the process of debugging your VBA procedures, it should be done with care. When you use the Next Statement option, you tell Visual Basic for Applications that this is the line you want to execute next. All lines in between are ignored. This means that certain things you may have expected to occur don't happen, which can lead to unexpected errors.

---

### **Showing the Next Statement**

---

If you are not sure where procedure execution will resume, you can choose Debug | Show Next Statement, and VBA will place the cursor on the line that will run next. This is particularly useful when you have been looking at other procedures and are not sure where execution will resume. The Show Next Statement option is available only in break mode.

## NAVIGATING WITH BOOKMARKS

---

In the process of analyzing or reviewing your VBA procedures, you will often find yourself jumping to certain areas of code. Using the built-in bookmark feature, you can easily mark the spots you want to navigate between.

To set up a bookmark:

1. Click anywhere in the statement you want to define as a bookmark.
2. Choose **Edit | Bookmarks | Toggle Bookmark** (or click the **Toggle Bookmark** button on the Edit toolbar).

Visual Basic will place a blue, rounded rectangle in the left margin beside the statement, as shown in Figure 9.15.

Once you've set up two or more bookmarks, you can jump between the marked locations of your code by choosing **Edit | Bookmarks | Next Bookmark** or simply clicking the **Next Bookmark** button on the Edit toolbar. You may also right-click anywhere in the Code window and select **Next Bookmark** from the shortcut menu. To go to the previous bookmark, select **Previous Bookmark**. You can remove bookmarks at any time by choosing **Edit | Bookmarks | Clear All** or by clicking the **Clear All Bookmarks** button on the Edit toolbar. To remove a single bookmark, click anywhere in the bookmarked statement and choose **Edit | Bookmarks | Toggle Bookmark**, or click the **Toggle Bookmark** button on the Edit toolbar.

```
Option Compare Database
Option Explicit

Sub MyProcedure()
    Dim myName As String

    myName = Forms!frmTimeSheet.Controls(1).Name

    ' choose Step Over to avoid stepping through the
    ' lines of code in the called procedure - SpecialMsg
    SpecialMsg myName
End Sub

Sub SpecialMsg(n As String)
    If n = "Label1" Then
        MsgBox "You must change the name."
    End If
End Sub

Sub TestDebugAssert()
    Dim i As Integer
    For i = 1 To 100
        Debug.Assert i <> 50
    Next
End Sub
```

FIGURE 9.15 Using bookmarks, you can quickly jump between often-used sections of your procedures.

## STOPPING AND RESETTING VBA PROCEDURES

---

At any time while stepping through the code of a procedure in the Code window, you can press F5 to execute the remaining instructions without stepping through them or choose Run | Reset to finish the procedure without executing the remaining statements. When you reset your procedure, all the variables lose their current values. Numeric variables assume the initial value of zero (0), variable-length strings are initialized to a zero-length string (""), and fixed-length strings are filled with the character represented by the ASCII character code 0, or Chr(0). Variant variables are initialized to Empty, and the value of Object variables is set to Nothing.

## TRAPPING ERRORS

---

No one writes bug-free programs the first try. For this reason, when you create VBA procedures you have to determine how your program will respond to errors. Many unexpected errors happen at runtime. For example, your procedure may try to give a new file the same name as an open file.

Runtime errors are often discovered not by a programmer but by the user who attempts to do something that the programmer has not anticipated. If an error occurs when the procedure is running, Visual Basic displays an error message and the procedure is stopped. The error message that VBA displays to the user is often quite cryptic.

You can keep users from seeing many runtime errors by including error-handling code in your VBA procedures. This way, when Visual Basic encounters an error, instead of displaying a default error message, it will show a much friendlier, more comprehensive error message, perhaps advising the user how to correct the error or whom to contact.

How do you implement error handling in your VBA procedure? The first step is to place the `On Error` statement in your procedure. This statement tells VBA what to do if an error happens while your program is running. In other words, VBA uses the `On Error` statement to activate an error-handling procedure that will trap runtime errors. Depending on the type of procedure, you can exit the error trap by using one of the following statements: `Exit Sub`, `Exit Function`, `Exit Property`, `End Sub`, `End Function`, or `End Property`.

You should write an error-handling routine for each procedure. Table 9.3 shows how the `On Error` statement can be used.

**TABLE 9.3** On Error statement options.

On Error Statement	Description
On Error GoTo Label	Specifies a label to jump to when an error occurs. This label marks the beginning of the error-handling routine. An <i>error handler</i> is a routine for trapping and responding to errors in your application. The label must appear in the same procedure as the On Error GoTo statement.
On Error Resume Next	When a runtime error occurs, Visual Basic ignores the line that caused the error and continues the procedure with the next line. An error message is not displayed.
On Error GoTo 0	Turns off error trapping in a procedure. When VBA runs this statement, errors are detected but not trapped within the procedure.

**SIDE BAR** *Is This an Error or a Mistake?*

In programming, mistakes and errors are not the same thing. A mistake—such as a misspelled or missing statement, a misplaced quotation mark or comma, or an assignment of a value of one type to a variable of a different (and incompatible) type—can be removed from your program through proper testing and debugging. But even though your code may be free of mistakes, errors can still occur. An *error* is a result of an event or operation that doesn't work as expected. For example, if your VBA procedure accesses a certain file on disk and someone deleted this file or moved it to another location, you'll get an error no matter what. An error prevents the procedure from carrying out a specific task.

## Using the Err Object

Your error-handling code can utilize various properties and methods of the `Err` object. For example, to check which error occurred, check the value of `Err.Number`. The `Number` property of the `Err` object will tell you the value of the last error that occurred, and the `Description` property will return a description of the error. You can also find the name of the application that caused the error by using the `Source` property of the `Err` object (this is very helpful when your procedure launches other applications). After handling the error, use the `Err.Clear` statement to reset the error number. This will set `Err.Number` back to zero.

To test your error-handling code you can use the `Raise` method of the `Err` object. For example, to raise the “Disk not ready” error, use the following statement:

```
Err.Raise 71
```

The following OpenToRead procedure demonstrates the use of the `On Error` statement and the `Err` object.



## Hands-On 9.6 Error-Trapping Techniques

1. Copy the `Vacation.txt` file from the companion files to your `VBAAccess2021_ByExample_Primer` folder.
2. In the Visual Basic Editor window, insert a new module and rename it `ErrorTraps`.
3. In the Code window, enter the following `OpenToRead` procedure:

```
Sub OpenToRead()
    Dim strFile As String
    Dim strChar As String
    Dim strText As String
    Dim FileExists As Boolean

    FileExists = True

    On Error GoTo ErrorHandler

    strFile = InputBox("Enter the name of file to open:")
    Open strFile For Input As #1

    If FileExists Then
        Do While Not EOF(1) ' loop until the end of file
            strChar = Input(1, #1) ' get one character
            strText = strText + strChar
        Loop
        Debug.Print strText
        ' Close the file
        Close #1
    End If
    Exit Sub

ErrorHandler:
    FileExists = False
    Select Case Err.Number
        Case 71
            MsgBox "The CD/DVD drive is empty."
        Case 53
            MsgBox "This file can't be found on the specified drive."
    End Select
End Sub
```

```
Case 76
    MsgBox "File Path was not found."
Case Else
    MsgBox "Error " & Err.Number & " :" & Err.Description
    Exit Sub
End Select
Resume Next
End Sub
```

Before continuing with this hands-on, let's examine the code of the OpenToRead procedure. The purpose of this procedure is to read the contents of the user-supplied text file character by character. When the user enters a filename, various errors can occur. For example, the filename may be wrong, the user may attempt to open a file from a CD-ROM or DVD disc without placing the disc in the drive, or he may try to open a file that is already open. To trap these errors, the error-handling routine at the end of the OpenToRead procedure uses the `Number` property of the `Err` object. The `Err` object contains information about runtime errors. If an error occurs while the procedure is running, the statement `Err.Number` will return the error number.

If errors 71, 53, or 76 occur, Visual Basic will display the user-friendly messages given inside the `Select Case` block and then proceed to the `Resume Next` statement, which will send it to the line of code following the one that had caused the error. If another (unexpected) error occurs, Visual Basic will return its error code (`Err.Number`) and error description (`Err.Description`).

At the beginning of the procedure, the variable `FileExists` is set to `True`. If the program doesn't encounter an error, all the instructions inside the `If FileExists Then` block will be executed. However, if VBA encounters an error, the value of the `FileExists` variable will be set to `False` (see the first statement in the error-handling routine just below the `ErrorHandler` label).

If you comment the `Close #1` instruction, Visual Basic will encounter the error on the next attempt to open the same file. Notice the `Exit Sub` statement before the `ErrorHandler` block. Put the `Exit Sub` statement just above the error-handling routine. You don't want Visual Basic to carry out the error handling if there are no errors.

How does this procedure accomplish the read operation? The `Input` function allows you to return any character from a sequential file. *Sequential access files* are files where data is retrieved in the same order as it is stored, such as files stored in the CSV format (comma-delimited text), TXT format (text separated by tabs), or PRN format (text separated by spaces). Configuration files, error logs, HTML files, and all sorts of plain text files are all sequential

files. These files are stored on disk as a sequence of characters. The beginning of a new text line is indicated by two special characters: the carriage return, and the linefeed. When you work with sequential files, start at the beginning of the file, and move forward character by character, line by line, until you encounter the end of the file. Sequential access files can be easily opened and manipulated by just about any text editor.

If you use the VBA function named `LOF` (length of file) as the first argument of the `Input` function, you can quickly read the contents of the sequential file without having to loop through the entire file.

For example, instead of the following `Do...While` loop statement block:

```
Do While Not EOF(1) ' loop until the end of file
    strChar = Input(1, #1) ' get one character
    strText = strText + strChar
Loop
```

you can simply write the following statement to get the contents of the file at once:

```
strText = Input(LOF(1), #1)
```

The `LOF` function returns the number of bytes in a file. Each byte corresponds to one character in a text file.

To read data from a file, you must first open the file with the `Open` statement using the following syntax:

```
Open pathname For mode[Access access][lock] As [#]filenumber _
[Len=reclength]
```

The `Open` statement has three required arguments: `pathname`, `mode`, and `filenumber`. `Pathname` is the name of the file you want to open. The filename may include the name of a drive and folder.

`Mode` is a keyword that determines how the file was opened. Sequential files can be opened in one of the following modes: `Input`, `Output`, or `Append`. Use `Input` to read the file, `Output` to write to a file and overwrite any existing file and `Append` to write to a file by adding to any existing information.

`Filenumber` is a number from 1 to 511. This number is used to refer to the file in subsequent operations. You can obtain a unique file number using the VBA built-in `FreeFile` function.

The optional `Access` clause can be used to specify permissions for the file (Read, Write, or Read Write). The optional `lock` argument determines which file operations are allowed for other processes. For example, if a file is open

in a network environment, `lock` determines how other people can access it. The following `lock` keywords can be used: Shared, Lock Read, Lock Write, or Lock Read Write. The last element of the `Open` statement, `reclength`, specifies the buffer size (total number of characters) for sequential files.

Therefore, to open a sequential file in order to read its data, the example procedure uses the following instruction:

```
Open strFile For Input As #1
```

And to close the sequential file, the following statement is used:

```
Close #1
```

4. Click anywhere within the `OpenToRead` procedure and press F5 to run it. When prompted for the file to open, type `C:\VBAAccess2021_ByExample_Primer\Vacation.txt` in the input dialog box and click **OK**. The procedure reads the contents of the `Vacation.txt` file into the Immediate window.
5. Run the `OpenToRead` procedure again. When prompted for the file to open, type `P:\VBAAccess2021_ByExample_Primer\Vacation.txt` in the input dialog box and click **OK**. This time Visual Basic cannot find the specified file, so it displays the message “File Path was not found.”
6. Run the `OpenToRead` procedure again. This time, when prompted for the filename, enter the name of any file that references your CD/DVD drive (when the drive slot is empty). This should trigger error 71 and result in the message “The CD/DVD drive is empty.” Skip this step if you don’t have a CD/DVD drive.
7. Comment the `Close #1` statement and run `OpenToRead`. When prompted for the file, enter `C:\VBAAccess2021_ByExample_Primer\Vacation.txt` as the filename. Run the same procedure again, supplying the same filename. The second run will cause the statements within the `Case Else` block to run. You should get an error 55 “File already open” message because the text file will still be open in memory. To remove the file from memory, type `Close #1` in the Immediate window and press **Enter**. Next, uncomment the `Close # 1` statement in the `OpenToRead` procedure to return it to the original state.

## Procedure Testing

---

You are responsible for the code you write. Before you give your procedure to others to test, you should test it yourself. After all, you understand best how it is supposed to work. Some programmers think testing their own code is some sort of degrading activity, especially when they work in an organization that has

a team devoted to testing. *Don't make this mistake.* The testing process at the programmer level is as important as the code development itself. After you've tested the procedure yourself, you should give it to the users to test. Users will provide you with answers to questions such as: Does the procedure produce the expected results? Is it easy and fun to use? Does it follow the standard conventions? Also, it is a good idea to give the entire application to someone who knows the least about using this type of application and ask them to play around with it and try to break it.

You can test the ways your program responds to runtime errors by causing them on purpose:

- Generate any built-in error by entering the following syntax:

```
Error error_number
```

For example, to display the error that occurs on an attempt to divide by zero (0), type the following statement in the Immediate window:

```
Error 11
```

When you press Enter, Visual Basic will display the error message saying, "Run-time error 11. Division by zero."

- To check the meaning of the generated error, use the following syntax:

```
Error(error_number)
```

For example, to find out what error number 7 means, type the following in the Immediate window:

```
?Error(7)
```

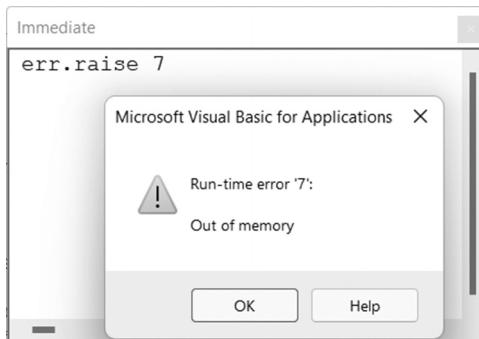
When you press Enter, Visual Basic returns the error description:

```
"Out of memory"
```

To generate the same error at runtime in the form of a message box like the one in Figure 9.16, enter the following in the Immediate window or in your procedure code:

```
Err.Raise 7
```

When you finish debugging your VBA procedures, make sure you remove all statements that raise errors.



**FIGURE 9.16** To test your error-handling code, use the `Raise` method of the `Err` object. This will generate a runtime error during the execution of your procedure.

When testing your VBA procedure, use the following guidelines:

- If you want to analyze your procedure, step through your code one line at a time by pressing F8 or by choosing Debug | Step Into.
- If you suspect that an error may occur in a specific place in your procedure, use a breakpoint.
- If you want to monitor the value of a variable or expression used by your procedure, add a watch expression.
- If you are tired of scrolling through a long procedure to get to sections of code that interest you, set up a bookmark to quickly jump to the desired location.

### Setting Error-Trapping Options

You can specify the error-handling settings for your current Visual Basic project by choosing Tools | Options and selecting the General tab (shown in Figure 9.17). The Error Trapping area located on the General tab determines how errors are handled in the Visual Basic environment. The following options are available:

- Break on All Errors  
This setting will cause Visual Basic to enter the break mode on any error, no matter whether an error handler is active or whether the code is in a class module (class modules were covered in Chapter 8).
- Break in Class Module  
This setting will trap any unhandled error in a class module. Visual Basic

will activate the break mode when an error occurs and will highlight the line of code in the class module that produced this error.

- Break on Unhandled Errors

This setting will trap errors for which you have not written an error handler. The error will cause Visual Basic to activate the break mode. If the error occurs in a class module, the error will cause Visual Basic to enter break mode on the line of code that called the offending procedure of the class.

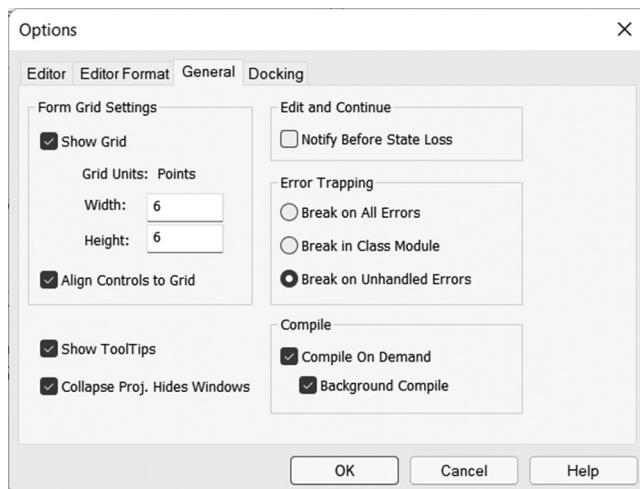


FIGURE 9.17 Setting the error-trapping options in the Options dialog box will affect all instances of Visual Basic started after you change the setting.

## SUMMARY

---

In this chapter, you learned how to test your VBA procedures to make sure they perform as planned. You debugged your code by stepping through it using breakpoints and watches. You learned how to work with the Immediate window in break mode; you found out how the Locals window can help you monitor the values of variables; and you learned how the Call Stack dialog box can be helpful in keeping track of where you are in a complex program. You also learned how to mark your code with bookmarks so you can easily navigate between sections of your procedure. Additionally, this chapter showed you how to trap errors by

including an error-handling routine inside your VBA procedure and how to use the VBA Err object.

By using the built-in debugging tools, you can quickly pinpoint the problem spots in your Access VBA procedures. Try to spend more time getting acquainted with the Debug menu options and debugging tools discussed in this chapter. Mastering the art of debugging can save you hours of trial and error.

This chapter completes the fundamentals of VBA programming. With the basics mastered, you are now ready to delve into more advanced programming topics that will allow you to take full control of your Access database applications.

Part



# *ACCESS VBA PROGRAMMING WITH DAO AND ADO*

**T**here are two sets of programming objects known as Data Access Objects (DAO) and ActiveX® Data Objects (ADO) that enable Microsoft Access and other client applications to access and manipulate data. In this part of the book, you learn how to use DAO and ADO objects in your VBA procedures to connect to a data source; create, modify, and secure database objects; and read, add, update, and delete data.

- Chapter 10 Data Access Technologies in Microsoft Access
- Chapter 11 Creating and Manipulating Databases with DAO
- Chapter 12 Creating and Manipulating Databases with ADO



# Chapter 10 DATA ACCESS TECHNOLOGIES IN MICROSOFT ACCESS

Database access is quite a complex task. Before you delve into programming, you need to understand the choices that Access provides for accessing data programmatically. In this chapter, you learn about two database engines that Access uses: the older (Jet) and the newer (ACE). Next, we look at versions and file formats supported by Access 2021 and discuss the importance of setting up various library references in your VBA modules. After reviewing various libraries that provide objects for your procedures, you learn about connection strings. All this information is important, as you may need to access your database not only from Access itself but also from various external applications. Nothing great can be achieved until you are able to make a database connection.

## **UNDERSTANDING DATABASE ENGINES: JET/ACE**

---

Since version 1.0 (1992), an integrated part of Microsoft Access has been its database engine, commonly referred to as *Microsoft Jet* (Joint Engine Technology (JET)) or *Jet database engine*. Microsoft Jet is a multiuser relational database engine that provides support for the standard DBMS (Database Management

System) functionality such as data definition, data manipulation, querying, security, and maintenance, as well as remote data access.

Jet stores data in the Microsoft Access database file format (.mdb) according to the Indexed Sequential Access Method (ISAM). Queries are performed by the Jet query engine. A replication engine is used to create copies (replicas) of database structures on multiple systems with periodic synchronization. Jet provides password-protected security and different levels of access via the user and group accounts. The user information is kept in a separate system database (MDW). Security is also built into the database tables in the form of object permissions.

The Microsoft Jet database engine enables you to access data that resides in Microsoft Jet databases (.mdb files), external data sources such as Microsoft Excel spreadsheets, SharePoint lists, Microsoft Outlook folders, legacy dBASE files, text files, XML files, or HTML documents, as well as Open Database Connectivity (ODBC) data sources such as SQL Server<sup>\*</sup>, and Oracle. To access external data via ODBC, you need a specific ODBC driver installed on the computer containing the data source.

The main component of the Microsoft Jet database engine is a *dynamic link library* file (.dll) (see Table 10.1). On the Windows platform, DLLs are libraries of common code that can be used by more than one application. The Jet DLL provides a simple interface to the data. If the data source is an .mdb file, then Jet reads and writes directly to the file. If the data source is external, Jet calls on the appropriate ODBC driver to perform the request.

Different versions of Access use different versions of Jet (see Table 10.1).

Beginning with Office Access 2007, Microsoft made many enhancements to the database engine, making it private for Office suite applications. This private version of the database engine, called the Access Connectivity Engine (ACE), uses the file extension .accdb and offers many useful features to Access users and developers alike (see Table 10.2).

**TABLE 10.1** Database engine versions in Access 2021 and earlier

MS Access Version	Database Engine Used	Dynamic Link Library (DLL) File
Access 2007–2021	ACE 12	ace.dll
Access 2000–2003	Jet 4.0	msjet40.dll
Access 97	Jet 3.5	msjet35.dll

## UNDERSTANDING ACCESS VERSIONS AND FILE FORMATS

---

In Access 2007–2021, the default file format is .accdb; however, you can still directly open and use Jet databases (.mdb files) created in Access 2000–2003. Jet databases created with Access 97 or earlier must be either enabled or converted for use in Access 2007–2021. When an older database is enabled, it is made compatible with Access so that you can make changes to the data. However, any design changes must be made in the version of Access that was used when the database was first created. When you opt to convert an Access 97 or earlier database to the .accdb file format, you must first convert it to Access 2000–2003. Table 10.2 lists various file formats that are supported since the release of Access 2007.

**TABLE 10.2** File formats supported in Access 2007–2021

File Format	Description	Additional Notes
.accdb	<p>File format first introduced in Access 2007 (default).</p> <p>This file format is not readable by Access versions prior to 2007.</p> <p>DO NOT use this file format if you need to support:</p> <ul style="list-style-type: none"> <li>• Replication</li> <li>• User-level security</li> </ul>	<p><b>Note:</b> Access 2013–2021 do not support replicated databases.</p> <p>Use Access 2010–2007 to create a replica of an MDB database formatted in Access 2000–2003 file format.</p>
.accde	<p>File extension for Access 2007–2021 .accdb files that are in execute only mode. These files have all VBA source code removed.</p> <p>This file extension replaces the .mde file extension used in earlier versions of Access.</p>	<p>Users can only execute VBA code; they cannot view or modify it. In addition, users do not have permissions to make design changes to forms or reports.</p> <p>If you need to save the Access 2021 database in .accde format, open the database and choose File   Save As   Save Database As. Select Make ACCDE and click the Save As button.</p>
.accdt	<p>This is an Access Database Template file. Access 2007–2021 all come with professionally designed database templates.</p>	<p>Templates provide you with predefined tables/table relationships, forms, reports, queries, and macros.</p>

(Contd.)

File Format	Description	Additional Notes
		To save the Access 2021 database as a template, open the database and choose File   Save As   Save Database As. Select Template (.accdt) and click the Save As button.
.accdr	This file extension denotes an Access 2007–2021 database functioning in runtime mode.	To create a “locked-down” version of your Access 2021 database, simply change the file extension from .accdb to .accdr. To restore the full database functionality, do the reverse: change the file extension from .accdr to .accdb.
.mdb (Access 97, Access 2000, Access 2002, Access 2003)	Access database file format used in versions prior to 2007. Note: In Access 2007–2021 you can create files in either the Access 2000 format or the Access 2002–2003 database format. These files will have the extension .mdb.	Use the .mdb file format if the database will be used in earlier versions of Access to: <ul style="list-style-type: none"><li>• Support replication</li><li>• Support user-level security</li></ul>
.mde (Access 97, Access 2000, Access 2002, Access 2003)	An .mde file is a compiled version of an .mdb database without any VBA code. This change prevents a database user from reading or changing your VBA code. Users cannot edit the design of forms, reports, or modules.	An .accde file is the Access 2007–2021 version of the .mde file in earlier versions of Access.
.adp	This is a file extension for a Microsoft Access Data Project file that lets you connect to an SQL Server database or the Microsoft Data Engine (MSDE) on your PC and create client/server applications. A project file does not contain any data or data definition objects such as tables, views, stored procedures, or user-defined	Access 2013–2021 does not support the .adp file format. If you need to open and edit an existing ADP database that was created in an earlier version of Access or create a new ADP database, use Access 2007–2010.

File Format	Description	Additional Notes
	functions. All database objects are stored in the SQL Server database. An .adp file stores only database frontend forms, reports, and other application objects (macros, modules).	
.ade	This is a file format for a Microsoft Access project (.adp) file with all modules compiled and all editable source code removed. Similar to .mde files, projects stored in the .ade file format prevent users from making design changes to the frontend and gaining access to your VBA source code.	Access 2021 does not support the .ade file format. To create an .ade file from your Access Data Project (ADP), use Access 2007–2010.
.mdw (Access 97, Access 2000, Access 2002, Access 2003)	This file format is used by a Workgroup Information File. The .mdw files store information for secured MDB databases.	There are no changes to the .mdw file format in Access 2016–2021. The .mdw files created in earlier versions of Access (2000 through 2003) can be used by Access 2016–2021. When an MDB database is opened, you can choose File   Info   Users & Permissions   User-Level Security Wizard to create a new Workgroup Information File (.mdw).
.ldb	This is a locking file extension for the MDB database. This file prevents users from writing data to pages that have been locked by other users and lets you determine which computer/user has a file or record locked. The .ldb file keeps track of usernames/computer names of the people who are currently logged into the MDB.	A locking file is created automatically when the database is opened and is deleted automatically when the last user closes a shared database. <b>Note:</b> You can view the information stored in this file by opening it with Windows Notepad.

(Contd.)

File Format	Description	Additional Notes
.accordion	This is the file extension for a locking file used by Access 2007–2021 (.accdb file format).	As with the .ldb file, the .accdb file is created automatically when the database is opened and is deleted automatically when the last user closes a shared database. Note: Because different locking files are created for MDB and ACCDB databases in Access 2007–2021, .mdb and .accdb files can be opened in Access 2007–2021 without causing conflicts in the locking file.

## UNDERSTANDING LIBRARY REFERENCES

---

A Microsoft Access database consists of various types of objects stored in different object libraries. *Libraries* are components that provide specific functionality. They are listed in the References dialog box, shown in Figure 10.1, which can be opened from the Visual Basic Editor window by selecting Tools | References. If you create an Access 2021 database in the default .accdb file format, you will see the following default references in the References dialog box:

- Visual Basic for Applications
- Microsoft Access 16.0 Object Library
- OLE Automation
- Microsoft Office 16.0 Access database engine Object Library

The Visual Basic for Applications and Access libraries that appear at the top of the References dialog box are built in. Access will not allow you to remove them from the database. The references that are checked are listed by priority. References that are not checked are listed alphabetically, other than the few exceptions seen in Figure 10.1. When your VBA procedure references an object, Visual Basic searches each referenced object library in the order in which the libraries are displayed in the References dialog box. If the referenced libraries have objects with the same name, Visual Basic uses the object definition provided by the library listed higher in the Available References list. You can change the priority of an object library by selecting its name and clicking the up or down arrow button in the References dialog box. To help Visual Basic resolve

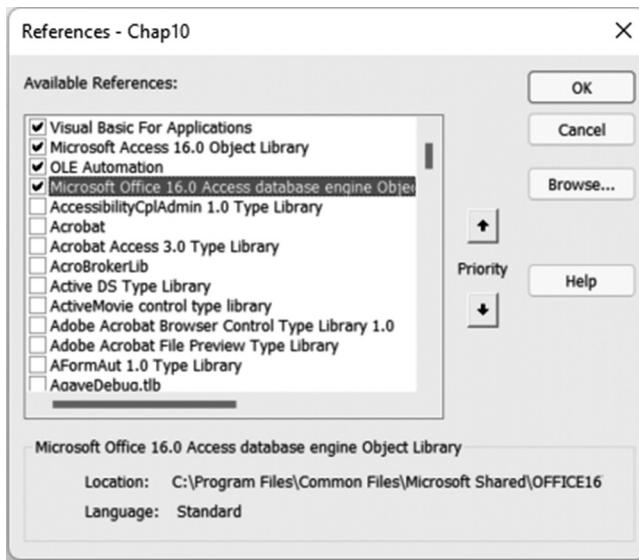


FIGURE 10.1. The default object libraries for Access 2021.

library references, specify in your code the name of the library you intend to use. For example, to specify that the DAO Recordset should be used, declare it like this:

```
Dim rst As DAO.Recordset
```

To use the ADO Recordset, use the following declaration:

```
Dim rst As ADODB.Recordset
```

You can reference additional libraries in your Access database if your VBA application requires features that are not provided by the default libraries. For example, if your VBA procedures need to access files and folders on the computer, you may want to check the box next to Microsoft Scripting Runtime.

**NOTE**

*Do not add references to libraries you don't plan to use as they consume memory and may make your Access VBA project more time-consuming to compile and harder to debug.*

**SIDE BAR** *Missing Library*

If the library is marked as Missing in the References dialog box, click the Browse button, and locate the correct library file. You can disable a missing reference by clearing the checkbox to the left of the reference labeled "Missing."

**SIDE BAR** *Library Does Not Show in the References Dialog Box*

If the library you want to reference is not shown in the Available References list box, you may need to unregister and reregister it with Windows.

To unregister a library, close Microsoft Access. Choose Run from the Start menu (or press Windows Key + R) and enter `regsvr32 -u` followed by a space and the full path to the library file surrounded by quotation marks. For example, to unregister `msjro.dll`, enter the following:

```
regsvr32 -u "C:\Program Files\Common Files\System\Ado\msjro.dll"
```

To register a library, repeat the same code but without the `-u` switch. For example:

```
regsvr32 "C:\Program Files\Common Files\System\Ado\msjro.dll"
```

The `regsvr32` command registers the DLL file whether it is 64-bit or 32-bit. If you are running a 64-bit Windows version, the 32-bit files are in `Windows\SysWOW64` folder and the 64-bit DLL files are in `Windows\System32` folder. If your Windows is 32-bit, the DLL files are in `Windows\System32` folder and there is no `SysWOW64` folder.

If you are using a 64-bit Windows and you want to register a DLL file which is 32-bit, follow these steps:

In the File Explorer, activate the `Windows\System32` folder.

Copy the DLL file and paste it into `Windows\SysWOW64` folder.

Press Windows Key + R, and enter the following:

```
C:\Windows\SysWOW64\regsvr32 msjro.dll
```

Click OK to proceed with registration.

If you run into any errors when performing the above steps, try to disable the User Account Control (UAC) in Windows that may be preventing you from successfully registering the DLL file. If you are still having an issue, try to perform the steps in an elevated Command prompt. To do this, click Start, and type `cmd` and choose Run as administrator, and click Yes in the UAC dialog. The exact steps may vary depending on your version of Windows.

Once you register a missing library, the library name should be listed in the References dialog box next time you open Access.

**NOTE**

*If you move a library file from where it was originally installed, be sure to reregister it with the operating system or things may not work as expected.*

Because referencing a wrong library for the version of Access used can cause data corruption, it is important to know which library files were designed for a particular version of Access. The next section introduces you to library files that you will find useful in creating and manipulating databases in the .MDB and .ACCDB file formats using VBA code.

## **OVERVIEW OF OBJECT LIBRARIES IN MICROSOFT ACCESS**

The object library contains information about its objects, properties, and methods. To work with the VBA programming examples included in this book, you will need to access objects from the libraries listed in the following subsections.

### **The Visual Basic for Applications Object Library (VBA)**

Objects contained in this library allow you to access your computer's file system, work with date and time functions, perform mathematical and financial computations, interact with users, convert data, and read text files. The VBA library is stored in the VBE7.DLL file.

### **The Microsoft Access 16.0 Object Library**

This library provides objects that are used to display data and work with the Microsoft Access application. In Access 2016-2021, the Access library is stored in the MSACC.OLB file.

### **OLE Automation**

Object Linking and Embedding (OLE) library allows you to embed or link an object from another application. In Access 2016-2021, the library is stored in the STDOLE2.TLB file.

### **The Microsoft Office 16.0 Access Database Engine Object Library**

This library is the enhanced version of the DAO Object Library. It was built specifically for working with the ACE database engine. In Access 2016-2021, the library is stored in the ACEDAO.DLL file. This library is used when you open an Access database in the default Access format (.accdb).

### **The Microsoft DAO 3.6 Object Library**

This library is stored in the DAO360.DLL file and is used by Access MDB databases created in Access 2000 through 2021. Access 97 uses DAO350.DLL.

DAO provides programmatic access to Jet Access databases. It consists of a hierarchy of objects that supply methods and properties for designing and manipulating databases. The DBEngine object positioned at the top of the DAO object hierarchy is often referred to as the Jet engine and is used to reference the database engine as a whole. All the other objects and collections in the DAO object hierarchy fall under DBEngine. The DBEngine contains the following two collections of objects:

- The Errors collection, which stores a list of errors that have occurred in the DBEngine. These errors are represented by the Error objects and should not be confused with the Err object, which stores runtime errors generated in Visual Basic.
- The Workspaces collection (the default collection of the DBEngine object), which contains the Workspace objects and is used for database security in multiuser applications. The Workspace object is used in conjunction with User and Group objects.

Each open database is represented by the Database object. The Database object is used to reference a Microsoft Access database file (.mdb) or another external database represented by an ODBC data source. The Databases collection contains all currently open databases. The Containers, QueryDefs, Relations, and TableDefs collections contain objects that are used to reference various components of the Database object. For example, the TableDef object represents a table or a linked table in a Microsoft Jet workspace. The QueryDef object represents a query in DAO. If values are supplied to a query, they are represented in DAO by a Parameter object. The Parameters collection contains all of the Parameter objects defined for a QueryDef object. The Relation object represents a relationship between fields in tables and queries. The Container object is used to access collections of saved objects that represent databases, tables, queries, and relationships.

The Recordsets collection contains all open Recordset objects. Each Recordset object represents a set of records within a database. You will use Recordset objects for retrieving, adding, editing, and deleting records from a database.

The Field object represents a field in a table, query, index, relation, or recordset. The Fields collection is the default collection of a TableDef, QueryDef, Index, Relation, or Recordset object.

Some DAO objects have a Properties collection. The Properties collection contains a separate object for each property of the DAO object that is referenced. You can use an object's Properties collection to enumerate its properties or to return their settings. You can also define your own custom properties on DAO objects. You will work with DAO objects in Chapter 11.

### The Microsoft ActiveX Data Objects 6.1 Library (ADO)

This library is stored in the MSADO15.DLL file. ActiveX Data Objects (ADO) that are provided by this library are used for accessing and manipulating data from a variety of sources through an OLE DB provider.

**NOTE**

*If you scroll down the list of the Available References (Figure 10.1), you may get confused to see several different versions of the Microsoft ActiveX Data Objects Library. Which version you should use depends on whether the users of your Access applications are on Windows 7 and above or Vista and XP. For Windows 7 and above, use version 6.1 of this library. For Windows Vista, stick to version 6.0 which came with Vista, and for Windows XP SP3 or Windows Server 2003 SP1, select version 2.8 or lower.*

ADO works with the technology known as *OLE DB*. This technology is object-based, but it is not limited to relational databases. OLE DB can access both relational and non-relational data sources such as directory services, mail stores, multimedia, and text files, as well as mainframe data (VSAM and MVS). You do not need any specific drivers installed on your computer to access external data with OLE DB because OLE DB does not use drivers; it uses data providers to communicate with data stores. *Data providers* are programs that enable access to data. OLE DB has many providers, such as Microsoft OLE DB Provider for SQL Server and Microsoft Jet 4.0 OLE DB Provider. There are also providers for Oracle, Active Directory<sup>®</sup>, and ODBC.

Similar to DAO, ADO objects make it possible to establish a connection with a data source in order to read, insert, modify, and delete data. ADO offers to programmers many advanced features that are not available in DAO. For example, the ADO Connection object's State property lets you determine whether the connection is closed (adStateClosed), open and ready (adStateOpen), still trying to connect (adStateConnecting), processing a command (adStateExecuting), or fetching data (adStateFetching). The ADO Recordsets can be hierarchical, fabricated, disconnected, or persisted on disk.

ADO consists of three object models, each providing a different area of functionality (see Table 10.3). Because of this, only the objects necessary for a specific task need to be loaded at any given time.

**TABLE 10.3.** Components of ADO

Object Model	What It's Used For
ADODB (ActiveX Data Objects)	Data manipulation Access and manipulate data through an OLE DB provider. With ADO objects you can connect to a data source and read, add, update, or delete data. Library Name: Microsoft ActiveX Data Objects 6.1 Library Library File: msado15.dll
ADOX (ADO Extensions for DDL and Security)	Data definition and security With ADOX objects you can define data such as tables, views, indexes, or relationships, as well as create and modify user and group accounts, and grant and revoke permissions on objects. Library Name: Microsoft ADO Ext. 6.0 for DDL and Security (ADOX) Library File: msadox.dll
JRO (Jet and Replication Objects)	Replication (used with .mdb databases only) With JRO objects you can compact a Jet database, and create, modify, and synchronize replicas. JRO can be used only with Microsoft Jet databases. Library Name: Microsoft Jet and Replication Objects 2.6 Library (JRO) Library File: msjro.dll This is a 32-bit library that is not supported on the 64-bit Windows systems.

You will work with ADO objects in Chapter 12.

<b>NOTE</b>	<i>Access 2000 was the first version to support ADO. In an attempt to promote universal data access, Microsoft made ADO the default library in Access 2000 and 2002. DAO was to be phased out and Access programmers were advised to move their application code from DAO to ADO. Since then, having found out that DAO still performed faster in most cases, was easier to use, and offered features that were specifically designed with Jet/ODBC databases in mind, Microsoft has returned to DAO as the main data access layer. In Access 2007, DAO was enhanced to use the new data types and other improvements available in the .accdb format. This enhanced version of DAO was offered as the Microsoft Office 12.0 Access database engine Object Library. In Access 2016-2021, it is offered as the Microsoft Office 16.0 Access database engine Object Library.</i>
-------------	---

**SIDE BAR** *ADO Classic versus ADO.NET*

The classic ADO used in VBA in Microsoft Access and other Office applications is a completely different object model from ADO.NET used with the Microsoft .NET framework. ADO.NET is not built on ActiveX technology and its objects cannot be used directly in a VBA project.

## CREATING A REFERENCE TO THE ADO LIBRARY

Prior to declaring variables as ADO objects in your VBA procedures, make sure that the reference to the library you are intending to use is set in the References dialog box. Hands-On 10.1 demonstrates how to create a reference to the Microsoft ActiveX Data Objects 6.1 Object Library.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 10.1 Setting Up a Reference to the ADO Object Library

**NOTE**

*Create a new folder on your computer named VBAAccess2021\_ByExample and designate it as a trusted folder (see Chapter 1 for details). We will use this folder to store database files created in Chapters 10–25.*

1. Start Microsoft Access 2021 and create a new database named **Chap10.accdb** in your C:\VBAAccess2021\_ByExample folder.
2. Press **Alt+F11** to switch to the Visual Basic Editor window, and choose **Tools | References**.
3. Scroll down the list of available references until you locate the **Microsoft ActiveX Data Objects 6.1 Library**. Click the checkbox to the left of the name to select it.
4. Click **OK** to close the References dialog box.

All libraries that are checked in the References dialog box can be browsed using the Object Browser. This is a good way to become familiar with the names of objects that are available in a specific library and their various properties and methods (see Figure 10.2).

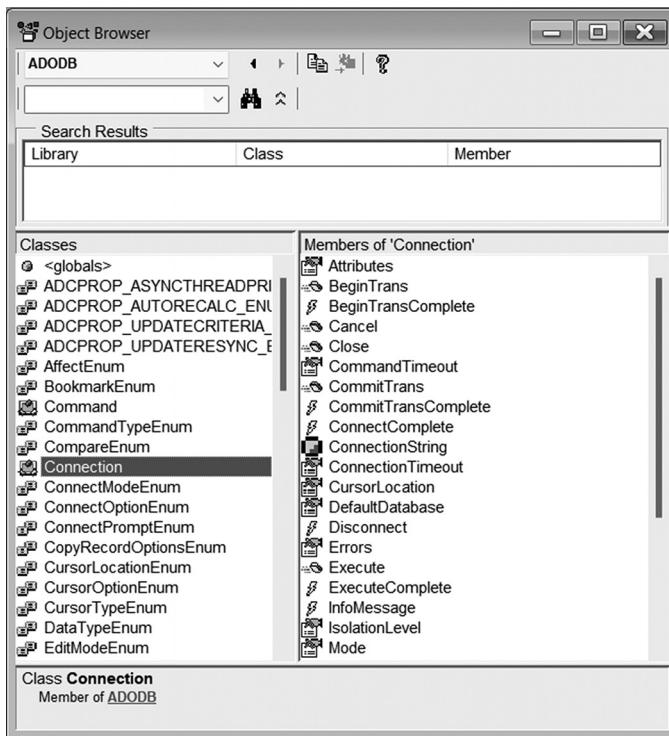


FIGURE 10.2. Use the Object Browser to find the objects available in a specific library.

## UNDERSTANDING CONNECTION STRINGS

---

Needless to say, to retrieve or write data to a database, you will need to open it. There are many ways to connect to a database or an external data source from Microsoft Access 2021. The first thing to know about establishing database connections from your VBA procedures is how to prepare and use connection strings.

A *connection string* is a string variable that tells your VBA application how to establish a connection to a data source. There are two types of connection strings:

- ODBC connection strings (used by ODBC drivers)
- OLE DB connection strings (used by the OLE DB Provider)

The syntax of ODBC and OLE DB connection strings is very similar. The connection string consists of a series of keyword and value pairs separated by semicolons:

```
Keyword1=value; Keyword2=value
```

*Please note that the connection string does not contain spaces before or after the equal sign (=).* The parameters in the connection string may vary depending on the ODBC driver or OLE DB provider used and the data store that you are connecting to such as Microsoft Access, SQL Server, and so forth.

Let's examine the connection string you would need to connect to an older Microsoft Access database in the .mdb file format. For the ODBC connection, the following connection string will allow you to connect to an Access database called Northwind.mdb:

```
"Driver={Microsoft Access Driver (*.mdb)};" & _  
"DBQ=C:\VBAAccess2021_ByExample\Northwind.mdb;"
```

In the preceding connection string, `Driver` specifies what type of database you're using. `DBQ` is the physical path to the database. If the Northwind.mdb file is protected with a password, you must provide additional information in the connection string:

```
"Driver={Microsoft Access Driver (*.mdb)};" & _  
"DBQ=C:\VBAAccess2021_ByExample\Northwind.mdb;" & _  
"UID=admin;PWD=secret;"
```

`UID` specifies the username. `PWD` specifies the user password.

To create an OLE DB connection to the same Northwind.mdb database that uses standard security, you will need to write the connection string as follows:

```
"Provider=Microsoft.Jet.OLEDB.4.0;" & _  
"Data Source=C:\VBAAccess2021_ByExample\Northwind.mdb;" & _  
"User Id=Admin;Password=;"
```

Or

```
"Provider=Microsoft.ACE.OLEDB.12.0;" & _  
"Data Source=C:\VBAAccess2021_ByExample\Northwind.mdb;" & _  
"User Id=Admin;Password=;"
```

`Provider` identifies the OLE DB provider for your database; in this case, we want to use the Jet OLE DB Provider or the ACE OLE DB Provider. You should use the Microsoft.ACE.OLEDB Provider if you are running Windows 64-bit and Office 64-bit.

Data Source specifies the full path and filename of the .mdb database file.

To create an OLE DB connection to the SQL database called AdventureWorks, use the following connection string:

```
"Provider=SQLOLEDB;Data Source=(local);" & _  
"Integrated Security=SSPI;Initial Catalog=AdventureWorks"
```

In this connection string, SQLOLEDB is the name of the OLE DB provider for SQL Server databases. The Data Source parameter specifies the name or address of the SQL Server. To connect with an SQL Server running on the same computer, use the keyword (local) for the Data Source. For a trusted connection (Microsoft Windows NT integrated security), set the Integrated Security parameter to SSPI. Use the Initial Catalog parameter to specify which database you want to connect to.

**NOTE**

*If the Provider keyword is not included in the connection string, the OLE DB provider for ODBC (MSDASQL) is the default value. This provides backward compatibility with ODBC connection strings.*

## USING ODBC CONNECTION STRINGS

---

When you choose to connect to a data source via the ODBC, you must specify the connection information. You do this by creating a DSN (Data Source Name) or DSN-less connection. DSN connections store the connection information in the Windows Registry or in a .dsn file. In a *DSN-less* connection, all connection information is specified in the connection string. The following subsections explain each ODBC connection type in detail.

### Creating and Using ODBC DSN Connections

---

Windows uses an ODBC Data Source Administrator (see Figure 10.3) to manage ODBC drivers and data sources available on the computer. You can access this tool by opening Control Panel | System and Security | Administrative Tools (in Windows 11, use Windows Tools) | ODBC Data Sources (32-bit) or ODBC Data Sources (64-bit).

The DSN contains information about database configuration, location, and user security. There are three types of DSNs:

**User DSN**—A User DSN is stored locally in the Windows Registry and limits database connectivity to the user who creates it. In other words, if you create a

User DSN under your user account, no other user will be able to see it or use it. Hands-On 10.2 demonstrates how to create this type of DSN so that you can run the example code on your computer.

- **File DSN**—A File DSN is a special type of file that stores all the connection settings. File DSNs are saved by default in the Program Files\Common Files\Odbc\Data Sources folder. Because the connection parameters and values are stored in a file, they can be easily shared with other users. If other users require the same connection, simply send them the DSN file and you won't need to configure a DSN for each system.
- **System DSN**—A System DSN is stored locally in the Windows Registry and allows any logged-on user, process, and service to see it and use it. System DSNs are often used in establishing connections to external data sources from Active Server Pages (ASP).

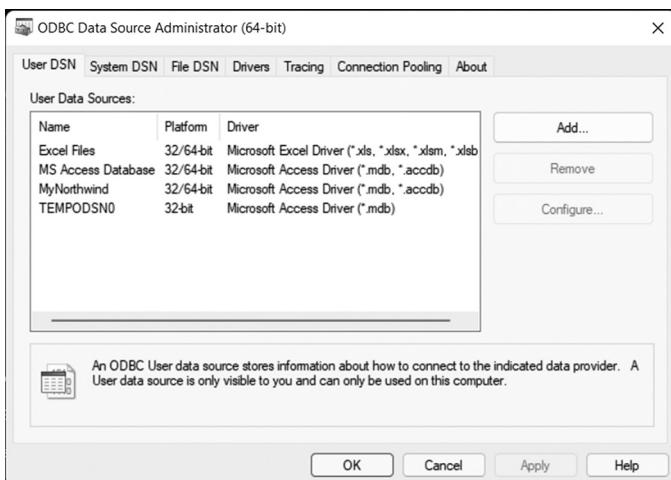


FIGURE 10.3. The ODBC Data Source Administrator allows you to set up appropriate connections with the required data provider via the User, System, or File DSN.

Hands-On 10.2 will get you started with the ODBC Data Source Administrator by walking you through the creation of a User DSN named MyDbaseFile to access data in a legacy dBASE database file (Customer.dbf). You will then use this data source name to programmatically open a dBASE file with ADO using the ODBC DSN connection. If you don't have the dBASE driver on your newer Windows machine, you will need to download and install *Microsoft Access*

*Database Engine 2010 Redistributable* that contains a set of components used in data transfers. See the following site for the download:

<https://www.microsoft.com/en-US/download/details.aspx?id=13255>



## Hands-On 10.2 Creating and Using the ODBC DSN Connection to Read Data from a dBASE File

The procedure code in this Hands-On relies on the reference to the ActiveX Data Objects Library that was set in Hands-On 10.1.

1. Copy the **Customer.dbf** file from the companion files to your **C:\VBAAccess2021\_ByExample** folder.
2. Open the Control Panel, activate **Administrative Tools**, and double-click **ODBC Data Sources (64-bit)**.  
The ODBC Data Source Administrator dialog box appears, as shown earlier in Figure 10.3.
3. With the **User DSN** tab selected, click the **Add** button.
4. Select **Microsoft Access dBASE driver (\*.dbf, \*.ndx, \*.mdx)** and click **Finish**.
5. Enter **MyDbaseFile** as the Data Source Name and choose **dBASE 5.0** for the database version, as shown in Figure 10.4. Make sure you clear the **Use Current Directory** checkbox, then click the **Select Directory** button.

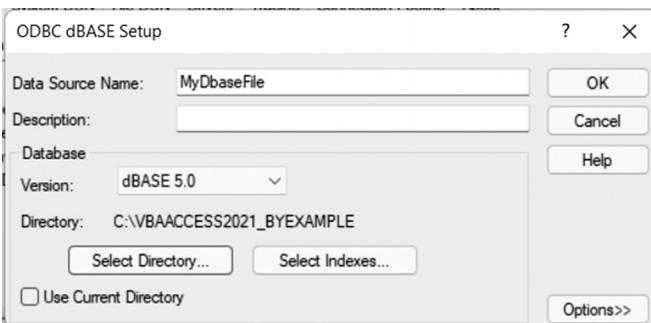


FIGURE 10.4. Creating a Data Source Name (DSN) to access a dBASE file.

6. In the Select Directory dialog box, select the **C:\VBAAccess2021\_ByExample** folder where the **Customer.dbf** file is located, and click the **OK** button.
7. Click **OK** to exit the ODBC dBASE Setup dialog box.  
The **MyDbaseFile** data source now appears in the list of User Data Sources in the ODBC Data Source Administrator dialog box.
8. Click **OK** to close the ODBC Data Source Administrator dialog box.

9. Activate the Visual Basic Editor window and choose **Insert | Module**.
10. In the module's Code window, enter the following **Open\_AndRead\_dBaseFile** procedure:

```
Sub Open_AndRead_dBaseFile()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    conn.Open "Provider=MSDASQL;DSN=MyDbaseFile;"

    Debug.Print conn.ConnectionString

    Set rst = New ADODB.Recordset
    rst.Open "Customer.dbf", conn

    Do Until rst.EOF
        Debug.Print rst.Fields(1).Value
        rst.MoveNext
    Loop

    rst.Close
    Set rst = Nothing
    conn.Close
    Set conn = Nothing
End Sub
```

11. Choose **Run | Run Sub/UserForm** to execute the procedure.
12. Press **Ctrl+G** to open the Immediate window to view the data returned by the procedure.

<b>NOTE</b>	<p><i>If Visual Basic displays the runtime error “Data source name not found and no default driver specified,” make sure there are no extra spaces in the connection string:</i></p> <p style="padding-left: 20px;"><code>conn.Open "Provider=MSDASQL;DSN=MyDbaseFile;"</code></p> <p><i>This is a very common error and it’s hard to trace because spaces are difficult to spot.</i></p>
-------------	---

The `Open_AndRead_dBaseFile` procedure uses the ADO Connection object to establish a connection with the data source. Prior to using ADO objects in your VBA procedures, make sure that the References dialog box contains the refer-

ence to the ActiveX Data Objects Library (see Hands-On 10.1). The procedure begins by declaring an object variable of Connection type, like this:

```
Dim conn As ADODB.Connection
```

**NOTE**

*The Connection object variable can be declared at procedure level or at module level. By declaring the variable at the top of the module, you can reuse it in multiple procedures in your module.*

To handle data retrieval, an object variable of Recordset type is also declared:

```
Dim rst As ADODB.Recordset
```

Before you can use the declared ADO Connection object, you must initialize the object variable by using the Set keyword:

```
Set conn = New ADODB.Connection
```

At this point you can proceed to opening the data source by using the ADO Connection object's Open method. The required database connection information is passed to the Open method in the connection string, like this:

```
conn.Open "Provider=MSDASQL;DSN=MyDbaseFile;"
```

MSDASQL is the Microsoft OLE DB Provider for all ODBC data sources. The names of common data providers used with ADO are listed in Table 10.4. The Provider property of the ADO Connection object is used in the connection string as the provider name. DSN is the name of the data source that you specified for your connection settings in the ODBC Data Source Administrator dialog box. Since MSDASQL is the default provider for ODBC, it's also permitted to leave it off, like this:

```
conn.Open «DSN=MyDbaseFile;»
```

TABLE 10.4. Common data providers used with ADO

Provider Name	Provider Property	Description
Microsoft ACE	Microsoft.ACE.OLEDB.12.0	Used by Access 2021-2010 databases in .accdb file format. By default, this provider opens databases in Read/Write mode.
Microsoft Jet	Microsoft.Jet.OLEDB.4.0	Used for Jet 4.0 databases (in .mdb file format). By default, this provider opens databases in Read/Write mode.
Microsoft SQL Server	SQLOLEDB	Used to access SQL Server databases.

Provider Name	Provider Property	Description
Oracle	MSDAORA	Used to access Oracle databases.
ODBC	MSDASQL	Used to access ODBC data sources without a specific OLE DB provider. This is the default provider for ADO.
Active Directory Service	ADSDSOObject	Used to access Windows NT 4.0 directory services, Novell® directory services, and LDAP-compliant directory services.
Index Server	MSIDXS	Read-only access to Web data.

Once the connection to the dBASE database file is open, the procedure initializes the `rst` object variable using the `Set` keyword in order to gain access to its data:

```
Set rst = New ADODB.Recordset
```

The ADO Recordset object's `Open` method is used to open the `Customer.dbf` file, like this:

```
rst.Open "Customer.dbf", conn
```

When you open the recordset, you need to specify at the minimum the data you want to retrieve (`Customer.dbf`) and how to connect to that data (`conn`). Once the recordset is open, you can start reading its data. The `Do Until` loop will iterate through the recordset until the `EOF` (End of File) is reached. Each time through the loop, VBA will write to the Immediate window the value of the first field. When the procedure ends you should see in the Immediate window the names of all customers from the `Customer.dbf` file.

When you are done reading the records, the procedure uses the `Close` method to close the recordset and destroy the `rst` object variable by setting it to `Nothing`:

```
rst.Close
Set rst = Nothing
```

This statement completely releases the resources used by the Recordset object. The same should be done with the Connection object variable (`conn`) when it is no longer needed:

```
conn.Close
Set conn = Nothing
```

## Creating and Using DSN-Less ODBC Connections

It is possible that your VBA application that relies on database access via ODBC DSN (Data Source Name) may suddenly fail because the DSN was modified or deleted. Therefore, it may be a better idea to use another type of connection known as a DSN-less connection. Instead of setting up a DSN as you did in Hands-On 10.2, specify your ODBC driver name and all driver-specific information in your connection string. Different types of databases can require that you specify different parameters. Because the ODBC DSN setup using the ODBC Administrator dialog is not required, this type of connection is called “DSN-less.”

### Additional Code on CD-ROM

You can rewrite the procedure in Hands-On 10.2 to use a DSN-less ODBC connection. See the HandsOn10.2\_Supplement.txt on the companion files.

**TABLE 10.5.** ODBC connection strings for common data sources

Data Source Driver	ODBC Connection String (used in DSN-less connections)
Microsoft Access 2021–2007 (accessing .mdb or .accdb files in Access versions 2007 through 2021)	“Driver={Microsoft Access Driver (*.mdb, *.accdb)}; DBQ=path to mdb/accdb file;UID=admin;PWD=;”
Microsoft Access 2003–97 (accessing .mdb files from Access 2003/2002/2000/97)	<p><i>Using standard security:</i> “Driver={Microsoft Access Driver (*.mdb)}; DBQ=C:\filepath\myDb.mdb;UID=admin;PWD=;”</p> <p><i>Using user-level security (workgroup information file):</i> “Driver={Microsoft Access Driver (*.mdb)}; DBQ=C:\filepath\myDb.mdb;SystemDB=C:\filepath\myDb.mdw; UID=myUserName;PWD=myPassword;”</p>
Microsoft Excel 2021–2007 (accessing .xls, .xlsx, .xlsm, and .xlsb files from Excel 2021–2007)	“Driver={Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)};DBQ=path to xls/xlsx/xlsm/xlsb file;”
Microsoft Excel (accessing .xls files from Excel 2003–97)	“Driver={Microsoft Excel Driver (*.xls)}; DBQ=C:\filepath\Spreadsheet.xls;”
dBASE	<p>32-bit driver “Driver={Microsoft dBASE Driver (*.dbf)}; DBQ=C:\filepath;”</p> <p>64-bit driver (from Microsoft Access Database Engine 2010 Redistributable)</p>

Data Source Driver	ODBC Connection String (used in DSN-less connections)
	“Driver={Microsoft Access dBASE Driver (*.dbf, *.ndx, *.mdx)}; DBQ=C:\filepath;”
Text	“Driver={Microsoft Text Driver (*.txt, *.csv)}; DefaultDir=C:\filepath\myText.txt;”
Microsoft SQL Server	<p><i>Using Trusted Connection security:</i>            “Driver={SQL Server};            Server=myServerName;            Database=myDatabaseName;            UID=;PWD=;”</p> <p><i>Using standard security:</i>            “Driver={SQL Server};            Server=myServerName;            Trusted_Connection=no;            Database=myDatabaseName;            UID=myUserName;PWD=myPassword;”</p>
Oracle	“Driver={Microsoft ODBC for Oracle}; Server=OracleServer.World; UID=myUserName;PWD=myPassword;”

## USING OLE DB CONNECTION STRINGS

---

In numerous VBA procedures in this book, we'll use an OLE DB provider to communicate with a data source. See Table 10.4 earlier in this chapter for the names of common OLE DB providers used with ADO. Table 10.6 shows OLE DB connection strings for common data sources.

TABLE 10.6 OLE DB connection strings for common data sources

Data Source	OLE DB Connection String
Microsoft Access 2021–2007	“Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\VBAAccess2021_ByExample\Northwind 2007.accdb”
Microsoft Access (prior to 2007)	“Provider=Microsoft.Jet.OLEDB.4.0; Data Source=C:\VBAAccess2021_ByExample\Northwind. mdb;”

Data Source	OLE DB Connection String
Microsoft Excel 2021–2010	“Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\VBAAccess2021_ByExample\Report2021.xlsx; Extended Properties=’”Excel 12.0;HDR=Yes”’;”
Microsoft Excel 2007	“Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\VBAAccess2021_ByExample\Report2021.xlsx; Extended Properties=’”Excel 12.0;HDR=Yes”’;”
Microsoft Excel (prior to 2007)	“Provider=Microsoft.Jet.OLEDB.4.0; Data Source=C:\VBAAccess2021_ByExample\Report.xls; Extended Properties=’”Excel 8.0;HDR=Yes”’;”
Microsoft SQL Server	“Provider=SQLOLEDB;Data Source=myServerName;Network Library=DBMSSOCN;Initial Catalog=Pubs;”
Oracle	“Provider=MSDAORA;Data Source=myTable;”

## CONNECTION STRING VIA A DATA LINK FILE

---

If you are using the Windows operating system and are looking for an easy way to create and test a connection string that uses an ODBC driver or OLE DB provider, you may want to use the Data Link Properties dialog box, which is shown in Figure 10.5.

A universal data link file (.udl) is a text file containing the connection information. Hands-On 10.3 demonstrates how to create the .udl file to connect to a Microsoft Access 2021 database. You can use the same technique to create a valid connection string to other external data sources as long as the ADO provider is installed on your computer.



### Hands-On 10.3 Creating and Using a Universal Data Link File

1. In Windows File Explorer, select the C:\VBAAccess2021\_ByExample folder. Make sure that the option to hide extensions for known file types is deselected in the View tab of the Folder Options. Next, Create a new Text Document in this folder.
2. A new file named New Text Document.txt appears in the VBAAccess2021\_ByExample folder. Rename this file **ConnectToAccdb.udl**.

When changing the filename, be sure to type the new extension (.udl) as indicated.

Windows will display a warning message that changing the file extension can cause the file to become unusable. Ignore this message and click **OK**.

Windows creates an empty universal data link file. Notice that the file size is 0 Kb.

- Double-click the **ConnectToAccdb.udl** file to open the Data Link Properties dialog. Windows opens the Data Link Properties dialog box (Figure 10.5), which contains the following four tabs:

Data Link Tab	Description
Provider	Lists the names of the ADO providers installed on your computer. The provider name you select must be appropriate for the data source you want to use. For example, if you select Microsoft Jet 4.0 OLE DB provider, you must select an Access database in .mdb format.
Connection	Allows you to define a data source name for the selected provider type. The entries shown here are specific to the provider type selected via the Provider tab. The Connection tab is active by default when you activate the Data Link Properties dialog box.
Advanced	Allows you to view and set other initialization properties for your data connection.
All	Allows you to review and edit all OLE DB initialization properties available for the selected OLE DB provider.

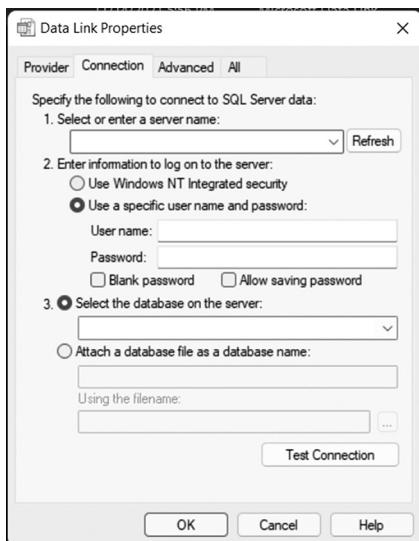


FIGURE 10.5 The Data Link Properties dialog box appears after you launch the .udl file.

- Click the **Provider** tab and select **Microsoft Office 12.0 Access Database Engine OLE DB Provider**, as shown in Figure 10.6.

**NOTE**

If you don't see the above-mentioned data provider, you will need to download and install the Microsoft Access Database Engine Redistributable from Microsoft:  
<https://www.microsoft.com/en-us/download/details.aspx?ID=13255>  
Follow Microsoft instructions on the download page. If you run into installation issues, be sure to follow the workaround:  
<https://support.microsoft.com/en-us/help/2874601/can-not-use-access-odbc-or-oledb-provider-outside-office-c2r-apps>

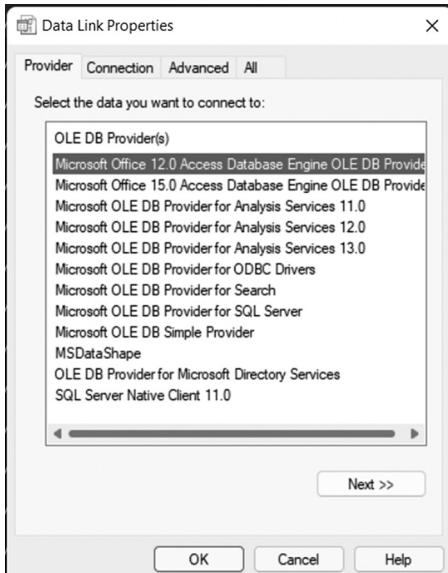
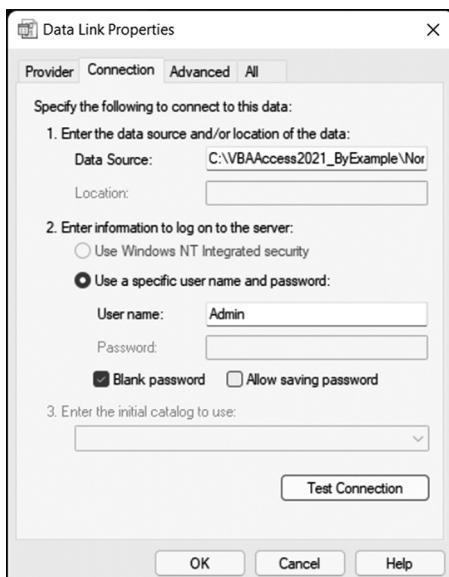


FIGURE 10.6. The Provider tab in the Data Link Properties dialog box lists the names of the ADO providers installed on your computer.

- Click the **Next** button or activate the **Connection** tab.  
The entries shown on the Connection tab are related to the type of provider you selected in step 4.
- In the Data Source box, type the location and filename of the Access database you want to connect to: C:\VBAAccess2021\_ByExample\Northwind 2007.accdb (see Figure 10.7).

7. Click the **Test Connection** button to test whether you can connect to the specified database using the chosen data provider.
8. Click **OK** to the message box “Test connection succeeded.”  
If you misspelled a filename or Windows cannot locate the file in the specified folder, you will get an error.



**FIGURE 10.7.** Use the Data Link Properties dialog box to define a data source name for the selected provider type. Be sure to enter .accdb as the extension for the Northwind 2007 database (the Data Source text box is too short to capture the entire path in this image).

- At this point your connection string is ready to use.
9. Click **OK** to close the Data Link Properties dialog box.

When writing a VBA procedure to connect to the Northwind 2007.accdb database, you can simply pass the .udl filename to the Connection object's `Open` method:

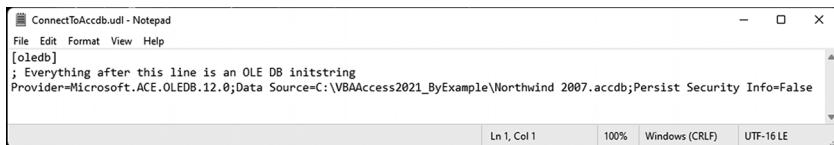
```
Dim conn As ADODB.Connection  
Set conn As New ADODB.Connection  
conn.Open "File Name=C:\VBAAccess2021_ByExample\ConnectToAccdb.udl;"
```

When you use .udl files to store connection information, it is very easy to switch your procedure's data source without having to make changes to your code. Simply double-click the .udl file and make desired modifications in the Data Link Properties dialog box.

If you'd rather use the connection string in your VBA procedure, then go ahead and copy the string from the .udl file. You can open this file in Notepad in one of the following ways:

- Right-click the .udl filename and choose Open With, then select Notepad. If Notepad is not available in the shortcut menu, select Choose Program, then choose Notepad, and click OK.
- Make a copy of the .udl file. Change the .udl extension of the created copy to .txt. Double-click the file to open it in Notepad.

The connection string is shown in Figure 10.8.



```
ConnectToAccdb.udl - Notepad
File Edit Format View Help
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\VBAAccess2021_ByExample\Northwind 2007.accdb;Persist Security Info=False
```

FIGURE 10.8. You can obtain the connection string from the universal data link (.udl) file by opening the file in Windows Notepad.

## SUMMARY

---

In this chapter, you were introduced to Access database engines (Jet and ACE) as well as several object libraries that provide objects, properties, and methods for your VBA procedures. You learned how to set up and troubleshoot library references in your VBA modules. Finally, you learned about various types of connection strings and used the ODBC Data Source Administrator and Universal Data Link (UDL) Properties dialog to create your connections.

In the next chapter, you will learn how to create and manipulate databases using objects from the DAO Object Library.

You can create and manipulate Access databases programmatically by using Data Access Objects (DAO) or ActiveX Data Objects (ADO). This chapter walks you through performing various database tasks using DAO. To work with the ADO objects, refer to Chapter 12.

## **UNDERSTANDING THE DBENGINE AND WORKSPACE OBJECTS**

---

In Access, a *workspace* is a named user session that contains all open databases and provides a mechanism for simultaneous transactions and security. You can have more than one workspace open at a time. All the workspaces you create are stored in the `Workspaces` collection. You can use the `Workspace` object to manage the current session or to start an additional session.

When you start Access, the program automatically creates a default workspace named #Default Workspace#. This workspace can be referenced using any of the following ways:

```
DBEngine.Workspaces(0)  
DBEngine(0)  
Workspaces(0)
```

To quickly find out the name of the current workspace, open any Access database and activate the Visual Basic Editor screen. Press Ctrl+G to open the Immediate window, and enter the following:

```
?Workspaces(0).Name
```

When you press Enter, you should see #Default Workspace# on the next line. Try the other two methods using the Name property and you will get the same response:

```
?DBEngine.Workspaces(0).Name  
#Default Workspace#  
?DBEngine(0).Name  
#Default Workspace#
```

You can create a new workspace using the CreateWorkspace method of the DBEngine object.

The DBEngine object is the top-level object in the DAO object model and is a property of the Access Application object. It contains all other objects in the DAO object hierarchy and has two collections: Workspaces and Errors. The following procedure creates a new workspace and adds it to the Workspaces collection. The For Each loop is then used to iterate through the Workspaces collection and print each workspace name to the Immediate window. The procedure code can be found in the Workspace.mdb database in the companion files.

```
Sub Create_NewWorkspace()  
    Dim wsSecond As DAO.Workspace  
    Dim userN As String  
    Dim userPwd As String  
    Dim ws As Variant  
  
    userN = "Admin"  
    userPwd = ""  
    ' create a new Access workspace  
    Set wsSecond = DBEngine.CreateWorkspace("myNewWS", _  
        userN, userPwd, dbUseJet)
```

```
'Add the new workspace to the Workspaces collection
Workspaces.Append wsSecond
Debug.Print "Total Workspaces: " & Workspaces.Count
For Each ws In Workspaces
    Debug.Print ws.Name
Next

' close and release object variable
wsSecond.Close
Set wsSecond = Nothing
End Sub
```

## THE DAO ERRORS COLLECTION

---

In Chapter 9, you learned about the VBA `Err` object that can be used to find out the information about the encountered error. The `Err` object stores information about the last VBA error. In DAO, you use the DAO `Errors` collection to find information about the last DAO error. DAO objects can generate one or more errors. All these errors are stored in the `DBEngine.Errors` collection. By enumerating the `Errors` collection in your error handling code, you can determine the types of problems to watch for and take the most appropriate action. You should retrieve the information about the error as soon as it occurs. The reason for it is that the DAO Errors collection is automatically cleared as soon as another error occurs. In other words, the new error replaces the previous one. Keep in mind that the first error in the `Errors` collection is indexed at 0 (zero). It is recommended that you always verify that the last error in the VBA `Err` object is the same as the last error in the `DBEngine.Errors` collection.

The following procedure is a modified version of the previous procedure with an additional statement that triggers the error. The statements that follow retrieve the error information from the `Errors` collection and the `Err` object. Notice that the `On Error Resume Next` statement ensures that we can continue with the remaining code without interruption despite the error. The error will be added to the `Errors` collection so that we can retrieve it. If you comment out this statement, Access displays an error message as soon as the error occurs.

```
Sub Create_NewWorkspace2()
    Dim wsSecond As DAO.Workspace
    Dim userN As String
    Dim userPwd As String
```

```
Dim ws As Variant
Dim itm As Variant

userN = "Admin"
userPwd = ""

' create a new Access workspace
Set wsSecond = DBEngine.CreateWorkspace("myNewWS", _
    userN, userPwd, dbUseJet)

On Error Resume Next
'Add the new workspace to the Workspaces collection
Workspaces.Append wsSecond

' add the same workspace again to generate the error
Workspaces.Append wsSecond

' get information about the error
For Each itm In DBEngine.Errors
    Debug.Print "Error Number: " & itm.Number
    Debug.Print "Error Description: " & itm.Description
    Debug.Print "Error Source: " & itm.Source
Next
Debug.Print "VBA error number in Err object: " & Err.Number

Debug.Print "Total Workspaces: " & Workspaces.Count
For Each ws In Workspaces
    Debug.Print ws.Name
Next

' close and release object variable
wsSecond.Close
Set wsSecond = Nothing
End Sub
```

When you run the above procedure, you should see the following output in the Immediate window:

```
Error Number: 3367
Error Description: Cannot append. An object with that name
already exists in the collection.
Error Source: DAO.Workspaces
VBA error number in Err object: 3367
Total Workspaces: 2
#Default Workspace#
myNewWS
```

## CREATING A DATABASE WITH DAO

The `Workspace` object has several useful methods, and the most frequently used are `CreateDatabase` (for creating a new database) and `OpenDatabase` (for opening an existing database). The `CreateDatabase` method requires that you specify the name and path of your database, as well as the built-in constant indicating a collating order for creating the database. Use the built-in constant `dbLangGeneral` for English, German, French, Portuguese, Italian, and Modern Spanish.

The procedure in Hands-On 11.01 creates a new Access 2021 database and displays the number of system tables that Access automatically creates for its own use.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 11.1 Creating a Database Using DAO

1. Start Access and create a new database named `Chap11.accdb` in your `C:\VBAAccess2021_ByExample` folder.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following `CreateNewDB.DAO` procedure:

```
Sub CreateNewDB.DAO()
    Dim db As DAO.Database
    Dim dbName As String

    dbName = "C:\VBAAccess2021_ByExample\TestDAO.accdb"

    On Error GoTo ErrorHandler

    Set db = CreateDatabase(dbName, dbLangGeneral)

    MsgBox "The database contains " & _
        db.TableDefs.Count & " tables."
    db.Close
    Set db = Nothing
    Exit Sub

ErrorHandler:
    MsgBox Err.Description
End Sub
```

4. Choose **Run | Run Sub/UserForm** to execute the procedure.

To create a `Database` object in code, first declare an object variable of type `Database`. Once the `Database` object variable is defined, set the variable to the object returned by the `CreateDatabase` method:

```
Set db = CreateDatabase(dbName, dbLangGeneral)
```

The `CreateDatabase` method creates a new `Database` object and appends it to the `Databases` collection. The new database contains several system tables that Access creates for its own use. If the database already exists, an error occurs. You can check for the existence of the database by using an `If` statement in combination with the VBA `Dir` function, as shown in Hands-On 11.02, and then use the VBA `Kill` statement to delete the database.

## COPYING A DATABASE

---

At times you may want to duplicate your database programmatically. This can be easily done in DAO with the `DBEngine` object's `CompactDatabase` method.

Before using the `CompactDatabase` method, make sure the source database is closed and there is enough disk space to create a duplicate copy. Creating a copy of your database in code requires that you define two string variables: one to hold the name of the source database and the other to specify the name for the duplicate version.

Hands-On 11.02 shows how to use the `CompactDatabase` method to copy a database.



### Hands-On 11.2 Copying a Database

This hands-on exercise makes a copy of the `TestDAO.accdb` database created in Hands-On 11.01.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `CopyDB.DAO` procedure:

```
Sub CopyDB.DAO()
    Dim dbName As String
    Dim dbNewName As String

    dbName = InputBox("Specify a database you " & _
        "want to copy: ", "Create a copy of", _
        "C:\VBAAccess2021_ByExample\TestDAO.accdb")
```

```
If dbName = "" Then Exit Sub

If Dir(dbName) = "" Then
    MsgBox dbName & " was not found. " & Chr(13) _
        & "Check the database name or path."
    Exit Sub
End If

dbName = InputBox("Duplicate database name: ", _
    "Save As", _
    "C:\VBAAccess2021_ByExample\Copy_TestDAO.accdb")

If dbNewName = "" Then Exit Sub

If Dir(dbNewName) <> "" Then
    Kill dbNewName
End If

DBEngine.CompactDatabase dbName, dbNewName
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.
4. You will be prompted to specify the name of the database you want to copy and the name for the copy.

This procedure uses the VBA `Dir` function to check for the existence of the database with the specified name:

```
If Dir(dbNewName) <> "" Then
    Kill dbNewName
End If
```

Because the database cannot be deleted programmatically using DAO, the VBA `Kill` statement is used to perform the deletion. The last statement in the `CopyDB.DAO` procedure uses the `CompactDatabase` method of the `DBEngine` object to create a copy of a database using the user-supplied arguments: a source database name (`dbName`) and a destination database name (`dbNewName`).

## OPENING MICROSOFT ACCESS DATABASES

---

In this section, you will learn how to use DAO to open existing Access databases in .ACCDB and .MDB format. These databases may be opened in read/write mode or in read-only mode, and some of them may be protected with database passwords or user-level security.

## Opening a Microsoft Jet Database in Read/Write Mode

The easiest way to open an existing Access database from a VBA procedure is by using the Microsoft Access database engine's `OpenDatabase` method. This method requires that you provide at least one parameter—the name of the existing database. When you open the database with the `OpenDatabase` method, always remember to close it using the `Close` method.

Hands-On 11.3 demonstrates how to open an Access database in .accdb or .mdb format using the DAO's `OpenDatabase` method. This example will list containers and documents in the open database. Each `Database` object has a `Containers` collection that consists of built-in Container objects. The `Containers` collection is used for storing Microsoft Access's own objects. The Jet engine creates the following Container objects: `Databases`, `Tables`, and `Relations`. Other Container objects are created by Access (`Forms`, `Reports`, `Macros`, and `Modules`).

Table 11.1 lists the Container objects and the type of information they contain.

TABLE 11.1. Container objects

Container Name	Type of Information Stored
Databases	Saved databases
Tables	Saved tables and queries
Relations	Saved relationships
Forms	Saved forms
Modules	Saved modules
Reports	Saved reports
Scripts	Saved scripts

Each Container object contains a `Documents` collection. Each document in this collection represents an object that can be found in an Access database. For example, the `Forms` container stores a list of all saved forms in a database, and each form is represented by a `Document` object. You cannot create new Container and Document objects; you can only retrieve the information about them.



### Hands-On 11.3 Opening a Database in Read/Write Mode

1. In the `Chap11.accdb` database that you created in Hands-On 11.1, switch to the Visual Basic Editor window and choose **Insert | Module** to add a new module to the current VBA project.

2. In the module's Code window, type the following **openDB.DAO** procedure:

```
Sub openDB.DAO()
    Dim db As DAO.Database
    Dim dbName As String
    Dim c As Container
    Dim doc As Document

    dbName = InputBox("Enter a name of an existing database:", _
                      "Database Name")

    If dbName = "" Then Exit Sub
    If Dir(dbName) = "" Then
        MsgBox dbName & " was not found."
        Exit Sub
    End If

    Set db = OpenDatabase(dbName)
    With db
        ' list the names of the Container objects
        For Each c In .Containers
            Debug.Print c.Name & " container:" & _
                        c.Documents.Count
            ' list the document names
            ' in the specified Container
            If c.Documents.Count > 0 Then
                For Each doc In c.Documents
                    Debug.Print vbTab & doc.Name
                Next doc
            End If
        Next c
        .Close
    End With
End Sub
```

This procedure uses the `OpenDatabase` method of the `DBEngine` object to open the specified database in the default workspace. The database is opened as shared with read/write access. By supplying additional arguments to the `OpenDatabase` method you could open the database exclusively (a database opened exclusively can be accessed by a single user at a time) or as read-only.

The `openDB.DAO` procedure uses a `For Each...Next` loop to retrieve the names of all the `Container` objects in the opened database. If the specified container is not empty, the inner `For Each...Next` loop will print the name of each `Document` object in the Immediate window.

3. Position the insertion point anywhere within the code of openDB\_DAO and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.  
When you run this procedure, you are prompted to enter the name of the Access database.
4. Enter **C:\VBAAccess2021\_ByExample\Northwind 2007.accdb** or **C:\VBAAccess2021\_ByExample\Northwind.mdb** and press **OK**. Check the procedure output in the Immediate window.

### **Opening a Microsoft Access Database in Read-Only Mode**

---

You can open an Access database in read-only mode by providing settings for optional arguments in the `OpenDatabase` method.

Additional code is available in the companion files.

File Name: `openDB_DAOReadOnly.txt`

Description: Open a database for shared, read-only access using DAO

### **Opening a Microsoft Jet Database Secured with a Password**

---

Using passwords to secure the database or objects in the database is known as *share-level security*. When you set a password on the database, users will be required to enter a password to gain access to the data and database objects. Keep in mind that passwords are case-sensitive. When using DAO to change the password of an existing Access database in a VBA procedure, follow these steps:

1. Open the database in exclusive mode by setting the second argument of the `OpenDatabase` method to `True`.
2. To set a database password, use the `NewPassword` property of the `Database` object. This property requires that you first specify the old password and then the new one. Passwords can be up to 20 characters long and can include any characters except the ASCII character 0 (Null). To specify that the database does not have a password, use a zero-length string ("") in the first parameter of the `NewPassword` property. To clear the password, use the zero-length string for the second parameter of the `New Password` property.

To open a password-protected database using DAO, you must specify the database password in the `Connect` parameter of the `OpenDatabase` method, as shown in Hands-On 11.4.



#### Hands-On 11.4 Setting a Database Password and Opening a Password-Protected Database

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new module to the currently open **Chap11.accdb** database.
2. In the module's Code window, type the **setPass\_AndOpenDB\_withDAO** procedure shown here:

```
Sub setPass_AndOpenDB_withDAO()
    Dim db As DAO.Database
    Dim strDb As String

    ' strDb = "C:\VBAAccess2021_ByExample\Northwind 2007.accdb"
    strDb = "C:\VBAAccess2021_ByExample\Northwind.mdb"

    ' open the database in exclusive mode
    ' to set database password
    Set db = DBEngine.OpenDatabase(strDb, True)
    db.NewPassword "", "secret"
    MsgBox "Access Database version: " & Int(db.Version)
    db.Close

    ' open password-protected database
    Set db = DBEngine.OpenDatabase(Name:=strDb, _
        Options:=False, _
        ReadOnly:=False, _
        Connect:=";PWD=secret")

    MsgBox "Successfully opened a password-protected database."
    db.Close
    MsgBox "Password-protected database was closed."

    ' remove password protection from the database
    Set db = DBEngine.OpenDatabase(Name:=strDb, _
        Options:=True, _
        ReadOnly:=False, _
        Connect:=";PWD=secret")
    db.NewPassword "secret", ""

    MsgBox "Password protection was removed."
    db.Close
End Sub
```

3. Position the insertion point anywhere within the code of the **setPass\_AndOpenDB\_withDAO** procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.

When you run this procedure, Access displays the version number of the Microsoft Jet or Microsoft Access database engine using the Version property of DBEngine. The version number consists of the version number, a period, and the release number. The procedure uses the VBA `Int` function to display only the integer portion of the number. Access 2021 uses the Microsoft Access database engine 14.0. Version 12 is returned for Access 2007-2019. Databases created in versions 2000–2003 use Microsoft Jet 4.0. Microsoft Access 97 uses Microsoft Jet 3.5.

**NOTE**

*If a VBA procedure uses a method or a property that requires two or more parameters, you can make the procedure more readable by specifying the names of the parameters like this: Constants for complex data types and the dbAttachment data type do not apply to versions prior to Access 2007.*

```
Set db = DBEngine.OpenDatabase(Name:=strDb, _
    Options:=False, _
    ReadOnly:=False, _
    Connect:=";PWD=secret")
```

Use the Microsoft Visual Basic help to find the names of methods and properties and the names of the required and optional parameters.

Hands-On 11.5 demonstrates how to use DAO to open an Excel workbook.



### Hands-On 11.5 Opening an Excel Workbook

1. Copy the **Report2021.xlsx** and **Report.xls** workbook files from the companion files to your C:\VBAAccess2021\_ByExample folder.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following **Open\_Excel\_DAO** procedure:

```
Sub Open_Excel_DAO(strFileName)
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strHeader As String
    Dim strValues As String
    Dim fld As Variant

    strHeader = ""
    strValues = ""

    If Right(strFileName, 1) = "x" Then
```

```
Set db = OpenDatabase(CurrentProject.Path & _
    "\Report2021.xlsx", False, True, _
    "Excel 12.0; HDR=YES;")
Else
    Set db = OpenDatabase(CurrentProject.Path & _
        "\Report.xls", False, True, _
        "Excel 8.0; HDR=YES;")
End If

Set rst = db.OpenRecordset("Sheet1$")

' get column names
For Each fld In rst.Fields
    strHeader = strHeader & fld.Name & vbTab
Next

Debug.Print strHeader

' get cell values
Do Until rst.EOF
    For Each fld In rst.Fields
        strValues = strValues & fld.Value & _
            vbTab & vbTab
    Next
    Debug.Print strValues
    strValues = ""
    rst.MoveNext
Loop

rst.Close
Set rst = Nothing
db.Close
Set db = Nothing
End Sub
```

4. In the Visual Basic Editor window, press **Ctrl+G** to open the Immediate window or choose **View | Immediate Window**.
5. To run the `Open_Excel.DAO` procedure, type `Open_Excel.DAO "Report.xls"` in the Immediate window and press **Enter**.
6. Run the procedure again, supplying `Report2021.xlsx` as the parameter.

To run the `Open_Excel.DAO` procedure, you must provide the name of the workbook file to open. If the last character in the file extension is “x” (this is determined with the VBA `Right` function), then the procedure uses the

connection string designed for opening Excel 2007–2021 files. After making a connection to the Excel file, the procedure goes on to retrieve information stored in the desired worksheet. Using the DAO's `OpenRecordset` method, we can access the data on the Sheet1 worksheet. Notice a dollar sign (\$) appended to the sheet name. You must use the dollar sign syntax, Sheet1\$, to refer to a sheet. The procedure uses the `For Each...Next` loop to obtain the names of all worksheet columns. The heading string is then written to the Immediate window. Next, the `Do Until...Loop` block loops through the records until the end of file (`EOF`) is reached. Cell values from each worksheet row are written to the `strValues` variable and then to the Immediate window. Once the data retrieval is completed, the `Recordset` is closed and its variable is destroyed. The same is done with the `Connection` object.

## **CREATING AND ACCESSING DATABASE TABLES AND FIELDS**

---

Now that you know how to create an Access database programmatically and connect to it using multiple methods, it's time to fill it with some useful objects. The first object you will create is a table. Typical operations you may want to perform on database tables and fields include the following:

- Setting field properties
- Making a copy of a table
- Deleting a table
- Listing table properties
- Adding new fields to an existing table
- Changing field properties
- Deleting a field from a table
- Linking a table to a database
- Listing tables in a database
- Changing the AutoNumber
- Listing data types

In the following sections, you will write VBA procedures that use DAO objects to perform these database tasks.

## **CREATING A MICROSOFT ACCESS TABLE AND SETTING FIELD PROPERTIES**

Each saved table in an Access database is an object called a TableDef object. The TableDef object has a number of properties that characterize it, such as Name, RecordCount, DateCreated, and DateUpdated. The TableDef object also has methods that act on the object. For example, the `CreateField` method creates a new field for the TableDef object and the `OpenRecordset` method creates an object called Recordset that is used to manipulate the data in the table.

The procedure in Hands-On 11.6 illustrates how to create a table in the current database using DAO.



### **Hands-On 11.6 Creating a Table**

1. Insert a new module in Chap11.accdb and in the module's Code window, type the following `CreateTableDAO` procedure:

```
Sub CreateTableDAO()
    Dim db As DAO.Database
    Dim tblNew As DAO.TableDef
    Dim fld As DAO.Field
    Dim prp As DAO.Property

    On Error GoTo ErrorHandler
    Set db = CurrentDB
    Set tblNew = db.CreateTableDef("Agents")

    Set fld = tblNew.CreateField("AgentID", dbText, 6)
    fld.ValidationRule = "Like 'A*'"
    fld.ValidationText = "Agent ID must begin with the " & _
        "letter 'A' and cannot contain more than 6 characters."
    tblNew.Fields.Append fld

    Set fld = tblNew.CreateField("Country", dbText)
    fld.DefaultValue = "USA"
    tblNew.Fields.Append fld

    Set fld = tblNew.CreateField("DateOfBirth", dbDate)
    fld.Required = True
    tblNew.Fields.Append fld
    db.TableDefs.Append tblNew

    ' Create Caption property and set its value
```

```

' add it to the collection of field properties
Set prp = tblNew.Fields("DateOfBirth"). _
    CreateProperty("Caption")
prp.Type = dbText
prp.Value = "Date of Birth"
fld.Properties.Append prp
MsgBox fld.Properties("Caption").Value

Set prp = tblNew.CreateProperty("Description")
prp.Type = dbText
prp.Value = "Sample table created with DAO code"
tblNew.Properties.Append prp

ExitHere:
Set fld = Nothing
Set tblNew = Nothing
Set db = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub

```

2. Choose **Run | Run Sub/UserForm** to execute the CreateTableDAO procedure.
3. Choose **File | Save** and click **OK** to save the Module when prompted. This will ensure that Access refreshes the application window and makes the newly created table visible in the navigation bar.

The CreateTableDAO procedure uses the `CurrentDb` method to define an object variable (`db`) to point to the database that is currently open in the Microsoft Access window. This method allows you to access the current database from Visual Basic without having to know the database name. Next, a table is created using the `CreateTableDef` method of a DAO Database object. This method requires that you specify a string or string variable to hold the name of the new TableDef object. For instance, the following line sets the object variable `tblNew` to point to a table named Agents:

```
Set tblNew = db.CreateTableDef("Agents")
```

Because a table must have at least one field, the next step in the table creation process is to use the `CreateField` method of the TableDef object to create fields. For instance, in the following statement:

```
Set fld = tblNew.CreateField("AgentID", dbText, 6)
```

- `tblNew` is a table definition variable.
- "AgentID" is a string specifying the name for the new field object.
- `dbText` is an integer constant that determines the data type of the new Field object (see Table 11.2).
- 6 is an integer indicating the maximum size in bytes for a text field. Text fields can hold from 1 to 255 bytes. This argument is ignored for other types of fields.

TABLE 11.2. Constants for the Type property in DAO Object Library (DataTypeEnum enumeration)

Data Type Name	Value	Description
dbAttachment	101	Attachment data
dbBigInt	16	Big integer data
dbBinary	9	Binary data
dbBoolean	1	Boolean (True/False) data
dbByte	2	Byte (8-bit) data
dbChar	18	Text data (fixed width)
dbComplexByte	102	Multivalue byte data
dbComplexDecimal	108	Multivalue decimal data
dbComplexDouble	106	Multivalue double-precision floating-point data
dbComplexGUID	107	Multivalue GUID data
dbComplexInteger	103	Multivalue integer data
dbComplexLong	104	Multivalue long integer data
dbComplexSingle	105	Multivalue single-precision floating-point data
dbComplexText	109	Multivalue text data (variable width)
dbCurrency	5	Currency data
dbDate	8	Date value data
dbDecimal	20	Decimal data (ODBCDirect only)
dbDouble	7	Double-precision floating-point data
dbFloat	21	Floating-point data (ODBCDirect only)
dbGUID	15	GUID data
dbInteger	3	Integer data
dbLong	4	Long integer data
dbLongBinary	11	Binary data (bitmap)

(Contd.)

Data Type Name	Value	Description
dbMemo	12	Memo data (extended text)
dbNumeric	19	Numeric data (ODBCDirect only)
dbSingle	6	Single-precision floating-point data
dbText	10	Text data (variable width)
dbTime	22	Data in time format (ODBCDirect only)
dbTimeStamp	23	Data in time and date format (ODBCDirect only)
dbVarBinary	17	Variable binary data (ODBCDirect only)

<b>NOTE</b>	<i>ODBCDirect workspaces are not supported since the release of Access 2007. Use ADO if you want to access external data sources without using the Microsoft Access database engine.</i> <i>Constants for complex data types and the dbAttachment data type do not apply to versions prior to Access 2007.</i>
-------------	---

When creating fields for your table, you may want to set certain field properties such as Validation Rule, Validation Text, Default Value, and Required. The Validation Rule property is a text string that describes the rule for validation. In the CreateTableDAO procedure, we require that each entry in the AgentID field begin with the letter “A.”

The Validation Text property is a string that is displayed to the user when the validation fails; that is, when the user attempts to enter data that does not comply with the specific validation rule.

The Default Value property sets or returns the default value of a Field object. In this example procedure, we make the data entry easier for the user by specifying “USA” as the default value in the Country field. Each new record will automatically have an entry of USA in the Country field. Because certain fields should not be left blank, you can ensure that the user enters data in a particular field by setting the Required property of that field to True.

In addition to built-in properties of an object, there are two other types of properties:

- Application-defined properties
- User-defined properties

The application-defined property is created only if you assign a value to that property. A classic example of such a property is the Description property of the TableDef object. To set the Description property of a table in the Access user interface, simply right-click on the table name and choose Table Properties, then

type the text you want in the Description field. Access will create a Description property for the table and will append it automatically to the Properties collection for that TableDef object. If you do not type a description in the Description field, Access will not create a Description property. Therefore, if you use the Description property in your code in this case, Access will display an error. For this reason, it is a good idea to check beforehand whether a referenced property exists. Users may create their own properties to hold additional information about an object.

The CreateTableDAO procedure demonstrates how to use the `CreateProperty` method of the TableDef object to create application-defined or user-defined properties. To create a property you will need to supply the name for the property, the property type, and the property value. For example, here's how to use the `CreateProperty` method to create a Caption property for the DateOfBirth field in the newly created table Agents:

```
Set prp = tblNew.Fields("DateOfBirth").CreateProperty("Caption")
```

Next, the data type of the Property object is defined:

```
prp.Type = dbText
```

See Table 11.2 earlier in the chapter for the names of the Type property constants in VBA.

Finally, a value is assigned to the new property:

```
prp.Value = "Date of Birth"
```

Instead of writing three separate lines of code, you can create a new property of an object with the following line:

```
Set prp = tblNew.Fields("DateOfBirth").  
CreateProperty("Caption", dbText, "Date of Birth")
```

A user-defined property must be appended to the Properties collection of the corresponding object. In this example procedure, the Caption property is appended to the Properties collection of the Field object, and the Description property is appended to the Properties collection of the TableDef object:

```
fld.Properties.Append prp  
tblNew.Properties.Append prp
```

After creating a field and setting its built-in, application-defined, or user-defined properties, the `Append` method is used to add the field to the Fields collection, as in the following example:

```
tblNew.Fields.Append fld
```

Once all the fields have been created and appended to the Fields collection, remember to append the new table to the TableDefs collection, as in the following example:

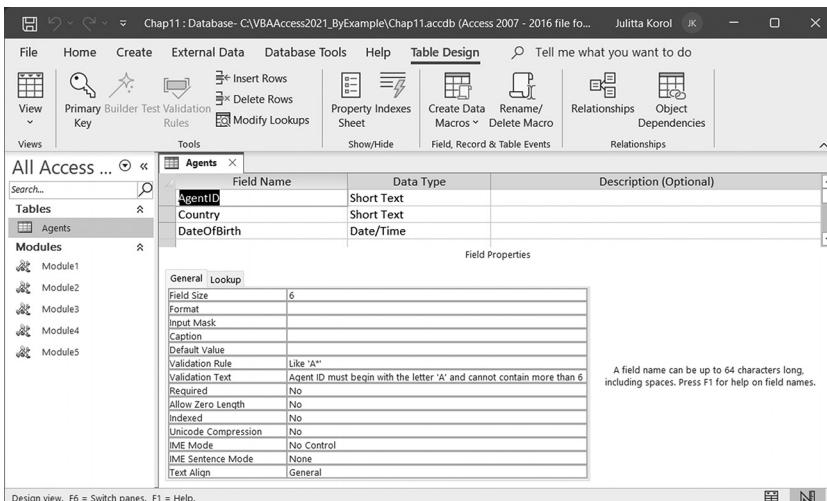
```
db.TableDefs.Append tblNew
```

You can delete user-defined properties from the Properties collection, but you can't delete built-in properties. If you set a property in the user interface, you don't need to create and append the property in code because the property is automatically included in the Properties collection.

After running the procedure code, a new table named Agents appears in the Microsoft Access window.

To check the value of the Description property for the Agents table that was set as a result of running the example procedure, right-click the Agents table in the database window, and choose Table Properties from the shortcut menu.

To check the properties that were set and defined in this procedure, activate the Agents table in Design view, click the field name for which you set or created a custom property in the code, and examine the corresponding field properties. Figure 11.1 shows the current settings of the Validation Rule and Validation Text properties for the AgentID field.



**FIGURE 11.1** You can create a database table like this one using VBA code. You can also set appropriate field properties programmatically.

In DAO, use the `CreateField` and `Append` methods to add new fields to the existing table.



### Hands-On 11.7 Adding a New Field to a Table

1. Copy the **Northwind\_Chap11.mdb** database file from the companion files to your C:\VBAAccess2021\_ByExample folder.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following Add\_NewFieldsDAO procedure:

```
Sub Add_NewFieldsDAO() +  
    Dim db As DAO.Database  
    Dim tdf As DAO.TableDef  
    Dim tblName As String  
  
    tblName = "Customers"  
  
    On Error GoTo ErrorHandler  
    Set db = OpenDatabase _  
        ("C:\VBAAccess2021_ByExample\Northwind_Chap11.mdb")  
    Set tdf = db.TableDefs(tblName)  
  
    MsgBox "Number of fields in the table: " & _  
        db.TableDefs(tblName).Fields.Count  
  
    With tdf  
        .Fields.Append .CreateField("NoOfMeetings", dbInteger)  
        .Fields.Append .CreateField("Result", dbMemo)  
    End With  
  
    MsgBox "Number of fields in the table: " & _  
        db.TableDefs(tblName).Fields.Count  
    db.Close  
    Exit Sub  
ErrorHandler:  
    MsgBox Err.Number & ": " & Err.Description  
End Sub
```

4. Choose **Run | Run Sub/UserForm** to run the Add\_NewFieldsDAO procedure. The Add\_NewFieldsDAO procedure uses the following `With...End With` construct to quickly add two new fields to an existing table:

```
With tdf  
    .Fields.Append .CreateField("NoOfMeetings", dbInteger)  
    .Fields.Append .CreateField("Result", dbMemo)  
End With
```

Each new field is appended to the Fields collection of the specified DAO TableDef object. In this example, we create a new field on the fly while calling the `Append` method. Be sure to include a space between the `Append` method and the dot operator in front of the `CreateField` method. To add two new fields to an existing table without using the `With...End With` construct, you would use the following statements:

```
tdf.Fields.Append tdf.CreateField("NoOfMeetings", dbInteger)  
tdf.Fields.Append tdf.CreateField("Result", dbMemo)
```

However, using the `With...End With` construct makes the code both clearer and faster to execute.

## **CREATING CALCULATED FIELDS**

---

Access has the ability to store calculated values in tables via a so-called *calculated field*. A classic example of the calculated field is a person's full name. A person's first and last names are stored in separate fields in an Access table. In versions of Access prior to 2010, the full name was generally obtained via a query by writing an expression that concatenated the first and last name:

```
Select [FirstName] & " " & [LastName] AS FullName
```

In Access 2010–2021, you can define the expression for the calculation in the calculated field and Access will store the calculated values in the table. With this feature, there is no need to calculate the person's full name in multiple locations in your Access application. When the underlying values change (for example, a female employee got married and the last name field used in the expression was updated), the expression will automatically update the value that is stored in the calculated field.

Calculated columns can be added to Access tables manually or with VBA. To create a calculated column using the manual method, open the table in Design view and enter the field name. In the Data Type column, select Calculated. At this point, Access will display the Expression Builder dialog box where you can enter the expression (see Figure 11.2).

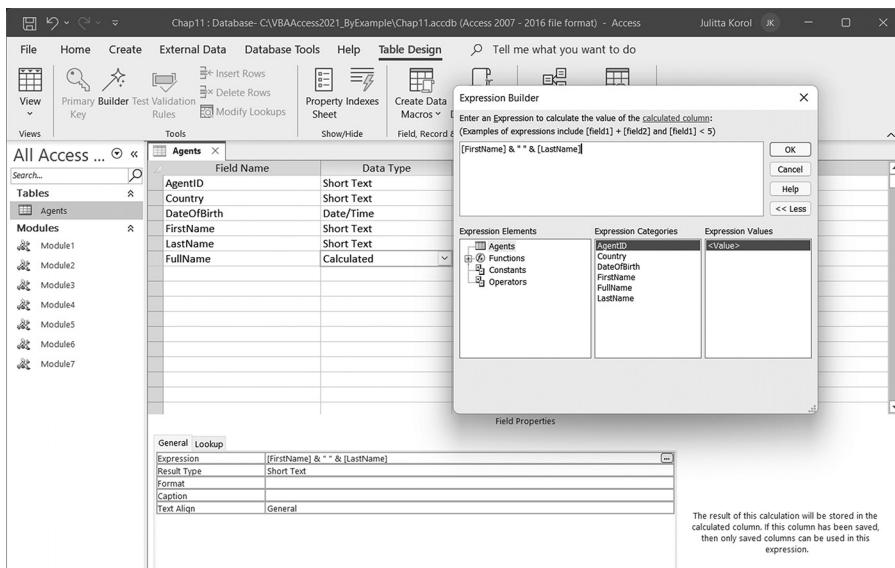
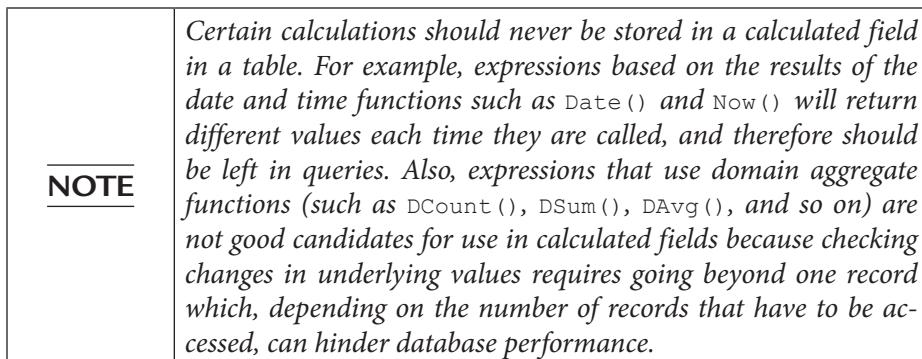


FIGURE 11.2. You can add a calculated field to a table using the table Design view and the Expression Builder.

To create a calculated field in DAO, you will need to set the Expression property of the DAO.Field2 object to the expression you'd like to use for the calculated field, as shown in Hands-On 11.8. A Field2 object represents a column of data in an Access table. It contains all of the same properties and methods as the Field object with the addition of several properties and methods that support field types added in Access 2007 (multivalue lookup fields and attachment fields) and Access 2010 (calculated fields).



### Hands-On 11.8 Creating a Calculated Field

1. In the VBE screen, choose **Insert | Module** and enter the following procedure in the module's Code window:

```
Sub CreateCalcField()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field2

    On Error GoTo ErrorHandler

    Set db = CurrentDb
    Set tdf = db.TableDefs("Agents")

    ' add two text fields
    tdf.Fields.Append tdf.CreateField _
        ("FirstName", dbText, 25)
    tdf.Fields.Append tdf.CreateField _
        ("LastName", dbText, 25)

    ' add a calculated field
    Set fld = tdf.CreateField("FullName", dbText, 50)
    fld.Expression = "[FirstName] & " " " & [LastName]"
    tdf.Fields.Append fld

    ExitHere:
    Set fld = Nothing
    Set tdf = Nothing
    Set db = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = 3211 Then
        ' table is open; need to close it to continue
        DoCmd.Close acTable, "Agents", acSaveYes
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
        Resume ExitHere
    End If
End Sub
```

2. Run the **CreateCalcField** procedure.

The **CreateCalcField** procedure adds three new fields (**FirstName**, **LastName**, and **FullName**) to the existing **Agents** table in the current database. To append

fields to the table, Access needs exclusive access to the table definition. The included ErrorHandler executes the statement that closes the table if it is found open:

```
DoCmd.Close acTable, «Agents», acSaveYes
```

Notice that before you can create a calculated field you need to ensure that the fields the calculation is based upon are also present in the table. After adding the required fields to the table, the calculated field is added and its expression for the calculation is defined as follows:

```
fld.Expression = "[FirstName] & " " & [LastName]"
```

Figure 11.2 earlier in this section shows the Agents table in Design view displaying the properties of the calculated field. To manually change the calculation expression, click the ellipsis button to the right of the Expression property to bring up the Expression Builder.

## **CREATING MULTIValue LOOKUP FIELDS**

---

Thanks to the introduction of the complex multivalue data type in the .accdb file format, table columns can store more than one value. This makes it easy for an Access user to create a lookup field without having to know much about setting table relationships. Access will automatically store the values entered in multivalue fields in hidden system tables and create proper table relationships if necessary. The source data for a multivalue field can be one of the following: value list, field list, or table/query. To have Access guide you in the creation of a multivalue field, choose Lookup Wizard in the Data Type column of the table's Design view.

Multivalue lookup fields are often referred to as complex fields because they use data types that begin with dbComplex (see Table 11.3).

**TABLE 11.3** Data types used by multivalue lookup fields

Data Type	Value	Description
dbComplexByte	102	Multivalue byte data
dbComplexDecimal	108	Multivalue decimal data
dbComplexDouble	106	Multivalue double-precision floating-point data
dbComplexGUID	107	Multivalue GUID data

(Contd.)

Data Type	Value	Description
dbComplexInteger	103	Multivalue integer data
dbComplexLong	104	Multivalue long integer data
dbComplexSingle	105	Multivalue single-precision floating-point data
dbComplexText	109	Multivalue text data (variable width)

The following hands-on exercise demonstrates how to use VBA to add a multivalue field named Literature to the Northwind 2007\_Chap11.accdb database's Customers table.



### Hands-On 11.9 Creating a Multivalue Lookup Field

1. Copy the Northwind 2007\_Chap11.accdb database file from the companion files to your C:\VBAAccess2021\_ByExample folder.
2. In the VBE screen, choose **Insert | Module** and enter the following **CreateMultiValueFld** procedure in the module's Code window:

```

Sub CreateMultiValueFld()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field
    Dim strDBName As String
    Dim strTblName As String
    Dim strLitItems As String
    Dim strPath As String

    On Error GoTo ErrorHandler

    strPath = "C:\VBAAccess2021_ByExample\
    strDBName = "Northwind 2007_Chap11.accdb"
    strLitItems = "Product Brochure;Product Flyer A;"
    strLitItems = strLitItems & "Product Flyer B"
    strTblName = "Customers"

    Set db = OpenDatabase(strPath & strDBName)
    Set tdf = db.TableDefs(strTblName)
    Set fld = tdf.CreateField("Literature", dbComplexText)
    tdf.Fields.Append fld

    With fld
        .Properties.Append .CreateProperty( _
            "DisplayControl", dbText, acComboBox)
    End With
End Sub

```

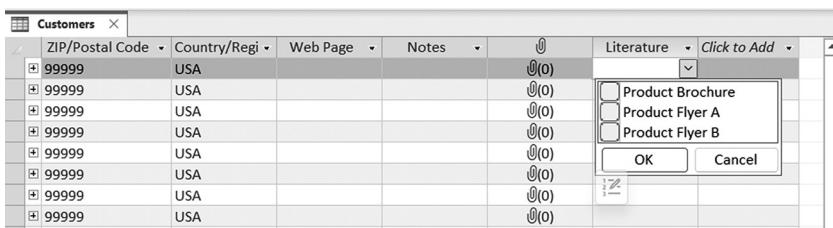
```

.Properties.Append .CreateProperty( _
"RowSourceType", dbText, "Value List")
.Properties.Append .CreateProperty( _
"RowSource", dbText, strLitItems)
.Properties.Append .CreateProperty( _
"BoundColumn", dbInteger, 1)
.Properties.Append .CreateProperty( _
"ColumnCount", dbInteger, 1)
.Properties.Append .CreateProperty( _
"ColumnWidths", dbText, "1")
.Properties.Append .CreateProperty( _
"ListWidth", dbText, "1.5")
.Properties.Append .CreateProperty( _
"AllowMultipleValues", dbBoolean, True)
.Properties.Append .CreateProperty( _
"AllowValueListEdits", dbBoolean, True)
End With

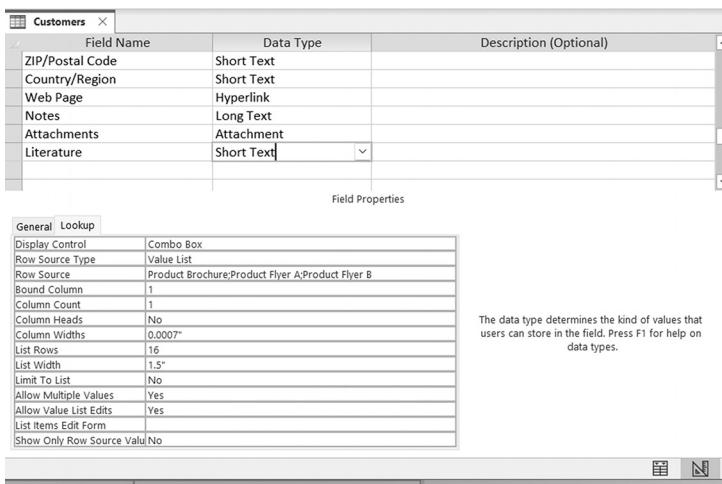
ExitHere:
    db.Close
    Set fld = Nothing
    Set tdf = Nothing
    Set db = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ": " & Err.Description
    Resume ExitHere
End Sub

```

3. Run the **CreateMultiValueFld** procedure.
4. Open the **C:\VBAAccess2021\_ByExample\Northwind 2007\_Chap11.accdb** database and check the newly created Literature field in the Customers table (see Figures 11.3 and 11.4).



**FIGURE 11.3** The multivalue lookup field (Literature) created by the VBA procedure in Hands-On 11.10 displays a combo box.



**FIGURE 11.4** The Field Properties Lookup tab contains numerous properties that tell Access how to display values in the Literature field.

## CREATING ATTACHMENT FIELDS

The Attachment data type makes it possible to store various types of external files directly in the database. This data type is only available in Access databases created in the .accdb file format in Access 2007–2021. Earlier versions of Access used the OLE Object data type for embedding external files within MDB databases, and this format continues to be available in Access 2021 for backward compatibility. The Attachment data type eliminates the bloating issues that plagued Access MDB databases whenever the OLE Object data type was used. To keep .accdb files as small as possible, Access compresses the uncompressed files in the attachments before storing them in a database.

The Attachment data type allows you to add multiple attachments to a single record. However, keep in mind that the maximum size of an attached data file cannot exceed 256 MB (megabytes). You can store as many external files as you want as long as you stay within 2 GB (gigabytes) of data, which is the maximum size of an Access database. You cannot restrict how many attachments are allowed in a database field. Also, some attachment file types are not supported. (You can see the list of blocked file extensions in the Access online help.)

You can work with attachments manually via the Attachments dialog box (see Figure 11.5) or programmatically using the Attachment object.

Hands-On 11.10 demonstrates how to create an Attachment field. You will revisit the attachments topic when you start working with records later in this chapter.



### Hands-On 11.10 Adding an Attachment Field to an Existing Table

1. In the VBE screen, choose **Insert | Module** and enter the following **CreateAttachmentFld** procedure in the module's Code window:

```
Sub CreateAttachmentFld()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field2

    On Error GoTo ErrorHandler

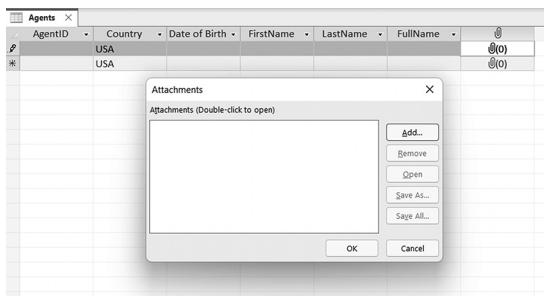
    Set db = CurrentDb
    Set tdf = db.TableDefs("Agents")

    ' add an attachment field
    Set fld = tdf.CreateField("AttachLiterature", _
        dbAttachment)
    tdf.Fields.Append fld

    ExitHere:
    Set fld = Nothing
    Set tdf = Nothing
    Set db = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = 3211 Then
        ' table is open; need to close it to continue
        DoCmd.Close acTable, "Agents", acSaveYes
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
        Resume ExitHere
    End If
End Sub
```

2. Run the **CreateAttachmentFld** procedure.

After running this procedure, the Agents table in the current database contains an extra field as shown in Figure 11.5. To add attachments, double-click the @() in the record to bring up the Attachments dialog box. To add and manipulate attachments programmatically, see Hands-On 11.23.

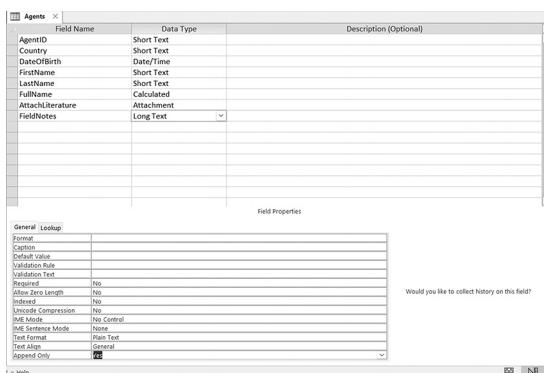


**FIGURE 11.5** The attachment field added with the VBA procedure in Hands-On 11.8 currently does not contain any attachments.

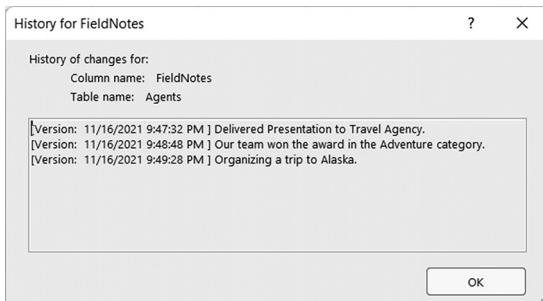
## CREATING APPEND ONLY MEMO FIELDS

Another type of complex multivalue field available in the .accdb file format is the Append Only memo field (see Figure 11.6). When the Append Only property is set to Yes, you can append data to the field, but you are not allowed to change the data that has been previously entered into this field. This feature is useful for keeping track of the changes made to the field.

Let's say you want to preserve the history of problems submitted by users. Every time you edit the data in the Append Only memo field, the date and time stamp and your changes are automatically saved to the version history of the field (see Figure 11.7). You can view the history of an Append Only memo field by right-clicking a value in the field and selecting Show column history from the shortcut menu. Custom Project 11.1 demonstrates how to add the Append Only memo field to an existing table and how to retrieve the history of data changes from this field.



**FIGURE 11.6** To collect history on a memo field, you must set the field's Append Only property to Yes.



**FIGURE 11.7** To access the memo field's history, right-click the field and select Show column history from the shortcut menu.

**NOTE**

*Beginning with Access 2013 there is no “memo” data type in the Data Type list. The Long Text data type has replaced the memo data type found in prior versions of Access. The Long Text data type is used for longer text fields (see FieldNotes field in Figure 11.6) and the Short Text data type is used for storing up to 255 characters.*



### Custom Project 11.1 Working with Append Only Memo Fields

1. In the VBE screen, choose **Insert | Module** and enter the following **CreateAppendOnlyFld** function procedure in the module's Code window:

```
Function CreateAppendOnlyFld(strTableName As String, _
    strFieldName As String)
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field2

    On Error GoTo ErrorHandler

    Set db = CurrentDb
    Set tdf = db.TableDefs(strTableName)

    ' create a memo field
    Set fld = tdf.CreateField(strFieldName, dbMemo)
    tdf.Fields.Append fld

    ' set the memo field to track version history
    fld.AppendOnly = True
```

```

ExitHere:
    Set fld = Nothing
    Set tdf = Nothing
    Set db = Nothing
    Exit Function
ErrorHandler:
    If Err.Number = 3211 Then
        ' table is open; need to close it to continue
        DoCmd.Close acTable, strTableName, acSaveYes
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
        Resume ExitHere
    End If
End Function

```

- Run the **CreateAppendOnlyFld** function procedure by typing the following statement in the Immediate window and pressing Enter:

```
CreateAppendOnlyFld "Agents", "FieldNotes"
```

Notice that after creating the FieldNotes memo field, you the AppendOnly property of this field is set to True to ensure that Access keeps the history of changes for this field.

- Let's enter some data in one or two records.
- Switch to the main Access window, and double-click the Agents table in the navigation pane.
  - Type in the data as shown in Figure 11.8. Recall that you don't need to enter data in the FullName field because this is a calculated field and Access will perform the required calculation based on the defined expression.



The screenshot shows a Microsoft Access datasheet titled "Agents". The table has columns: AgentID, Country, Date of Birth, FirstName, LastName, FullName, and FieldNotes. There are three rows of data:

AgentID	Country	Date of Birth	FirstName	LastName	FullName	FieldNotes
A100	USA	7/12/1995	Barbara	McDonald	Barbara McDonald	0(0)
A101	USA	5/4/1964	Ronald	Sepia	Ronald Sepia	0(0)
*	USA					0(0)

**FIGURE 11.8.** Entering sample data in a Datasheet view.

- In the FieldNotes field for Barbara McDonald, type the following text:  
**Delivered Presentation to Travel Agency.**
- In the FieldNotes field for Ronald Sepia, type the following text:  
**Mr. Brook invited us to a dinner party tomorrow.**

7. Move back to Barbara McDonald's record and in the FieldNotes enter the following text, overwriting the previously written text:  
**Our team won the award in the Adventure category.**
8. Press **Enter** to record the changes to the field.
9. Move back to the FieldNotes field of Barbara McDonald's record and type the following text:  
**Organizing a trip to Alaska.**
10. Continue to enter more data in the first two records of the Agents table so that you can build up some history in the FieldNotes Append Only memo field.
11. After you are done with the data entry, right-click on the FieldNotes field in each record and choose **Show Column History** to check out the version history. The record history for the first record is shown in Figure 11.7 earlier.
12. Close the Agents table.

You can retrieve the history of values that have been stored in a memo field by using the `ColumnHistory` method of the Application object. For example, the following statement entered on one line in the Immediate window will print all the notes for an agent whose AgentID equals A100 (see Figure 11.9).

```
?Application.ColumnHistory("Agents", "FieldNotes",
"AgentID='A100'")
```

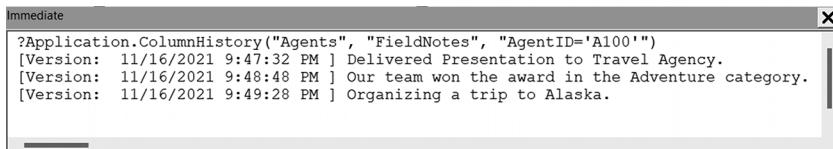


FIGURE 11.9 Retrieving the history of values stored in a memo field.

Notice that the `ColumnHistory` method requires three parameters: the name of the table that contains the Append Only memo field, the name of the memo field, and a string used to locate a record in the table. Let's now write a complete procedure that will retrieve the history data into three separate items: MemoDate, MemoTime, and MemoText.

13. In the same module where you entered the previous function procedure in this project, enter the following **RetrieveMemoHistory** function procedure:

```
Function RetrieveMemoHistory(strTableName As String, _
    strMemoFldName As String, strSearchFld As String, _
    strCriteria As String)
    Dim arrayString() As String
    Dim MemoText As String
```

```

Dim i As Integer
Dim strSearch As String
Dim startPos As Integer
Dim EndDatePos As Integer
Dim EndTimePos As Integer
Dim MemoDate As Date
Dim MemoTime As Date

arrayString = Split(Application.ColumnHistory( _
    strTableName, strMemoFldName, _
    strSearchFld & "=" & strCriteria & ""), "[Version: ]")

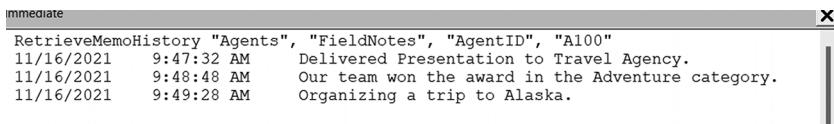
If UBound(arrayString) = -1 Then
    MsgBox "There is no history data for this field."
    Exit Function
End If
For i = 1 To UBound(arrayString)
    startPos = 1
    strSearch = arrayString(i)
    EndDatePos = InStr(startPos, strSearch, " ")
    MemoDate = CDate(Left(strSearch, EndDatePos - 1))
    startPos = EndDatePos + 1
    EndTimePos = InStr(startPos, arrayString(i), "]") - 3
    MemoTime = CDate(Mid(strSearch, startPos, _
        EndTimePos - startPos))
    startPos = EndTimePos + 3
    strSearch = Trim(Replace(strSearch, vbCrLf, ""))
    MemoText = Right(strSearch, Len(strSearch) - startPos)
    Debug.Print MemoDate, MemoTime, MemoText
Next
End Function

```

- 14.** Run the **RetrieveMemoHistory** function procedure from the Immediate window by entering the following:

```
RetrieveMemoHistory "Agents", "FieldNotes", "AgentID", "A100"
```

- 15.** The procedure prints to the Immediate window the history string broken into three columns, as shown in Figure 11.10.



The screenshot shows the Microsoft Word Immediate window with the title bar 'Immediate'. The window contains the following text:

```

RetrieveMemoHistory "Agents", "FieldNotes", "AgentID", "A100"
11/16/2021 9:47:32 AM Delivered Presentation to Travel Agency.
11/16/2021 9:48:48 AM Our team won the award in the Adventure category.
11/16/2021 9:49:28 AM Organizing a trip to Alaska.

```

**FIGURE 11.10** The history data from the FieldNotes column is output to the Immediate window via a VBA procedure.

As mentioned earlier, the Application object's `ColumnHistory` method is used in VBA to retrieve memo column history data. Because Access returns this data in a single string and we want to divide it into separate items, we use the VBA `Split` function. This function is ideal for breaking a long string into an array of substrings based on a specified delimiter. It returns a zero-based, one-dimensional array where each substring is an element. To hold the result of this function, the `RetrieveMemoHistory` function defines an array variable of the String data type named `arrayString`.

The first argument of the `Split` function specifies the string expression you want to split. The string that will be returned by the `ColumnHistory` method of the Application object is as follows:

```
arrayString = Split(Application.ColumnHistory( _  
    strTableName, strMemoFldName, _  
    strSearchFld & "=" & strCriteria & ""), "[Version: ]")
```

The second argument of the `Split` function specifies a string that is used to identify substring limits. You can split a string on a single character, a space, or a group of characters. Because each history item is separated by a line feed and a carriage return (`vbCrLf`), you might think that it's a good idea to break the Access-generated history string into separate lines by using the `vbCrLf` delimiter. Well, it isn't, simply because memo fields allow carriage returns. A better delimiter is something that does not conflict with anything the user may enter into the memo field. You should be able to use the "[Version: " string that Access adds to each line of the history string without having to worry about unexpected results. Notice that there are two spaces after the colon that we also want to include in the delimiting string. Now that we have eliminated from the history string extraneous text ([Version:]), we need to iterate through the array elements using the `For...Next` loop. However, there is no point doing this if the `arrayString` variable does not contain any elements. To check for that, we can use the `UBound` function, which will return -1 when the array is empty. While enumerating the history data, the procedure uses several variables to determine the character position where the date and time strings end (`EndDatePos`, `EndTimePos`). We also use the `startPos` variable to specify at which position in the search string the search should begin. Before extracting the date and time strings, we find the end character positions for these strings using the VBA `InStr` function:

```
EndDatePos = InStr(startPos, strSearch, " ")
```

The `InStr` function returns the position of the first occurrence of one string within another. The first parameter is optional. It indicates the character position where the search should start. Obviously, we want to start at the first position so that we can examine the entire string. The second parameter is the string to search in. We are storing it in the `strSearch` variable. The third parameter of the `InStr` function is the string you want to find. In this case, we want to find a single space after the date. Notice that the single space separates the date from the time (see Figure 11.9 earlier). The `InStr` function also has an optional fourth argument that specifies the type of string comparison. When omitted, Access performs a binary comparison where each character matches only itself. This is the default. The `InStr` function will return a zero (0) when the string you are looking for is not found in the string you searched.

We also use other text functions (`Right`, `Left`, `Mid`) to extract a specified number of characters from the string. The `Right` function is used to extract characters from the right side of the string; the `Left` function does the same but from the left side of the string; and the `Mid` function extracts characters from the middle of the string. Notice that the text operations also require the use of the built-in `Len` function that returns the total number of characters in the specified string.

We defined the `MemoDate` and `MemoTime` variables as Date; thus, after extracting the date and time strings from the searched string, we use the `CDate` function to convert them into the Date format.

## CREATING RICH TEXT MEMO FIELDS

---

The .accdb file format boasts the Rich Text feature in memo fields. This allows you to format your memo fields in a datasheet with the bold, italics, underline, and other formatting options that are available via the Ribbon. To enable the Rich Text feature, open a table in Design view; in the Field Properties area for the selected memo field, set the Text Format property to Rich Text (see Figure 11.11). When you use the Rich Text feature in a memo field, Access stores the data in HTML format. Figure 11.12 shows an example of rich text formatting for a Notes field in the Employees table in the Northwind 2007\_Chap11.accdb database.

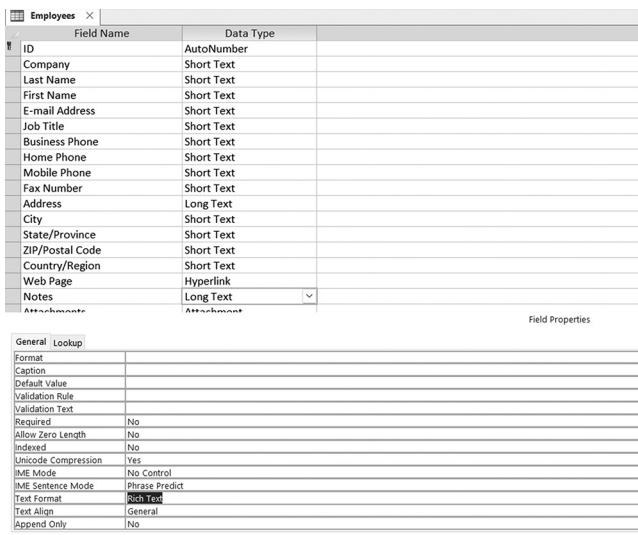


FIGURE 11.11 Enabling Rich Text Format for a memo field.

ID	Company	Last Name	First Name	E-mail Address	Job Title	City	Notes
1	Northwind Traders	Freehafer	Nancy	nancy@northwindtraders.com	Sales Representative	Seattle	
2	Northwind Traders	Cencini	Andrew	andrew@northwindtrader.s.com	Vice President, Sales	Bellevue	Joined the company as a sales representative, was promoted to sales manager and was then named vice president of sales.
3	Northwind Traders	Kotas	Jan	jan@northwindtraders.com	Sales Representative	Redmond	<i>Was hired as a sales associate and was promoted to sales representative.</i>

FIGURE 11.12 The Notes field for Jan Kotas in the Employees table of the Northwind 2007\_Chap11.accdb database is shown here with the rich text formatting.



### Hands-On 11.11 Creating a Rich Text Memo Field

1. In the VBE screen, choose **Insert | Module** and enter the following two procedures in the module's Code window:

```
Sub CreateRichMemoFld()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field2
    Dim strTbl As String
    Dim strFld As String

    On Error GoTo ErrorHandler
```

```
strTbl = "Agents"
strFld = "PersonalNotes"

Set db = CurrentDb
Set tdf = db.TableDefs(strTbl)

' add an attachment field
Set fld = tdf.CreateField(strFld, dbMemo)
tdf.Fields.Append fld

ConvertToRichText strTbl, strFld

ExitHere:
    Set fld = Nothing
    Set tdf = Nothing
    Set db = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = 3211 Then
        ' table is open; need to close it to continue
        DoCmd.Close acTable, strTbl, acSaveYes
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
        Resume ExitHere
    End If
End Sub

Sub ConvertToRichText(strTbl As String, _
    strFld As String)

    With CurrentDb
        With .TableDefs(strTbl)
            With .Fields(strFld)
                On Error Resume Next
                .Properties("TextFormat") = 1
                If Err.Number = 3270 Then _
                    .Properties.Append .CreateProperty _
                    ("TextFormat", dbByte, 1)
            End With
        End With
    End With
End Sub
```

2. Run the **CreateRichMemoFld** procedure.

The **CreateRichMemoFld** procedure begins by creating a memo field called **PersonalNotes** in the current database's **Agents** table. Once the field is appended to the **TableDefs** collection of the **Agents** table, we need to set its **TextFormat** property to **RichText**. We do this by calling the **ConvertToRichText** procedure. If the **TextFormat** property already exists, this procedure will set the **TextFormat** property of the **FieldNotes** field to 1, which denotes the Rich Text setting. The default value of the **TextFormat** property is 0 (Plain Text). If the property is not found, error 3270 will occur, and at this point we want Access to execute the statement that will create the new property called **TextFormat** and then append it to the **Properties** collection:

```
If Err.Number = 3270 Then  
    .Properties.Append .CreateProperty _  
        ("TextFormat", dbByte, 1)
```

The last argument in the **CreateProperty** method specifies the type of setting for the Rich Text memo field. As mentioned earlier, 1 represents Rich Text, and 0 represents Plain Text.

3. Open the **Agents** table in Design view and verify the changes made by the **CreateRichMemoFld** procedure in this hands-on exercise.
4. Close the **Agents** table.

## REMOVING A FIELD FROM A TABLE

---

You may remove any field from an existing table, whether or not this field contains data. However, you can't delete a field after you have created an index that references that field. You must first delete the index.

In DAO, use the **Fields** collection's **Delete** method to remove a field from an existing table.



### Hands-On 11.12 Removing a Field from a Table

The following procedure removes from the **Customers** table two fields that were added by the procedure in Hands-On 11.5.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **DeleteFields.DAO** procedure:

```
Sub DeleteFields.DAO()  
    Dim db As DAO.Database
```

```
Dim tdf As DAO.TableDef
Dim strDBName As String
Dim strTblName As String
Dim strFolder As String

On Error GoTo ErrorHandler

strFolder = "C:\VBAAccess2021_ByExample\
strDBName = "Northwind_Chap11.mdb"
strTblName = "Customers"

Set db = OpenDatabase(strFolder & strDBName)
Set tdf = db.TableDefs(strTblName)

MsgBox "Number of fields in the table: " & _
db.TableDefs(strTblName).Fields.Count

With tdf
    .Fields.Delete "NoOfMeetings"
    .Fields.Delete "Result"
End With

MsgBox "Number of fields in the table: " & _
db.TableDefs(strTblName).Fields.Count

db.Close
Exit Sub
ErrorHandler:
    MsgBox Err.Number & ": " & Err.Description
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

## **RETRIEVING TABLE PROPERTIES**

---

Additional code is available in the companion files.

File Name: ListTableProperties.DAO.txt

Description: Use the Properties collection of the DAO TableDef object to list properties of the Agents table in the Chap11.accdb database.

## LINKING A DBASE TABLE

---

In DAO, to link a table to an Access database, use the `CreateTableDef` method to create a new table:

```
Set myTable = db.CreateTableDef("TableDBASE")
```

Next, specify the `Connect` property of the `TableDef` object. For example, the following statement specifies the connect string:

```
myTable.Connect = "dBase 5.0;Database=C:\VBAAccess2021_ByExample"
```

Next, specify the `SourceTableName` property of the `TableDef` object to indicate the actual name of the table in the source database:

```
myTable.SourceTableName = "Customer.dbf"
```

Finally, use the `Append` method to append the `TableDef` object to the `TableDefs` collection:

```
db.TableDefs.Append myTable
```

Additional code is available in the companion files.

File Name: LinkDBaseTable.DAO.txt

Description: Linking a dBASE table to the current database

## CREATING INDEXES

---

In DAO, indexes are created using the `CreateIndex` method for a `TableDef` object. The following statement creates an index named `PrimaryKey`:

```
Set idx = tdf.CreateIndex("PrimaryKey")
```

To ensure that the correct type of index is created, you need to set the index properties. For example, the `Primary` property of an index indicates that the index fields constitute the primary key for the table:

```
idx.Primary = True
```

Use the `Unique` property to specify whether the values in an index must be unique:

```
idx.Unique = True
```

The `Required` property indicates whether the index can accept `Null` values. When you set this property to `True`, nulls will not be accepted:

```
idx.Required = True
```

Use the `IgnoreNulls` property to determine whether a record with a `Null` value in the index fields should be included in the index:

```
idx.IgnoreNulls = False
```

To actually index a table, you must use the `CreateField` method on the `Index` object to create a `Field` object for each field you want to include in the index:

```
Set fld = idx.CreateField("AgentID", dbText)
```

Note that in the Access 2021 User Interface the `AgentID` field will display its data type as Short Text when you open the table in the Design view. Beginning with Access 2016, the Short Text replaces the Text data type. In programming, use `dbText` to indicate that the field should hold character data. This has not changed from the prior versions.

Once the `Field` object is created, you need to append it to the `Fields` collection:

```
idx.Fields.Append fld
```

The last step in index creation is appending the `Index` object to the `Indexes` collection:

```
tdf.Indexes.Append idx
```

The procedure in Hands-On 11.13 uses DAO to create a primary key index in the Agents table.



### Hands-On 11.13 Creating a Primary Key

The procedure in this hands-on exercise uses the Agents table in the Chap11.accdb database.

1. In the Visual Basic Editor window, choose **Insert | Module**.

In the module's Code window, type the following `Create_PrimaryKeyDAO` procedure:

```
Sub Create_PrimaryKeyDAO()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field
    Dim idx As DAO.Index

    Set db = CurrentDb
    Set tdf = db.TableDefs("Agents")
```

```

' create a Primary Key
Set idx = tdf.CreateIndex("PrimaryKey")
idx.Primary = True
idx.Required = True
idx.IgnoreNulls = False
Set fld = idx.CreateField("AgentID", dbText)
idx.Fields.Append fld

' add the index to the Indexes collection in the Agents table
tdf.Indexes.Append idx
db.Close
Set db = Nothing
End Sub

```

**2. Choose Run | Run Sub/UserForm** to execute the procedure.

To verify that the index was created, open the Agents table in the Chap11.accdb database. Activate the Design view and click the Indexes button on the Ribbon. The result of running the Create\_PrimaryKeyDAO procedure is shown in Figure 11.13.

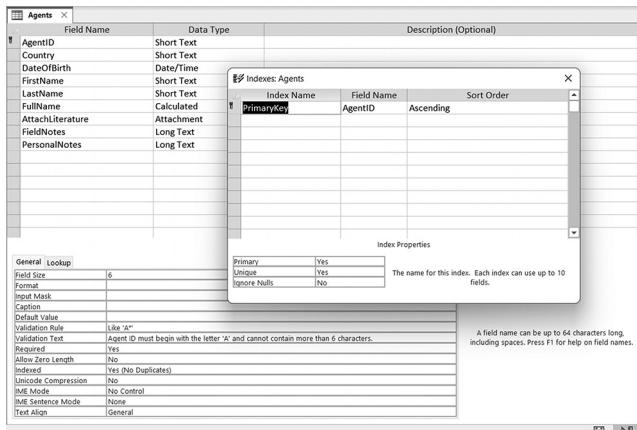


FIGURE 11.13. The Indexes window after running the procedure in Hands-On 11.12.

## ADDING A MULTIPLE-FIELD INDEX TO A TABLE

The procedure in the next hands-on exercise shows how to use DAO to add a multiple-field index to the Employees table in the Northwind\_Chap11.mdb database.



### Hands-On 11.14 Adding a Multiple-Field Index to an Existing Table

1. In the Visual Basic Editor window, choose **Insert | Module**.

In the module's Code window, type the **Add\_MultiFieldIndex** procedure shown here:

```
Sub Add_MultiFieldIndex()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field
    Dim idx As DAO.Index
    Dim strDB As String
    Dim strTblName As String

    strDB = "C:\VBAAccess2021_ByExample\Northwind_Chap11.mdb"
    strTblName = "Employees"
    Set db = OpenDatabase(strDB)
    Set tdf = db.TableDefs(strTblName)

    Set idx = tdf.CreateIndex("Location")
    Set fld = idx.CreateField("City", dbText)
    idx.Fields.Append fld

    Set fld = idx.CreateField("Region", dbText)
    idx.Fields.Append fld
    tdf.Indexes.Append idx

    db.Close
    Set db = Nothing
    Debug.Print "New index (Location) was created."
End Sub
```

2. Choose **Run | Run Sub/UserForm** to execute the procedure.

The **Add\_MultiFieldIndex** procedure creates a two-field index in the Employees table. To create an index, use the **CreateIndex** method on a **TableDef** object. Next, use the **CreateField** method on the **Index** object to create the first field to be included in the index, and then append this field to the **Fields** collection. Repeat the same steps for the second field you want to include in the index. It is important to remember that the order in which the fields are appended has an effect on the index order. If you open the Indexes window in the Employees table of the Northwind\_Chap11 database after running this procedure, the Location index will consist of two fields, as shown in Figure 11.14.



FIGURE 11.14 The Location index was created by running the procedure in Hands-On 11.13.

## FINDING AND READING RECORDS

---

To work with data, you need to learn about the `Recordset` object. The `Recordset` object represents a set of records in a table or a set of records returned by executing a stored query or an SQL statement. Each column of a recordset represents a field, and each row represents a record. The `Recordset` is a temporary object and is not saved in the database. All recordset objects cease to exist after the procedure ends. All open recordset objects are contained in the `Recordsets` collection. Creating and using the recordset objects depends on the type of object library (DAO/ADO) that you've selected for your programming task. In the following sections of this chapter, you will learn various methods of opening the `Recordset` object. You will also find out how to navigate in the recordset, and how to find, filter, read, and count the records. Only DAO recordsets will be covered here. Refer to the next chapter if you require the use of the ADO recordsets.

## INTRODUCTION TO DAO RECORDSETS

---

In DAO there are five types of Recordset objects:

- Table-type
- Dynaset-type

- Snapshot-type
- Forward-only-type
- Dynamic-type

Each of these recordsets offers a different functionality (see Table 11.4). You create a `Recordset` object using the `OpenRecordset` method. The type of the recordset is specified by the `type` argument of the `OpenRecordset` method. If the recordset's type is not specified, DAO will attempt to create a Table-type recordset. If this type isn't available, attempts are made to create a Dynaset, Snapshot, or Forward-only-type recordset object.

TABLE 11.4 Types of DAO Recordsets

Recordset Type	Description
Table-type	Used to access records in a table stored in an Access database. You can retrieve, add, update, and delete records in a single table.
Dynaset-type	Used to retrieve, add, update, and delete records from one or more tables in a database as well as any table that is linked to the Access database.
Snapshot-type	Used to access records from a local table stored in an Access database as well as any linked table or a query. Snapshot recordsets contain a copy of the records in RAM (random access memory) and provide no direct access to the underlying data. They are used for reading data only—you can't use them to add, update, or delete records.
Forward-only-type	This is a special type of a Snapshot recordset that only allows you to scroll forward through the records. It provides the fastest access when you want to make a single pass through the data.
Dynamic-type	This recordset is generated by a query based on one or more tables. It allows you to add, change, or delete records from a row-returning query. In addition, it includes the records that other users may have added, modified, or deleted.

To find and read database records, you must understand how to navigate through the recordset. When you open a `Recordset` object, the first record is the current record. All recordsets have a current record.

- To move to subsequent records, use the `MoveNext` method.
- To move to the previous record, use the `MovePrevious` method.
- The `MoveFirst` and `MoveLast` methods move the cursor to the first and last records, respectively.
- If you call the `MoveNext` method when the cursor is already pointing to

the last record, the cursor will move off the last record to the area known as end of file (EOF), and the EOF property will be set to `True`.

- If you call the `MoveNext` method when the EOF property is `True`, an error is generated because you cannot move past the end of the file. Similarly, by calling the `MovePrevious` method when the cursor is pointing to the first record, you will move the cursor to the area known as beginning of file (BOF). This will set the BOF property to `True`. When the BOF property is `True` and you call the `MovePrevious` method, an error will be generated.

When navigating through a recordset, you may want to mark a specific record in order to return to it at a later time. You can use the `Bookmark` property to obtain a unique identification for a specific record.

The `Recordset` object has numerous properties and methods. We will discuss only those properties and methods that are required for performing a specific task, as demonstrated in the example procedures.

### **Opening Various Types of Recordsets**

Use the `OpenRecordset` method to create or open a recordset. For example, to open a Table-type recordset on a table named `tblClients`, use the following statement:

```
Set rst = CurrentDb.OpenRecordset("tblClients", _ dbOpenTable)
```

Notice that the second argument in the `OpenRecordset` method specifies the type of recordset. The `RecordsetType` constants (shown in Table 11.5) can be used here.

**TABLE 11.5** Constants used to specify the type of a DAO Recordset object

Type Constant	Value	Description
<code>dbOpenTable</code>	1	Opens a Table-type recordset
<code>dbOpenDynaset</code>	2	Opens a Dynaset-type recordset
<code>dbOpenSnapshot</code>	4	Opens a Snapshot-type recordset
<code>dbOpenForwardOnly</code>	8	Opens a Forward-only-type recordset
<code>dbOpenDynamic</code>	16	Opens a Dynamic-type recordset

In the preceding example, if you don't specify a recordset type, a Table-type recordset will be created based on `tblClients`. A Table-type recordset represents the records in a single table in a database.

The `OpenRecordset` method opens a new recordset for reading, adding, updating, or deleting records from a database. The `OpenRecordset` method can also be performed on a query. Note that a query can only be opened as a Dynaset or Snapshot Recordset object. For example, to open a Recordset based on a query, use the following statements:

```
Dim db As DAO.Database
Dim rst As DAO.Recordset
Set db = CurrentDb()
Set rst = db.OpenRecordset("qryMyQuery", _ dbOpenSnapshot)
```

The procedure in Hands-On 11.15 demonstrates how to open various types of DAO recordsets on the Customers table in the Northwind 2007\_Chap11.accdb database and return the total number of records.



### Hands-On 11.15 Opening Table-, Dynaset-, and Snapshot-Type Recordsets

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **ThreeRecordsetsDAO** procedure:

```
Sub ThreeRecordsetsDAO()
    Dim db As DAO.Database
    Dim tblRst As DAO.Recordset
    Dim dynaRst As DAO.Recordset
    Dim snapRst As DAO.Recordset
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\"
    strDb = "Northwind 2007_Chap11.accdb"
    Set db = OpenDatabase(strPath & strDb)
    Set tblRst = db.OpenRecordset("Customers", _
        dbOpenTable)
    Debug.Print "Records in a table: " & _
        tblRst.RecordCount

    Set dynaRst = db.OpenRecordset("Customers", _
        dbOpenDynaset)
    Debug.Print "Records in a Dynaset: " & _
        dynaRst.RecordCount
    dynaRst.MoveLast
    Debug.Print "Records in a Dynaset: " & _
        dynaRst.RecordCount
```

```
Set snapRst = db.OpenRecordset("Customers", _
    dbOpenSnapshot)
Debug.Print "Records in a Snapshot: " & _
    snapRst.RecordCount
snapRst.MoveLast
Debug.Print "Records in a Snapshot: " & _
    snapRst.RecordCount

tblRst.Close
dynaRst.Close
snapRst.Close
db.Close
Set db = Nothing
SendKeys "^g"
End Sub
```

### 3. Choose **Run | Run Sub/UserForm** to execute the procedure.

Notice that to get the correct count of records in the Dynaset and Snapshot recordsets, you need to invoke the `MoveLast` method to access all the records. Counting records is covered in more detail in the next section.

The last statement in this procedure (`SendKeys "^\g"`) activates the Immediate window so that you can see the results for yourself. This is the same as pressing **Ctrl+G** keys.

## Opening a Snapshot and Counting Records

When you want to search tables or queries, you will get the fastest results by opening a Snapshot-type recordset. A snapshot is simply a non-updatable set of records that contain fields from one or more tables or queries. Snapshot-type recordsets can be used only for retrieving data. Use the `OpenRecordset` method to create or open a recordset. For example, to open a Snapshot-type recordset on a table named `Customers`, use the following statement:

```
Set rst = CurrentDb.OpenRecordset("Customers", dbOpenSnapshot)
```

At times, you may need to know where you are in a recordset. There are two properties that can be used to determine your position in the recordset:

- The `AbsolutePosition` property allows you to position the current record pointer at a specific record based on its ordinal position in a Dynaset- or Snapshot-type recordset object. This property lets you determine the current record number. Zero (0) refers to the first record in the recordset object. If there is no current record, the `AbsolutePosition` property returns -1. However, because the position of a record changes when

the preceding records are deleted, you should rely more on bookmarks to position the current record. The `AbsolutePosition` property can be used only with Dynasets and Snapshots. Because the `AbsolutePosition` property value is zero-based, 1 is added to the `AbsolutePosition` value to display the current record information:

```
MsgBox "Current record: " & rst.AbsolutePosition + 1
```

- The `PercentPosition` property shows the current position relative to the number of records that have been accessed. Both `AbsolutePosition` and `PercentPosition` are not accurate until you move to the last record.

The procedure in Hands-On 11.16 attempts to get the total number of records in a Snapshot-type recordset by using the `RecordCount` property.



### Hands-On 11.16 Opening a Snapshot-Type Recordset and Retrieving the Number of Records

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **OpenSnapshot** procedure shown here:

```
Sub OpenSnapshot()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset

    Set db = OpenDatabase("C:\VBAAccess2021_ByExample\" & _
        "Northwind 2007_Chap11.accdb")
    Set rst = db.OpenRecordset("Customers", _
        dbOpenSnapshot)

    MsgBox "Current record: " & rst.AbsolutePosition + 1
    MsgBox "Number of records: " & rst.RecordCount
    rst.MoveLast
    MsgBox "Current record: " & rst.AbsolutePosition + 1
    MsgBox "Number of records: " & rst.RecordCount
    rst.Close
    Set rst = Nothing
    db.Close
    Set db = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.  
The `RecordCount` property of the `Recordset` object returns the number of records that have been accessed.

Zero (0) is returned if there are no records in the recordset, and 1 is returned if there are records in the recordset. If you open a Table-type recordset and check the RecordCount property, it will return the total number of records in a table. However, if you open a Dynaset- or Snapshot-type recordset, the RecordCount property will return 1, indicating that the recordset contains records. To find out the total number of records in a Dynaset or Snapshot, call the `MoveLast` method prior to retrieving the RecordCount property value. The record count becomes accurate after you've visited all the records in the recordset.

### **Retrieving the Contents of a Specific Field in a Table**

---

To retrieve the contents of any field, start by creating a recordset based on the desired table or query, then loop through the recordset, printing the field's contents for each record to the Immediate window.

The procedure in Hands-On 11.17 generates a listing of all clients in the Customers table. Customer names are retrieved starting from the last record (see the `MoveLast` method). The BOF property of the Recordset object determines when the beginning of your recordset is reached.



#### **Hands-On 11.17 Retrieving Field Values**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **ReadFromEnd** procedure shown here:

```
Sub ReadFromEnd()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strDb As String

    strDb = "C:\VBAAccess2021_ByExample\" & _
            "Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(strDb)
    Set rst = db.OpenRecordset("Customers", _
                             dbOpenTable)
    rst.MoveLast

    Do Until rst.BOF
        Debug.Print rst!Company
        rst.MovePrevious
    Loop
```

```

SendKeys "^g"
rst.Close
Set rst = Nothing
db.Close
Set db = Nothing
End Sub

```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### Moving Between Records in a Table

---

All recordsets have a current position and a current record. The current record is usually the record at the current position. However, the current position can be before the first record and after the last record. You can use one of the `Move` methods in Table 11.6 to change the current position.

**TABLE 11.6** Move methods used with DAO Recordsets

Method Name	Description
<code>MoveFirst</code>	Moves to the first record
<code>MoveLast</code>	Moves to the last record
<code>MoveNext</code>	Moves to the next record
<code>MovePrevious</code>	Moves to the previous record
<code>Move n</code>	Moves forward or backward <i>n</i> positions

The procedure in Hands-On 11.18 demonstrates how to move between records in the Employees table using the Table-type or Dynaset-type recordset.



### Hands-On 11.18 Moving between Records in a Table

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `NavigateRecords` procedure:

```

Sub NavigateRecords()
    Dim db As DAO.Database
    Dim tblRst As DAO.Recordset
    Dim dynaRst As DAO.Recordset
    Dim strDb As String

    strDb = "C:\VBAAccess2021_ByExample\" & _
            "Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(strDb)
    Set tblRst = db.OpenRecordset("Employees")

```

```
tblRst.MoveFirst

Do While Not tblRst.EOF
    Debug.Print "Employee: " & tblRst! [Last Name]
    tblRst.MoveNext
Loop

Set dynaRst = db.OpenRecordset("Employees", _
    dbOpenDynaset)
dynaRst.MoveFirst

Do While Not dynaRst.EOF
    Debug.Print "Hello " & dynaRst! [Last Name]
    dynaRst.MoveNext
Loop

tblRst.Close
dynaRst.Close

Set tblRst = Nothing
Set dynaRst = Nothing
db.Close
Set db = Nothing
SendKeys "^g"
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### **Finding Records in a Table-Type Recordset**

---

While the `Move` methods are convenient for looping through records in a `Recordset` object, you should use the `Seek` or `Find` methods to look for specific records. When you know exactly which record you want to find in a Table-type recordset and the field you are searching is indexed, the quickest way to find that record is to use the `Seek` method. One thing to remember with the `Seek` method is that the table must contain an index. The `Index` property must be set before the `Seek` method can be used. If you try to use the `Seek` method on a Table-type recordset without first setting the current index, a runtime error will occur. The `Seek` method searches through the recordset and locates the first matching record. Once the record is found, it is made the current record and the `NoMatch` property is set to `False`. If the record is not found, the `NoMatch` property is set to `True` and the current record is undefined. Table 11.7 lists comparison operators that you can use with the `Seek` method.

**TABLE 11.7** Comparison operators used with the Seek method

Operator	Description
“=”	Finds the first record whose indexed field is equal to the specified value
“>=”	Finds the first record whose indexed field is greater than or equal to the specified value
“>”	Finds the first record whose indexed field is greater than the specified value
“<=”	Finds the first record whose indexed field is less than or equal to the specified value
“<”	Finds the first record whose indexed field is less than the specified value

The comparison operator used with the `Seek` method must be enclosed in quotes. If there are several records that match your criteria, the `Seek` method returns the first record it finds. The `Seek` method cannot be used to search for records in a linked table. You must use the `Find` methods (see the next section) for locating specific records in linked tables, as well as Dynaset- and Snapshot-type recordsets. The procedure in Hands-On 11.19 searches for an employee whose last name begins with the letter “K.”



### Hands-On 11.19 Finding Records in a Table-Type Recordset

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module’s Code window, type the following **FindRecordsInTable** procedure:

```
Sub FindRecordsInTable()
    Dim db As DAO.Database
    Dim tblRst As DAO.Recordset
    Dim strDb As String

    strDb = "C:\VBAAccess2021_ByExample\" &
           "Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(strDb)

    Set tblRst = db.OpenRecordset("Employees", _
        dbOpenTable)
    ' find the first employee in the table whose
    ' name begins with the letter "K"

    tblRst.Index = "Last Name"
    tblRst.Seek ">=", "K"
```

```

If Not tblRst.NoMatch Then
    MsgBox "Found the following employee: " & _
        tblRst![Last Name]
Else
    MsgBox "There is no employee with such a name."
End If

tblRst.Close
Set tblRst = Nothing
db.Close
Set db = Nothing
End Sub

```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### Finding Records in Dynasets or Snapshots

---

Use the `Find` methods to search for a record in Dynaset-type and Snapshot-type recordsets. Table 11.8 lists the available `Find` methods.

**TABLE 11.8** Find methods in a DAO Recordset

Method Name	Description
<code>FindFirst</code>	Finds the first matching record in the recordset
<code>FindNext</code>	Finds the next matching record, starting at the current record
<code>FindPrevious</code>	Finds the previous matching record, starting at the current record
<code>FindLast</code>	Finds the last matching record in the recordset

If a record is not found for the given criteria, the `NoMatch` property of the `Recordset` object is set to `True`. Before searching for records, set a bookmark at the current record. If the search fails, you will be able to use the bookmark to return to the current record; otherwise, you will get the error “No current record.” Each record in a `Recordset` object has a unique bookmark that you can use to locate that record. To get the current record’s bookmark, move the cursor to that record and assign the value of the `Bookmark` property of the `Recordset` object to a Variant variable:

```

Dim mySpot As Variant
mySpot = dynaRst.Bookmark

```

In Hands-On 11.20, the bookmark is set on the first record of a Dynaset-type recordset. The procedure then searches for employees whose name ends with the string “er.” The asterisk (\*) in the search string is a wildcard character representing any number of letters (\*er).

To return to the bookmarked record, set the `Bookmark` property to the value held by the Variant variable:

```
dynaRst.Bookmark = mySpot
```

While recordsets based on local Access tables support bookmarks, non-Access databases may not support them. To determine whether a Recordset object supports bookmarks, you can check the `Bookmarkable` property. Bookmarks are supported if this property is `True`.

```
If dynaRst.Bookmarkable Then  
    mySpot = dynaRst.Bookmark  
End If
```

If the Recordset object does not support bookmarks, an error occurs. You can set as many bookmarks as you wish. Bookmarks can be created for a record other than the current record by moving to the desired record and assigning the value of the `Bookmark` property to a String variable that identifies that record.



### Hands-On 11.20 Finding a Record in a Dynaset-Type Recordset

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `FindRecInDynaset` procedure:

```
Sub FindRecInDynaset()  
    Dim db As DAO.Database  
    Dim dynaRst As DAO.Recordset  
    Dim mySpot As Variant  
    Dim strDb As String  
  
    strDb = "C:\VBAAccess2021_ByExample\" & _  
        "Northwind 2007_Chap11.accdb"  
  
    Set db = OpenDatabase(strDb)  
    Set dynaRst = db.OpenRecordset("Employees", _  
        dbOpenDynaset)  
  
    MsgBox "Current employee: " & _  
        dynaRst![Last Name]  
    mySpot = dynaRst.Bookmark  
  
    ' find clients whose name ends  
    ' with the string "er"  
    dynaRst.FindFirst "[Last Name] Like '*er'"
```

```
Do While Not dynaRst.NoMatch
    Debug.Print dynaRst![Last Name]
    dynaRst.FindNext "[Last Name] Like '*er'"
Loop

dynaRst.Bookmark = mySpot
MsgBox "Back to: " & dynaRst![Last Name]
dynaRst.Close

Set dynaRst = Nothing
db.Close
Set db = Nothing
SendKeys "^g"
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

The names of all employees that match the search criteria are printed to the Immediate window.

### Finding the nth Record in a Snapshot

The procedure in Hands-On 11.21 demonstrates how to locate the *n*th record in a Snapshot-type recordset.



#### Hands-On 11.21 Finding the nth Record in a Snapshot-Type Recordset

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **FindNthRecord** procedure shown here:

```
Sub FindNthRecord()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim fld As DAO.Field
    Dim totalRec As Integer
    Dim nth As String
    Dim strDb As String

    strDb = "C:\VBAAccess2021_ByExample\" &
        "Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(strDb)
    Set rst = db.OpenRecordset("Employees", _
        dbOpenSnapshot)
    rst.MoveLast
```

```
totalRec = rst.RecordCount
rst.MoveFirst
nth = InputBox("Enter the number of positions" & _
    " to move forward:")

On Error Resume Next
If totalRec > nth Then
    rst.Move nth
    For Each fld In rst.Fields
        Debug.Print fld.Name & ":" & fld.Value
    Next fld
Else
    MsgBox "Please enter a value that is less than " &
        & totalRec & "."
End If

rst.Close
Set rst = Nothing
db.Close
Set db = Nothing
End Sub
```

**3.** Choose **Run | Run Sub/UserForm** to execute the procedure.

Notice that immediately after opening the recordset, the `MoveLast` method is used to ensure that all records have been visited. The total number of records is then retrieved with the `RecordCount` property of the `Recordset` object and stored in the `totalRec` variable. Next, the `MoveFirst` method is used to return to the first record and the `InputBox` method is used to prompt the user for the number of positions to move forward in the recordset. If the user-supplied value is less than the total number of records, the cursor moves to the specified record and the `For Each` loop is used to print this record's field names and values to the Immediate window. An attempt to move beyond the end of the recordset will cause an error. Therefore, the procedure displays a message if the user-supplied position to move to is greater than the total number of records.

---

## WORKING WITH RECORDS

Now that you've familiarized yourself with various methods of opening, moving around in, and finding records, and reading the contents of a recordset, let's look at DAO techniques for adding, modifying, copying, deleting, and sorting records.

## **Adding a New Record**

---

In the Access user interface, before you can add a new record to a table, you must first open the appropriate table. In code, you simply open the Recordset object by calling the `OpenRecordset` method. For example, the following statements declare and open the Recordset object based on the Employees table:

```
Dim tblRst As DAO.Recordset  
Set tblRst = db.OpenRecordset("Employees")
```

Once the Recordset object is open, use the `AddNew` method to create a blank record. The following is an example:

```
tblRst.AddNew
```

Next, you may set values for all or some of the fields in the new record. You must set the field's value if the `Required` property of a field is set to `True`. In the Access user interface in Table Design view, there will be a Yes entry next to the Required property if the entry in the selected field is required. Here are some examples of setting field values in code:

```
tblRst.Fields("Last Name").Value = "Smith"  
tblRst.Fields("Job Title").Value = "Marketing Director"
```

Note that because `Value` is the default property of a `Field` object, the use of this keyword is optional and it was omitted in the code of the example procedure in Hands-On 11.21.

After filling in the field values, you need to use the `Update` method on the Recordset object to ensure that the newly added record is saved:

```
tblRst.Update
```

Hands-On 11.22 demonstrates how to add a new record to the Employees table and populate some of its fields with values.



### **Hands-On 11.22 Adding a New Record to a Table**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **AddNewRec.DAO** procedure shown here:

```
Sub AddNewRec_DAO()  
    Dim db As DAO.Database  
    Dim tblRst As DAO.Recordset  
    Dim strDb As String
```

```
strDb = "C:\VBAAccess2021_ByExample\Northwind 2007_Chap11.accdb"

Set db = OpenDatabase(strDb)
Set tblRst = db.OpenRecordset("Employees")

With tblRst
    .AddNew
    .Fields("Company") = "Northwind Traders"
    .Fields("Last Name") = "Smith"
    .Fields("First Name") = "Regina"
    .Fields("Job Title") = "Marketing Director"
    .Fields("E-mail Address") = "regina@northwindtraders.com"
    .Update
End With

tblRst.Close
Set tblRst = Nothing
db.Close
Set db = Nothing
End Sub
```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

In a Table-type recordset, the new record is placed in the order identified by the table's index. In a Dynaset-type recordset, the new record is added at the end of the recordset. When you add a new record to a table, the new record does not become the current record. The record that was current prior to adding the new record remains current. In other words, while a new record is being added to the end of the table, the cursor remains in the record that was selected prior to adding a new record. You can, however, make the newly added record current by using the Bookmark and LastModified properties, like this:

```
tblRst.Bookmark = tblRst.LastModified
```

## Adding Attachments

---

In Hands-On 11.10, you learned how to programmatically add an Attachments field to a table. Hands-On 11.23 demonstrates how to use VBA to add external files to records in the Customers table of the Northwind 2007\_Chap11.accdb database.



### Hands-On 11.23 Using DAO to Add an Attachment to a Table Record

1. Copy the **External Docs** folder from the companion files to your **VBAAccess2021\_ByExample** folder.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following **AddAttachmentToRecord** procedure:

```
Option Compare Database
Option Explicit

Sub AddAttachmentToRecord()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset2
    Dim rstChild As DAO.Recordset2
    Dim addFlag As Boolean

    Const dirPath = "C:\VBAAccess2021_ByExample\
    Const subDirName = "External Docs\
    Const strFile = "California3.jpg"
    Const strDb = "Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(dirPath & strDb)

    ' Open the recordset for the Customers table
    Set rst = db.OpenRecordset("Customers")
    ' move to the 16th customer (count records from 0)

    rst.Move 15

    ' initialize child recordset
    Set rstChild = rst.Fields("Attachments").Value

    If rstChild.RecordCount > 0 Then
        ' check if the specified file is already attached
        Do Until rstChild.EOF
            If rstChild.Fields("FileName").Value = strFile Then
                addFlag = True
                Exit Do
            End If
        Loop
    End If
```

```
If addFlag Then MsgBox "The specified file " & _
    strFile & " is already attached to this record."  
  
If Not addFlag Then
    ' put the parent recordset in Edit mode
    rst.Edit
    ' add a new record to the child recordset
    rstChild.AddNew
    ' load the attachment file
    rstChild.Fields("FileData").LoadFromFile _
        dirPath & subDirName & strFile
    ' update both the child and parent recordsets
    rstChild.Update
    rst.Update
    MsgBox "Successfully attached " & strFile & _
        " to " & rst.Fields(1).Value & " record."
End If  
  
Set rstChild = Nothing
rst.Close
Set rst = Nothing
Set db = Nothing
End Sub
```

This procedure adds an attachment to the 16th record in the Customers table. This is a record for Company P. The child recordset holds the records for the Attachment field. Prior to adding a record to this recordset, the procedure checks the RecordCount property of the child recordset to verify that the specified file is not already attached. If RecordCount is greater than zero (0), then the `addFlag` Boolean variable is set to `True` and the user will see a message that the file is already attached. The procedure will then end. If the `addFlag` Boolean variable is `False`, then we know we can add the file. Note that before adding a new record to the child recordset, you must put the parent recordset in Edit mode using the `Edit` method of the Recordset object. Next, call the `AddNew` method of the child recordset to add a new child record, and use the `LoadFromFile` method to load the new attachment file. Be sure to update both the child and parent recordsets.

4. Run the **AddAttachmentToRecord** procedure.
5. Open the **Customers** table in the Northwind 2007\_Chap11.database. Find the 16th record in the table and check out the paper clip column. It should indicate that one record is attached. You can view the attached file by double-clicking the attachment field in the 16th record (see Figure 11.15).
6. Close the **Customers** table and exit the Northwind 2007\_Chap11 database.

7. Return to the Visual Basic Editor window in the Chap11.accdb database and run the AddAttachmentToRecord procedure again to test the condition when the attachment file already exists for the specified record.

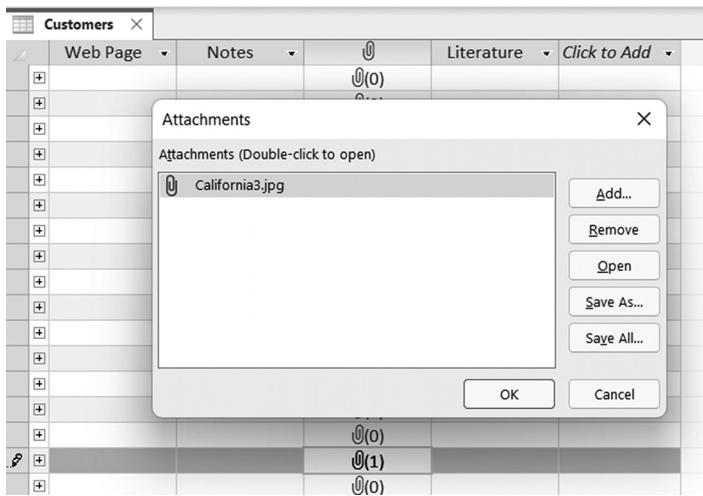


FIGURE 11.15 Attachment files can be added to records in an Access table manually using the Attachments dialog box or via VBA programming.

### Adding Values to Multivalue Lookup Fields

Earlier in this chapter, you used DAO to create a multivalue lookup field called Literature. You can add a new value to a multivalue field by modifying its RowSource property when the RowSourceType property is set to Value List. The function procedure in the next hands-on exercise adds new values to the Literature multivalue lookup field in the Customers table.



#### Hands-On 11.24 Using DAO to Add Values to a Multivalue Lookup Field

This hands-on exercise requires the completion of Hands-On 11.7.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **AddToMultiValueList** function procedure:

```
Function AddToMultiValueList(strTblName As String, _  
    strMultiFldName As String, strNewVal As String)
```

```
Dim db As DAO.Database
Dim tdf As DAO.TableDef
Dim fld As DAO.Field2
Dim prp As DAO.Property
Dim strDb As String
Dim strPath As String

strPath = "C:\VBAAccess2021_ByExample\
strDb = "Northwind 2007_Chap11.accdb"

On Error GoTo ErrorHandler

Set db = OpenDatabase(strPath & strDb)

Set tdf = db.TableDefs(strTblName)
Set fld = tdf.Fields(strMultiFldName)

If fld.Properties("RowSourceType").Value = _
    "Value List" Then
    Set prp = fld.Properties("RowSource")
    Debug.Print prp.Value
    If InStr(1, prp.Value, strNewVal) = 0 Then
        prp.Value = prp.Value & Chr(59) & Chr(34) & _
            strNewVal & Chr(34)
        Debug.Print prp.Value
    End If
End If
ExitHere:
Set prp = Nothing
Set fld = Nothing
Set tdf = Nothing
Set db = Nothing
Exit Function
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
GoTo ExitHere
End Function
```

This function procedure takes three arguments: the `strTblName` argument specifies the name of a table where a multivalue lookup field is located; the `strMultiFldName` argument specifies the name of a multivalue lookup field, and the `strNewVal` argument specifies the value you want to add to the list. To work with the specified table, we begin by setting the `tdf` object variable to point to our table:

```
Set tdf = db.TableDefs(strTblName)
```

Recall that the DAO `TableDefs` collection contains `TableDef` objects, which are table definitions. Each `TableDef` object contains a `Fields` collection. We set up the `fld` object variable to gain access to the specified multivalue lookup field via the `Fields` collection of the `TableDef` object:

```
Set fld = tdf.Fields(strMultiFldName)
```

The `Field` object has a collection of properties. Before we do any work, we check that the `RowSourceType` property is set to `Value List`. If this test is `True`, we need to get the current value of the `RowSource` property. We set up the `prp` object variable to point to this property and write the property value to the Immediate window:

```
Set prp = fld.Properties("RowSource")
Debug.Print prp.Value
```

Because we only want to have unique values in the multivalue lookup field, we need to check if the value passed in the `strNewVal` parameter is already in the value list. To do this, you can use the VBA `InStr` function that was introduced in Chapter 11:

```
If InStr(1, prp.Value, strNewVal) = 0 Then
```

Recall that the `InStr` function returns the position of the first occurrence of one string within another. The first parameter is optional. It indicates the character position where the search should start. Obviously, we want to start at the first position so that we can examine the entire value list string. The second parameter is the string to search in. The value of the `prp` variable contains the following string when the function is called:

```
"Product Brochure";"Product Flyer A";"Product Flyer B"
```

The third parameter of the `InStr` function is the string you want to find. We will specify this string when we call the function procedure in the next step. The `InStr` function also has an optional fourth argument that specifies the type of string comparison. When omitted, Access performs a binary comparison where each character matches only itself. This is the default.

The `InStr` function will return a zero (0) when the string you are looking for was not found in the string you searched in. We will then add the new item to the current `RowSource` value list:

```
prp.Value = prp.Value & Chr(59) & Chr(34) &
strNewVal & Chr(34)
```

To add a new value to the list, we use the concatenation character (&). The Chr(59) function will give us the required semicolon (;) and the Chr(34) is for the double quotes (""). The underscore character (\_) simply breaks the long code line into two lines. Notice that the procedure uses the ErrorHandler code to trap errors that may result from entering a nonexistent table or column name.

3. Run the AddToMultiValueList function procedure by typing the following statement in the Immediate window and pressing **Enter** to execute:

```
AddToMultiValueList "Customers", "Literature",
"Sales Contract"
```

After you execute the function procedure, the Immediate window should display the original value of the RowSource property and the new updated value:

```
"Product Brochure";"Product Flyer A";"Product Flyer B"
"Product Brochure";"Product Flyer A";"Product Flyer B";
"Sales Contract"
```

Run the AddToMultiValueList function procedure again by typing the following statement in the Immediate window and pressing **Enter** to execute:

```
AddToMultiValueList "Customers", "Literature", "Dinner
Invitation"
```

You should now see in the Immediate window the following two strings:

```
"Product Brochure";"Product Flyer A";"Product Flyer B";
"Sales Contract"
"Product Brochure";"Product Flyer A";"Product Flyer B";"Sales
Contract";"Dinner Invitation"
```

Open the **Customers** table in the Northwind 2007\_Chap11.accdb database and take a look at the drop-down list in the **Literature** field. In addition to the values added before, you should see two entries that were added by the VBA code in this hands-on exercise: *Sales Contract* and *Dinner Invitation*.

4. Close the **Customers** table and exit the Northwind 2007\_Chap11 database.

## **Modifying a Record**

---

To edit an existing record, use the `OpenRecordset` method to open the Recordset object. Next, locate the record you want to modify. In a Table-type recordset, you can use the `Seek` method and a table index to find a record that meets your

criteria. In Dynaset-type and Snapshot-type recordsets, you can use any of the Find methods (FindFirst, FindNext, FindPrevious, FindLast) to locate the appropriate record. However, recall that you can edit data only in Table-type or Dynaset-type recordsets (Snapshots are used for retrieving data only). Once you've located the record, use the Edit method of the Recordset object and proceed to change field values. When you are done with the record modification, invoke the Update method for the Recordset object.

The procedure in Hands-On 11.25 demonstrates how to modify a record in the Employees table.



### Hands-On 11.25 Modifying a Record in a Table

This hands-on exercise requires the completion of Hands-On 11.22.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **ModifyRecord.DAO** procedure:

```
Sub ModifyRecord_DAO()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strFind As String
    Dim intResult As Integer
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\"
    strDb = "Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(strPath & strDb)
    Set rst = db.OpenRecordset("Employees", dbOpenTable)

    rst.MoveFirst
    ' change the Zip/Postal Code of all employees
    ' from 99999 to 99998

    Do While Not rst.EOF
        With rst
            .Edit
            .Fields("Zip/Postal Code") = "99998"
            .Update
            .MoveNext
        End With
    Loop
```

```
' find the record with the last name of Smith
' enter data in Country/Region field
strFind = "Smith"
rst.MoveFirst
rst.Index = "Last Name"

rst.Seek "=", strFind
MsgBox rst! [Last Name]
Debug.Print rst.EditMode
rst.Edit

rst! [Country/Region] = "USA"
If rst.EditMode = dbEditMode Then
    intResult = MsgBox("Do you want to save the " & _
        "changes to this record?", vbYesNo, _
        "Save or Cancel Changes?")
End If
If intResult = 6 Then ' Save changes
    rst.Update
ElseIf intResult = 7 Then ' Cancel changes
    rst.CancelUpdate
End If

rst.Close
Set rst = Nothing
db.Close
Set db = Nothing
End Sub
```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

The procedure in Hands-On 11.25 opens a Table-type recordset based on the Employees table and makes a change in the Zip/Postal Code of all employees. Next, the procedure locates a specific employee record. Note that the `Index` property must be set before using the `Seek` method for searching the Table-type recordset. If you set the `Index` property to an index that doesn't exist, a runtime error will occur. Once the desired record is located, the procedure displays the employee name in a message box. The exclamation point (!) is used to separate an object's name from the name of the collection of which it is a member. Because the default collection of the `Recordset` object is the `Fields` collection, you can omit the default collection name. Next, the procedure places the found employee record into `Edit mode` and modifies the value of the `Country/Region` field. The `EditMode` property of the `Recordset` object is used to determine if the `Edit` operation is in progress. The `EditModeEnum`

constants, which are shown in Table 11.9, indicate the state of editing for the current record. Before committing the changes to the data, the user is asked to verify if the changes should be saved or canceled. If the Yes button is selected in the message box, the Recordset's `Update` method is called; otherwise, the `CancelUpdate` method of the Recordset object will discard the changes to the current record.

TABLE 11.9 EditModeEnum constants used in theEditMode property of the DAO Recordset object

Constant Name	Value	Description
<code>dbEditNone</code>	0	Edit method not invoked
<code>dbEditInProgress</code>	1	Edit method invoked
<code>dbEditAdd</code>	2	<code>AddNew</code> method invoked

<b>NOTE</b>	<p><i>At times when working with records, you will need to leave the record and discard the changes. To cancel any pending updates to the data, call the <code>CancelUpdate</code> method of the DAO Recordset object. This method aborts any changes you've made to the current row. You can use the <code>CancelUpdate</code> method to cancel any changes made after the <code>Edit</code> or <code>AddNew</code> method was invoked. You can check if there is a pending operation that can be canceled by using the <code>EditMode</code> property of the Recordset object.</i></p>
-------------	--

## Deleting a Record

To delete an existing record, open the Recordset object by calling the `OpenRecordset` method, then locate the record you want to delete. In a Table-type recordset, you can use the `Seek` method and a table index to find a record that meets your criteria. In a Dynaset-type recordset, you can use any of the `Find` methods (`FindFirst`, `FindNext`, `FindPrevious`, `FindLast`) to locate the appropriate record. Next, use the `Delete` method on the Recordset object to perform the deletion. Before using the `Delete` method, it is a good idea to write code to ask the user to confirm or cancel the deletion. Immediately after a record is deleted, there is no current record. Use the `MoveNext` method to move the record pointer to an existing record.

The example procedure in Hands-On 11.26 deletes those employees who have an ID greater than 9.



### Hands-On 11.26 Deleting a Record

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **DeleteRecord.DAO** procedure:

```
Sub DeleteRecord.DAO()
    Dim db As DAO.Database
    Dim tblRst As DAO.Recordset
    Dim counter As Integer
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\
    strDb = "Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(strPath & strDb)

    ' delete all the employees with ID greater than 9
    Set tblRst = db.OpenRecordset("Employees")
    tblRst.MoveFirst
    Do While Not tblRst.EOF
        Debug.Print tblRst!ID
        If tblRst![ID] > 9 Then
            tblRst.Delete
            counter = counter + 1
        End If
        tblRst.MoveNext
    Loop

    MsgBox "Number of deleted records: " & counter
    tblRst.Close
    Set tblRst = Nothing
    db.Close
    Set db = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

The statement `Do While Not tblRst.EOF` tells Visual Basic to execute the statements inside the loop until the end of file (EOF) is reached. The conditional statement inside the loop checks the value of the ID field and deletes the current record only if the specified condition is True. Every time a record is deleted, the `counter` variable's value is increased by 1. The `counter` variable stores the total number of deleted records. After the record is deleted, the `Mov-`

`eNext` method is called to move the record pointer to the next existing record as long as the end of file has not yet been reached. Even though you can use the `Delete` method and the `While` loop to remove the required records as shown in Hands-On 11.26, it is more efficient to delete records with a `Delete` query, which is covered later in this chapter.

## Deleting Attachments

---

The following hands-on exercise uses the `Delete` method of the `Recordset2` object to delete an attachment from a table record.



### Hands-On 11.27 Using DAO to Delete an Attachment from a Table Record

This hands-on exercise requires the completion of Hands-On 11.23.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following

**RemoveAttachmentFromRecord** procedure:

```
Sub RemoveAttachmentFromRecord()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset2
    Dim rstChild As DAO.Recordset2
    Dim removeFlag As Boolean

    Const dirName =
        "C:\VBAAccess2021_ByExample\External Docs\
    Const strFile = "California3.jpg"
    Const strDb =
        "C:\VBAAccess2021_ByExample\Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(strDb)

    ' Open the recordset for the Customers table
    Set rst = db.OpenRecordset("Customers")
    ' move to the 16th customer
    rst.Move 15
    ' get the child recordset for the Attachments field
    Set rstChild = rst.Fields("Attachments").Value
    ' search for the attachment file and remove it
    ' if found

    Do Until rstChild.EOF
```

```

If rstChild.Fields("FileName").Value = _
    strFile Then
    rstChild.Delete
    removeFlag = True
End If
rstChild.MoveNext
Loop

' display a message
If Not removeFlag Then
    MsgBox "The specified file " & strFile & _
        " is not attached to this record.", _
        vbOKOnly + vbInformation, "Nothing to Remove"
Else
    MsgBox "The specified file " & strFile & _
        " was deleted from this record.", _
        vbOKOnly + vbInformation, "Attachment Removed"
End If

' cleanup code
rstChild.Close
Set rstChild = Nothing
rst.Close
Set rst = Nothing
Set db = Nothing
End Sub

```

3. Run the RemoveAttachmentFromRecord procedure.
4. Open the **Customers** table in the Northwind 2007\_Chap11 database and navigate to the 16th record. The Attachment field in this record should indicate that there are no attachments.
5. Close the **Customers** table and exit the Northwind 2007\_Chap11 database.
6. Run the RemoveAttachmentFromRecord procedure again to test the condition when the attachment file for the specified record does not exist.

### Copying Records to an Excel Worksheet

You can copy the contents of a DAO Recordset object directly to an Excel worksheet or a worksheet range by using the Workbook Range object's `CopyFromRecordset` method.

- To copy all the records in the Recordset object to a worksheet range starting at cell A1, use the following statement:

```

Set rng = objSheet.Cells(2, 1)
rng.CopyFromRecordset rst

```

The `rst` following the name of the method is an object variable representing a Recordset object.

- To copy five records to a worksheet range, use the following statement:

```
Set rng = objSheet.Cells(2, 1)  
rng.CopyFromRecordset rst, 5
```

- To copy five records and four fields to a worksheet range, use the following statement:

```
Set rng = objSheet.Cells(2, 1)  
rng.CopyFromRecordset rst, 5, 4
```

You can also specify the number of records (rows) and fields to be copied using variables:

```
Set rng = objSheet.Cells(2, 1)  
rng.CopyFromRecordset rst, myRows, myColumns
```

The procedure in Hands-On 11.28 uses the `CopyFromRecordset` method to copy data from the Employees table to an Excel worksheet (see Figure 11.16).



### Hands-On 11.28 Copying Records to an Excel Worksheet

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `ExportToExcel.DAO` procedure:

```
Sub ExportToExcel.DAO()  
    Dim db As DAO.Database  
    Dim rst As DAO.Recordset  
    Dim xlApp As Object  
    Dim wkb As Object  
    Dim objSheet As Object  
    Dim rng As Object  
    Dim strExcelFile As String  
    Dim strDb As String  
    Dim strTable As String  
    Dim count As Integer  
    Dim iCol As Integer  
    Dim rowsToReturn As Integer  
    Dim strPath As String  
  
    strPath = "C:\VBAAccess2021_ByExample\"  
    strDb = "Northwind 2007_Chap11.accdb"
```

```
strTable = "Employees"
strExcelFile = CurrentProject.Path & _
    "\ExcelFromAccess.xls"

' If Excel file already exists, delete it
If Dir(strExcelFile) <> "" Then Kill strExcelFile

Set db = OpenDatabase(strPath & strDb)
Set rst = db.OpenRecordset(strTable)

' get the number of records from the recordset
count = rst.RecordCount

rowsToReturn = CInt(InputBox _
    ("How many records to copy?"))

If rowsToReturn <= count Then

    ' set the reference to Excel and make
    ' Excel visible
    Set xlApp = CreateObject("Excel.Application")
    xlApp.Application.Visible = True

    ' set references to the Excel workbook
    ' and worksheet
    Set wkb = xlApp.Workbooks.Add
    Set objSheet = xlApp.ActiveWorkbook.sheets(1)
    objSheet.Activate

    ' write column names to the first
    ' worksheet row
    For iCol = 0 To rst.Fields.Count - 1
        objSheet.Cells(1, iCol + 1).Value = _
            rst.Fields(iCol).Name
    Next

    ' specify the cell range that will
    ' receive the data
    Set rng = objSheet.Cells(2, 1)

    ' copy the specified number of records
    ' to the worksheet
    rng.CopyFromRecordset rst, rowsToReturn

    ' autofit the columns to make the data fit
```

```
objSheet.Columns.AutoFit

' close the workbook
' and save it in Excel 97-2003 file format
wkb.SaveAs FileName:=strExcelFile, _
    FileFormat:=56
wkb.Close

' quit Excel and release object variables
Set objSheet = Nothing
Set wkb = Nothing
xlApp.Quit
Set xlApp = Nothing
Else
    MsgBox "Please specify a number less than " _
        & count + 1 & "."
End If

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

3. Position the insertion point anywhere within the procedure code and choose **Debug | Step Into** to execute the procedure one line at a time. (Press **F8** to execute each statement.)

This procedure creates a recordset based on the Employees table and stores the total number of records in the `count` variable. The user is asked to specify the number of records to copy to Excel. If the specified number is less than or equal to the total number of records in the recordset, the code proceeds to copy the records to Excel using the `CopyFromRecordset` method. Notice that the procedure uses the `As Object` clause to declare object variables that will contain references to Excel objects when the procedure is run. When you define an object variable as `Object`, the variable is late bound. This means that VBA does not know what type of object the variable references until the program is run. To set a reference to Excel, it is necessary to use the `CreateObject` function. Once the object is created (`Excel.Application`), it is referenced with the object variable (`xlApp`). The `CreateObject` function will create a new instance of the Excel application. To use the current instance or to start Excel and load a specific file while Excel is already running, use the `GetObject` function. To view what's going on while the procedure is running, set the `Visible` property of the Microsoft Excel application to `True`. Then, if

you run the ExportToExcel\_DAO procedure in step mode, you will be able to check the contents of the Excel window as you execute each statement.

ID	Company	Last Name	First Name	E-mail Address	Job Title	Business Phone	Home Phone	Mobile Phone	Fax Number	Address
1	Northwind Traders	Freehafer	Nancy	nancy@northwindtraders.com	Sales Representative	(123)555-0100	(123)555-0102		(123)555-0103	123 1st Avenue
2	Northwind Traders	Cencini	Andrew	andrew@northwindtraders.com	Vice President, Sales	(123)555-0100	(123)555-0102		(123)555-0103	123 2nd Avenue
3	Northwind Traders	Kotas	Jan	jan@northwindtraders.com	Sales Representative	(123)555-0100	(123)555-0102		(123)555-0103	123 3rd Avenue
4	Northwind Traders	Sergienko	Mariya	mariya@northwindtraders.com	Sales Representative	(123)555-0100	(123)555-0102		(123)555-0103	123 4th Avenue
5	Northwind Traders	Thorpe	Steven	steven@northwindtraders.com	Sales Manager	(123)555-0100	(123)555-0102		(123)555-0103	123 5th Avenue
6	Northwind Traders	Nepper	Michael	michael@northwindtraders.com	Sales Representative	(123)555-0100	(123)555-0102		(123)555-0103	123 6th Avenue

FIGURE 11.16 Access records copied programmatically to Excel.

Before you can copy Access data to the Excel worksheet, you need to set references to the Workbook, Worksheet, and Range objects. Once these references are defined, the procedure uses the `Add` method to add a new Excel workbook and then activates the first worksheet. The Recordset fields' names are written as column names to the first worksheet row. Next, the reference is set to the Range object that will receive the data from the recordset. The `CopyFromRecordset` method is used to copy the specified number of records to the worksheet. Once data is placed in the worksheet, it is fit into the columns with the `AutoFit` property. The Excel worksheet is then saved in the file format compatible with Excel 97–2003. The Workbook's `SaveAs` method requires the `FileFormat` parameter that specifies the file format for the workbook. The following file formats are used in Excel:

- 50 (`xlExcel12`)—Excel binary workbook with or without macros (`.xlsb`)
- 52 (`xlOpenXMLWorkbookMacroEnabled`)—.xml file format with or without macros (`.xslm`)
- 51 (`xlOpenXMLWorkbook`)—.xml file format without macros (`.xlsx`)
- 56 (`xlExcel18`)—97–2003 format (`.xls`)

After saving the workbook, the procedure uses the `Workbook.Close` method to close the Excel file. The `Excel Application` object's `Quit` method is used to close the Excel application.

## Filtering Records Using the SQL WHERE Clause

When you want to work only with a certain subset of records, you can filter out those records you don't want to see by using the SQL `WHERE` clause or the `Filter` property. You can apply a filter to a Dynaset-type or Snapshot-type Recordset object. The fastest way to filter records is to open a new Recordset object by using an SQL statement that includes a `WHERE` clause. Hands-On 11.29 provides an example of using the SQL `WHERE` clause to retrieve product orders with an order quantity greater than 100.



### Hands-On 11.29 Filtering Records with the SQL WHERE Clause

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `FilterWithSQLWhere.DAO` procedure:

```
Sub FilterWithSQLWhere.DAO()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim qdf As DAO.QueryDef
    Dim qryName As String
    Dim mySQL As String
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\
    strDb = "Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(strPath & strDb)

    qryName = "qryOrdersOver100"
    mySQL = "SELECT * FROM " _
        & "[Product Orders] WHERE Quantity > 100;""
    Set qdf = db.CreateQueryDef(qryName)
    qdf.SQL = mySQL
    Set rst = db.OpenRecordset(qryName)
    Debug.Print "There are " & rst.RecordCount & _
    " orders with the order quantity greater than 100."

    rst.Close
    Set rst = Nothing
    Set qdf = Nothing
```

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

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

This procedure creates a simple Select query in the Northwind 2007\_Chap11.accdb database based on the Product Orders table. The SQL `WHERE` clause in the SQL statement specifies that only orders with a quantity greater than 100 should be returned. If the expression contained in the `WHERE` clause is `True`, then the record is selected; otherwise, the record is excluded from the opened set of records.

### Filtering Records Using the Filter Property

---

You can use the DAO Filter property to obtain a set of records that meet specific criteria.

Hands-On 11.30 uses the Filter property with the DAO Recordset to restrict the subset of records to those in which the employee's city begins with the letter "R."



#### Hands-On 11.30 Filtering Records Using the Filter Property

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **FilterRecords\_DAO** procedure shown here:

```
Sub FilterRecords_DAO()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim FilterRst As DAO.Recordset
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\
    strDb = "Northwind 2007_Chap11.accdb"

    Set db = OpenDatabase(strPath & strDb)
    Set rst = db.OpenRecordset("Employees", _
        dbOpenDynaset)
    rst.Filter = "City like 'R*'"
    Set FilterRst = rst.OpenRecordset()
```

```
Do Until FilterRst.EOF
    Debug.Print FilterRst.Fields -
        ("Last Name").Value
    FilterRst.MoveNext
Loop

FilterRst.Close
Set FilterRst = Nothing
rst.Close
Set rst = Nothing
db.Close
Set db = Nothing
End Sub
```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

This procedure begins by opening a Dynaset-type Recordset object based on the Employees table and setting the Filter property on this recordset:

```
rst.Filter = "City like 'R*' "
```

For the filter to take effect after you set it, you must open a new recordset based on the Recordset object to which the filter was applied:

```
Set FilterRst = rst.OpenRecordset()
```

Next, the procedure writes to the Immediate window the value of the Last Name field for all of the records in the filtered recordset.

## **CREATING AND RUNNING QUERIES**

---

Having worked with Access for a while, you already know that to retrieve relevant information from your database and perform data-oriented tasks, you need to write queries. *Queries* are SQL statements that are saved in the database and can be run at any time. Creating and executing queries are the most frequently performed database operations. Access 2021 supports several types of queries.

The simplest queries allow you to select a set of records from a table. However, when you need to extract information from more than one table at a time, you must write a more complex query by using a SQL `JOIN` statement. Other queries perform specific actions on existing data, such as creating a new table, appending rows to a table, updating the values in a table, or deleting rows from a table. Although Access provides a friendly interface—the Query Design view—for creating queries manually, the following sections teach you how to create

and execute the same queries by using DAO objects as well as SQL Data Manipulation Language (DML) statements in VBA code.

### **Creating a Select Query Manually**

Select queries retrieve a set of records from a database table. These queries are easily recognized by the `SELECT` and `FROM` keywords in their syntax. Let's take a look at some examples:

<code>SELECT LastName FROM Employees</code>	Selects the LastName field from the Employees table. If there is a space in the field name, enclose the field name in square brackets: [LastName].
<code>SELECT FirstName, LastName, PhoneNo FROM Employees</code>	Selects the FirstName, LastName, and PhoneNo fields from the Employees table.
<code>SELECT * FROM Employees</code>	Selects all fields for all records from the Employees table. The asterisk (*) is used to represent all fields.

Often, the `WHERE` clause is used with Select queries to specify criteria that determine which records the query will affect. Some examples of using the `WHERE` clause to restrict records are shown in the following table:

<code>SELECT * FROM Employees WHERE City IN ('Redmond', 'London')</code>	Selects from the Employees table all fields for all records that have the value Redmond or London in the City field.
<code>SELECT * FROM Employees WHERE City IN ('Redmond', 'London') AND ReportsTo LIKE 'Buchanan, Steven'</code>	Selects from the Employees table all fields for all records that have the value Redmond or London in the City field and have a value Buchanan, Steven in the ReportsTo field.
<code>SELECT * FROM Employees WHERE ((Year([HireDate])&lt;1993) OR (City='Redmond'))</code>	Selects from the Employees table all fields for all records that have a value less than 1993 in the HireDate field or have the value Redmond in the City field.
<code>SELECT * FROM Products WHERE UnitPrice BETWEEN 10 AND 25</code>	Selects from the Products table all fields for all records that have an amount in the UnitPrice field between \$10 and \$25.
<code>SELECT * FROM Employees WHERE ReportsTo IS NULL</code>	Selects from the Employees table all fields for all records that do not have a value in the ReportsTo field.

You can use expressions in WHERE clauses to qualify SQL statements. A SQL expression is a string that is used in SQL statements. Expressions can contain literal values, constants, field names, operators, and functions. Several operators that are often used in expressions are shown in Table 11.10.

TABLE 11.10 Operators commonly used in expressions

Operator Name	Description/Usage
IN	<p>The IN operator is used to determine whether the value of an expression is equal to any of several values in a specified list. If the expression is found in the list of values, the IN operator returns True; otherwise, it returns False. You can include the NOT logical operator to determine whether the expression is not in the list of values.</p> <p>For example, you can use NOT IN to determine which employees don't live in Redmond or London:</p> <pre>SELECT * FROM Employees WHERE City NOT IN ('Redmond', 'London')</pre>
LIKE	<p>The LIKE operator compares a string expression to a pattern in a SQL expression. For a pattern, you specify the complete value (for example, LIKE 'Buchanan, Steven'), or you can use wildcard characters to find a range of values (for example, LIKE 'B*'). You can use a number of wildcard characters in the LIKE operator pattern (see Table 15.2).</p>
BETWEEN...AND	<p>The BETWEEN...AND operator is used to determine whether the value of an expression falls within a specified range of values. If the value of the expression is between value1 and value2 (inclusive), the BETWEEN...AND operator returns True; otherwise, it returns False. You can include the NOT logical operator to evaluate the opposite condition, that is, whether the expression falls outside the range defined by value1 and value2.</p> <p>For example, you can select all products with the amount in the UnitPrice field less than \$10 and greater than \$25:</p> <pre>SELECT * FROM Products WHERE UnitPrice NOT BETWEEN 10 AND 25</pre>
IS NULL	<p>The IS NULL operator is used to determine whether the expression value is equal to the Null value. A Null value indicates missing or unknown data. You can include the NOT logical operator to return only records that have values in the specified field.</p> <p>For example, you can extract only the employee records that have a value in the ReportsTo field. Records where the ReportsTo field is blank will not be included:</p> <pre>SELECT * FROM Employees WHERE ReportsTo IS NOT NULL</pre>

**TABLE 11.11** Wildcard characters used in the LIKE operator patterns

Wildcard	Description
* (asterisk)	Matches any number of characters.
? (question mark)	Matches any single character.
% (percent sign)	Matches any number of characters (used only with the ADO and Jet OLE DB Provider; not in the Access user interface).
_ (underscore)	Matches any single character (used only with the ADO and Jet OLE DB Provider; not in the Access user interface).
# (number sign)	Matches any single digit.
[] (square brackets)	Matches any single character within the list of characters enclosed in brackets.
! (exclamation point)	Matches any single character that is not found in the list enclosed in the square brackets.
- (hyphen)	Matches any one of the range of characters enclosed in the square brackets.

In addition to the `WHERE` clause, you can use predicates to further restrict the set of records to be retrieved. A *predicate* is a SQL statement that qualifies the `SELECT` statement, similar to the `WHERE` clause; however, the predicate must be placed before the column list. Several popular predicates are shown in Table 11.12.

**TABLE 11.12** Commonly used predicates in SQL SELECT statements

Predicate Name	Description/Usage
ALL	<p>The ALL keyword is the default keyword and is used when no predicate is declared in the SQL statement.</p> <p>The following two examples are equivalent and return all records from the Employees table:</p> <pre>SELECT ALL * FROM Employees ORDER BY EmployeeID; SELECT * FROM Employees ORDER BY EmployeeID</pre>
DISTINCT	<p>The DISTINCT keyword eliminates duplicate values from the returned set of records. The values for each field listed in the <code>SELECT</code> statement must be unique.</p> <p>For example, to return a list of nonduplicate (unique) cities from the Employees table, you can write the following <code>SELECT</code> statement:</p> <pre>SELECT DISTINCT City FROM Employees</pre> <p><b>Note:</b> The output of a query that uses <code>DISTINCT</code> isn't updatable (it's read-only).</p>

Predicate Name	Description/Usage
DISTINCTROW	<p>While the DISTINCT keyword is based on duplicate fields, the DISTINCTROW keyword is based on entire rows. It is used only with multiple tables. For example, if you join the Customers and Orders tables on the CustomerID field, you can find customers that have at least one order. The Customers table contains no duplicate CustomerID fields, but the Orders table does because each customer can have many orders.</p> <pre>SELECT DISTINCTROW CompanyName FROM Customers, Orders WHERE Customers.CustomerID = Orders.CustomerID ORDER BY CompanyName;</pre> <p>Note: If you omit DISTINCTROW, this SELECT statement will produce multiple rows for each company that has more than one order. DISTINCTROW has an effect only when you select fields from some, but not all, of the tables used in the query. DISTINCTROW is ignored if your query includes only one table or if you output fields from all tables.</p>
TOP or PERCENT	<p>The TOP keyword returns a certain number of records that fall at the top or bottom of a range specified by an ORDER BY clause.</p> <p>For example, suppose you want to select the five most expensive products:</p> <pre>SELECT TOP 5 * FROM Products ORDER BY UnitPrice DESC</pre> <p>The TOP predicate doesn't choose between equal values. If there are equal values present, the TOP keyword will return all rows that have the equal value.</p> <p>You can also use the PERCENT keyword to return a percentage of records that fall at the top or bottom of a range specified by an ORDER BY clause.</p> <p>For example, to return the lowest 10 percent priced products, you can write the following statement:</p> <pre>SELECT TOP 10 PERCENT * FROM Products ORDER BY UnitPrice ASC;</pre> <p>Note: If you don't include the ORDER BY clause, the SELECT TOP statement will return a random set of rows.</p>

If you'd like to sort records returned by the SELECT statement, use the ORDER BY clause with the ASC (ascending sort) or DESC (descending sort) keywords, as shown in the following example:

SELECT * FROM Employees ORDER BY Country DESC	Select all records from the Employees table and arrange them in descending order based on the Country field. If no order is specified, the order is ascending (ASC) by default.
---	---

By default, records are sorted in ascending order. The fields you want to sort by do not need to be enumerated in the `SELECT` statement's field list. Instead of sorting by field name, you can sort by field position. For example, the statement,

```
SELECT * FROM EMPLOYEES ORDER BY 2
```

will sort the records in ascending order by the second field.

### **Creating a Select Query with DAO**

---

In DAO, the `QueryDef` object represents a saved query in a database. All `QueryDef` objects are contained in the `QueryDefs` collection. You can read and set the SQL definition of a `Query` object using the `SQL` property. To create a query in code, use the `CreateQueryDef` method. For example, to create a Select query named `myQuery`, the following statement is used:

```
Set qdf = db.CreateQueryDef("myQuery", strSQL)
```

When you specify the name for your query, the new `QueryDef` object is automatically appended to the `QueryDefs` collection when it is created. The second argument of the `CreateQueryDef` method is a string variable that holds a valid Access SQL statement. Prior to using this variable, you must assign to it a string expression:

```
strSQL = "SELECT * FROM Employees WHERE TitleOfCourtesy = 'Ms.'"
```

The `WHERE` clause is used with Select queries to specify criteria that determine which records the query will affect.

The procedure in Hands-On 11.31 selects from the `Employees` table all records that have a value of “Ms.” in the `TitleOfCourtesy` field. The keyword `LIKE` can be substituted for the equals sign (`=`), as in the following:

```
strSQL = "SELECT * FROM Employees WHERE TitleOfCourtesy LIKE 'Ms.'"
```

When creating queries in code, be sure to include an error handler. After all, the query you are trying to create may already exist, or an unexpected error could occur.



#### **Hands-On 11.31 Creating a Select Query**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the `Create_SelectQuery.DAO` procedure shown here:

```
Sub Create_SelectQuery.DAO()
```

```
Dim db As DAO.Database
Dim qdf As DAO.QueryDef
Dim strSQL As String
Dim strDb As String

strDb = "C:\VBAAccess2021_ByExample\Northwind_Chap11.mdb"

On Error GoTo Err_SelectQuery

strSQL = "SELECT * FROM Employees "
strSQL = strSQL & "WHERE TitleOfCourtesy = 'Ms.'"
Set db = OpenDatabase(strDb)
Set qdf = db.CreateQueryDef("myQuery", strSQL)
ExitHere:
Set qdf = Nothing
db.Close
Set db = Nothing
Exit Sub

Err_SelectQuery:
If Err.Number = 3012 Then
    MsgBox "Query with this name already exists."
Else
    MsgBox Err.Description
End If
Resume ExitHere
End Sub
```

**3.** Choose **Run | Run Sub/UserForm** to execute the procedure.

When you run the Create\_SelectQuery.DAO procedure, the next time you open the Northwind\_Chap11.mdb database you should see the query named myQuery in the list of stored queries in the Access window.

**NOTE**

*Instead of a query that is saved in the database for future use, it is possible to create a temporary query by setting the QueryDef.Name property to a zero-length string (""), as in the following example:*

```
Set qdf = db.CreateQueryDef("", strSQL)
```

*The advantage of temporary queries is that they don't clutter the Access Application window.*

## Creating and Running a Parameter Query

A special type of a Select query is known as a *Parameter* query. Instead of retrieving the same records each time a query is run, a user can enter the search criteria in a special dialog box at runtime. In DAO, the parameters of a Parameter query are represented by Parameter objects. The QueryDef object contains a Parameters collection. Parameter objects represent existing parameters.

To create a Parameter query, create a query string that includes the PARAMETERS keyword:

```
strSQL = "PARAMETERS [Enter Country] Text;" & _  
    "SELECT * FROM CUSTOMERS WHERE Country = [Enter Country];"
```

Before executing an existing Parameter query, assign a value to the parameter, as shown in Hands-On 11.32. Once the parameter value is specified, you need to open a recordset based on the query.

The next Hands-On demonstrates how to create and run a Parameter query to retrieve the names of the companies in the user-specified country.



### Hands-On 11.32 Creating a Parameter Query

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **CreateRun\_ParameterQuery\_DAO** procedure:

```
Sub CreateRun_ParameterQuery_DAO()  
    Dim db As DAO.Database  
    Dim qdf As DAO.QueryDef  
    Dim rst As DAO.Recordset  
    Dim strQryName As String  
    Dim strSQL As String  
    Dim strDb As String  
    Dim strPath As String  
  
    strPath = "C:\VBAAccess2021_ByExample\"  
    strDb = "Northwind_Chap11.mdb"  
  
    On Error GoTo Err_Handler  
  
    strQryName = "myParamQuery"  
    strSQL = "PARAMETERS [Enter Country] Text;" & _  
        "SELECT * FROM Customers WHERE " & _  
        "Country = [Enter Country];"  
  
    Set db = OpenDatabase(strPath & strDb)
```

```
Set qdf = db.CreateQueryDef(strQryName, strSQL)

RunQuery:
    ' specify the parameter
    qdf.Parameters("Enter Country") =
        InputBox("Enter the country name:", _
        "Which Country?", "Germany")

    If IsNull(qdf.Parameters("Enter Country").Value) _
        Then GoTo ExitHere

    ' open a recordset based on the specified query
    Set rst = qdf.OpenRecordset(dbOpenDynaset)
    rst.MoveLast
    MsgBox "Number of records: " & rst.RecordCount

    ' write the contents of the second field
    ' to the Immediate window
    rst.MoveFirst
    Do Until rst.EOF
        Debug.Print rst(1)
        rst.MoveNext
    Loop

ExitHere:
    If Not rst Is Nothing Then
        rst.Close
        Set rst = Nothing
    End If
    Set qdf = Nothing
    db.Close
    Set db = Nothing
    Exit Sub

Err_Handler:
    If Err.Number = 3012 Then
        MsgBox "This query already exists."
        Set qdf = db.QueryDefs(strQryName)
        Resume RunQuery
    Else
        MsgBox Err.Description
    End If
    Resume ExitHere
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

This procedure defines a Parameter query that contains one parameter named `Enter Country`. Prior to running this query, the procedure retrieves the name of the country from the user via the VBA `InputBox` method. While the suggested default country name is Germany, the user can supply the name of another country. The supplied value is then used as the value of the `Enter Country` parameter. Next, the recordset is opened based on the specified query, and the number of records for the specified country is retrieved via the `RecordCount` property of the `Recordset` object. In order to get the correct record count, we must move to the end of the recordset, using the `MoveLast` method, to access all records. The procedure ends by retrieving to the Immediate window the names of all the companies in the specified country. The procedure contains several labels such as `RunQuery`, `ExitHere`, and `Err_Handler`, which are used in error trapping and ensuring that certain code lines are run only when required. For example, when you execute this procedure again, the statement that attempts to create a query will fail and VBA will generate error 3012. At this point, we want to run the existing query, so we must set the `qdf` object variable with the following statement:

```
Set qdf = db.QueryDefs(strQryName)
```

And then we can safely resume running the code from the label `RunQuery`.

### **Creating and Running a Make-Table Query**

A Make-Table query creates a new table out of records from one or more tables or queries. Make-Table queries are often used to preserve data as it existed at a particular time or to create a backup copy of a table without backing up the entire database. Use the `SELECT INTO` statement to create a Make-Table query. This statement consists of the following parts:

<code>SELECT fieldname</code>	Field name (use * for all fields)
<code>INTO newTableName</code>	Name of the new table
<code>FROM table/queryName</code>	Name of a table or query from which data is taken
<code>WHERE condition</code>	Criteria/limit operation to desired rows (optional)
<code>ORDER BY fieldname</code>	Order of the records in the new table (optional)

The procedure in Hands-On 11.33 creates a table of the customers in Brazil.



### Hands-On 11.33 Creating and Running a Make-Table Query

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **MakeATableQuery.DAO** procedure:

```
Sub MakeATableQuery_DAO()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim strSQL As String
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\"

    strDb = "Northwind_Chap11.mdb"

    On Error GoTo Err_Handler

    strSQL = "SELECT * INTO SouthAmericanClients" & _
        " FROM Customers WHERE Country='Brazil';"
    Set db = OpenDatabase(strPath & strDb)
    Set qdf = db.CreateQueryDef("", strSQL)
    qdf.Execute

    ExitHere:
    Set qdf = Nothing
    db.Close
    Set db = Nothing
    Exit Sub

Err_Handler:
    MsgBox Err.Description
    Resume ExitHere
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

The `SELECT INTO` statement in the `MakeATableQuery.DAO` procedure is used to make a new table named `SouthAmericanClients` containing the names of all Brazilian customers from the `Customers` table in the `Northwind.mdb` database. Notice that by not assigning a name to the query, we create a Make-Table query that is temporary (not stored in the Access window):

```
Set qdf = db.CreateQueryDef("", strSQL)
```

## **Creating and Running an Update Query**

An Update query is a type of Action query. Update queries are very convenient to use when you want to change fields for a single record or for multiple records in a table. The `UPDATE` statement consists of the following three parts:

UPDATE	TableName or QueryName
SET	Expression/operation to perform
WHERE	Criteria/limit operation to desired rows

For example, to mark product 10 as discontinued, you would use the following `UPDATE` statement:

```
UPDATE Products SET Discontinued = True WHERE ProductID = 10
```

The condition in the `WHERE` clause is used to determine which rows will be updated. The Update query does not produce a result table. To avoid updating the wrong records, always determine which rows you want to be updated by creating and running a Select query first.

The `Execute` method of a `QueryDef` object is used to run any type of Action query. The procedure in Hands-On 11.34 demonstrates how to create and run an Update query with DAO.



### **Hands-On 11.34 Creating and Running an Update Query**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `CreateRunUpdateQuery_DAO` procedure:

```
Sub CreateRunUpdateQuery_DAO()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim strSQL As String
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\
    strDb = "Northwind_Chap11.mdb"

    On Error GoTo Err_Handler

    strSQL = "UPDATE Suppliers " & _
    "INNER JOIN Products ON " & _
    "Suppliers.SupplierID = Products.SupplierID " & _
```

```

"SET Products.UnitPrice = [UnitPrice]+2 " &
"WHERE (((Suppliers.CompanyName)='Tokyo Traders'));"

Set db = OpenDatabase(strPath & strDb)

Set qdf = db.CreateQueryDef("PriceIncrease", strSQL)
qdf.Execute
ExitHere:
Set db = Nothing
Exit Sub
Err_Handler:
If Err.Number = 3012 Then
    MsgBox "Query with this name already exists."
Else
    MsgBox Err.Description
End If
Resume ExitHere
End Sub

```

To perform the required update, this procedure needs to join two tables. The `Products` table is joined with the `Suppliers` table on the `SupplierID` field that exists in both tables. Use the `INNER JOIN` statement to combine column values from one row of a table with column values from another row of another (or the same) table to obtain a single row of data. The join condition is specified after the `ON` keyword and determines how the two tables are to be compared to each other to produce the join result. Because the update must occur only for a specific supplier, we also specify the supplier's company name in the `WHERE` clause.

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

After running this procedure, the prices for all products supplied by Tokyo Traders are increased by \$2.00.

The following procedure demonstrates how to use the `Execute` method of the DAO Database object to run an existing (previously saved) Update query.

```

Sub UpdateRun.DAO()
Dim db As DAO.Database
Dim strDb As String

strDb = "C:\VBAAccess2021_ByExample\Northwind_Chap11.mdb"

Set db = OpenDatabase(strDb)
db.Execute "PriceIncrease"
db.Close

```

```
Set db = Nothing
End Sub
```

## Running an Append Query

Append queries are used for adding records from one or more tables to other tables. You can append records to a table in a current database or another Access or non-Access database. An Append query is an Action query that adds new records to the end of an existing table or query. Append queries don't return records. They are useful for archiving records. Before you can archive the records, you need to create a new table structure to hold the records. To add a record or multiple records to a table, use the `INSERT INTO` statement. This statement has the following parts:

<code>INSERT INTO target [(Field1, Field2)]</code>	The name of the table or query to which records are appended. You may indicate the names of the fields to which data is appended.
<code>SELECT fieldName(s)</code>	The names of fields from which data is obtained.
<code>FROM tableName or expression</code>	The name of the table or tables from which records are inserted, the name of a saved query, or a <code>SELECT</code> statement.
<code>WHERE condition</code>	Criteria/limit operation to desired rows.

The procedure in Hands-On 11.35 demonstrates how to execute an Append query using the `Execute` method of the DAO Database object.



### Hands-On 11.35 Running an Append Query

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `RunAppendQry.DAO` procedure:

```
Sub RunAppendQry.DAO()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strSQL As String
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\"
    strDb = "Northwind_Chap11.mdb"

    strSQL = "SELECT * FROM " & _
        "SouthAmericanClients" & _
```

```

"WHERE Country = 'Argentina'"

Set db = OpenDatabase(strPath & strDb)
Set rst = db.OpenRecordset(strSQL, _
    dbOpenSnapshot)

If rst.EOF Or rst.BOF Then
    ' Argentina clients not found in
    ' destination table - proceed with insert
    db.Execute "INSERT INTO " & _
        "SouthAmericanClients " & _
        "SELECT * FROM Customers " & _
        "WHERE Country = 'Argentina'"
    MsgBox "Argentina clients have been appended."
Else
    MsgBox "Clients from Argentina already " & _
        "exist in the destination table."
End If

rst.Close
Set rst = Nothing
db.Close
Set db = Nothing
End Sub

```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

This procedure begins by opening the Northwind\_Chap11.mdb database and creating a Snapshot-type recordset based on the supplied SQL query string. Prior to executing the Append query that inserts customers from Argentina into the SouthAmericanClients table, we check the EOF and BOF properties of the DAO Recordset object to determine if the recordset contains any records. If `rst.EOF Or rst.BOF` is True, then there is no current record (the recordset is empty), so we use the `Execute` method of the database object to add Argentina customers to the destination table.

---

### Running a Delete Query

With a Delete query, you can delete a single record or multiple records from a database. The `DELETE` statement used to delete rows from a table consists of the following three parts:

DELETE	
FROM	Table name
WHERE	Criteria/limit operation to desired rows

For example, to delete discontinued products from the Products table, you would use the following `DELETE` statement:

```
DELETE FROM Products WHERE Discontinued = True
```

To delete all the rows from the Products table, the following statement can be executed:

```
DELETE FROM Products
```

You cannot reverse the operation performed by the `DELETE` statement. Always make a backup copy of your table prior to running a Delete query. It is a good idea to create and run a Select query before using `DELETE` to see which rows will be affected by the Delete operation.

The `Execute` method is used to run Action queries or execute an SQL statement. This method can take optional arguments. For example, in the statement,

```
qdf.Execute dbFailOnError
```

the constant `dbFailOnError` will generate a runtime error and will roll back updates or deletes if an error occurs. Use the `RecordsAffected` property of the `QueryDef` object to determine the number of records affected by the most recent `Execute` method. For example, the following statement displays the number of records that were deleted:

```
MsgBox qdf.RecordsAffected & " records were deleted."
```

The procedure in Hands-On 11.36 creates a Delete query and then executes it if the user responds positively to the message shown in Figure 11.17.

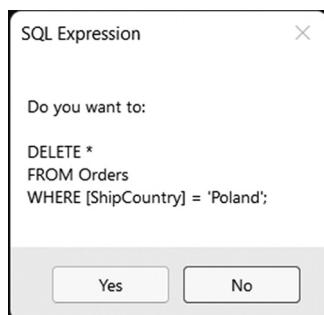


FIGURE 11.17 You can display an SQL statement underlying a query in a message box.



### Hands-On 11.36 Running a Delete Query

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **CreateRunDeleteQuery\_DAO** procedure:

```
Sub CreateRunDeleteQuery_DAO()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim strQryName As String
    Dim strSQL As String
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\"

    strDb = "Northwind_Chap11.mdb"

    On Error GoTo ErrorHandler

    strQryName = "DeletePolishOrders"
    strSQL = "DELETE * FROM Orders "
    strSQL = strSQL & "WHERE [ShipCountry] = 'Poland'"

    Set db = OpenDatabase(strPath & strDb)
    Set qdf = db.CreateQueryDef(strQryName, strSQL)

    ' Chr(13) & Chr(13) is a double carriage return
    If (MsgBox("Do you want to: " & _
        Chr(13) & Chr(13) & _
        & qdf.SQL, vbYesNo + vbDefaultButton2, &_
        "SQL Expression")) = vbYes Then

        qdf.Execute dbFailOnError
        MsgBox qdf.RecordsAffected & _
            " records were deleted."
    End If
    ExitHere:
    Set qdf = Nothing
    db.Close
    Set db = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = 3012 Then
        Set qdf = db.QueryDefs(strQryName)
        Resume Next
    End If
End Sub
```

```
Else
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End If
End Sub
```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

This procedure creates a Delete query named DeletePolishOrders in the Northwind database, then runs this query when the user clicks OK to the message. If the specified Delete query already exists in the database, the `qdf` object variable is set to the existing query name and the user is prompted to proceed or cancel the operation.

## Creating and Running a Pass-Through Query

A *Pass-Through query* works directly with an external ODBC (Open Database Connectivity) data source. Instead of linking to a table that resides on a server, you can send commands directly to the server to retrieve data.

To create a Pass-Through query manually in the Access window, choose Create | Query Design. Close the Show Table dialog box and click Design | Pass-Through. This will bring up a window where you can type a query statement. The SQL statement must be in the format understood by the external data source from which you are retrieving data. Pass-Through queries can also be used in lieu of Action queries when you need to bulk append, update, or delete data in remote databases.

Pass-Through queries can be created and executed programmatically from your VBA procedures. In DAO, use the Connect property to execute an SQL Pass-Through query. If you do not specify a connection string in the Connect property, Access will ask you for the connection information every time you run the Pass-Through query (and this can be very annoying).

The following procedure uses the MaxRecords property to return 15 records from the dbo.entity table located on a SQL server. Notice that the ReturnsRecords property is set to `True`. If your query does not need to return records, set the ReturnsRecords property to `False`.

```
Sub PassThruQry.DAO()
    Dim db As DAO.Database
    Dim qdfPass As DAO.QueryDef

    On Error GoTo err_PassThru

    Set db = CurrentDb
```

```
Set qdfPass = db.CreateQueryDef("GetRecords")

' enter your own connect string
' supply the server database name you
' want to connect to, your User ID,
' password, and the Data Source name

qdfPass.Connect =
    "ODBC;Database=myDbName; " &
    "UID=JKO;PWD=tester;DSN=myDataS"
qdfPass.SQL = "SELECT * FROM dbo.entity"
qdfPass.ReturnsRecords = True
qdfPass.MaxRecords = 15

DoCmd.OpenQuery "GetRecords"
Exit Sub
err_PassThru:
If Err.Number = 3151 Then
    MsgBox Err.Description
    Exit Sub
End If
db.QueryDefs.Delete "GetRecords"
Resume 0
Exit Sub
End Sub
```

Instead of displaying a datasheet with the records retrieved from the SQL database, the following procedure reads the records to a temporary query and proceeds to open a recordset based on that query. Next, the contents of two fields are printed to the Immediate window.

```
Sub PassThru2()
    Dim db As DAO.Database
    Dim qdfPass As DAO.QueryDef
    Dim rstTemp As DAO.Recordset

    On Error GoTo err_PassThru

    Set db = CurrentDb
    Set qdfPass = db.CreateQueryDef("")

    ' enter your own connection string
    ' supply the server database name you
    ' want to connect to, your User ID,
    ' password, and the Data Source name
```

```
qdfPass.Connect = _  
    "ODBC;Database=myDbName;UID=JKO;" & _  
    "PWD=tester;DSN=myDataS"  
qdfPass.SQL = "SELECT * FROM dbo.entity"  
qdfPass.ReturnsRecords = True  
qdfPass.MaxRecords = 15  
Set rstTemp = qdfPass.OpenRecordset()  
' print data to the Immediate window  
With rstTemp  
    Do While Not .EOF  
        Debug.Print .Fields("entity_id"), _  
            .Fields("entity_name")  
        .MoveNext  
    Loop  
    .Close  
End With  
SendKeys "^g"  
ExitHere:  
    Set db = Nothing  
    Exit Sub  
err_PassThru:  
    MsgBox Err.Number & ":" & Err.Description  
    Resume ExitHere  
End Sub
```

## PERFORMING OTHER OPERATIONS WITH QUERIES

---

Now that you know how to programmatically create and run various queries using DAO objects, you may be interested to find out how to use Visual Basic to perform other operations related to queries, such as retrieving a list of queries and their properties, deleting a query, and determining if a query is updatable.

### Retrieving Query Properties with DAO

---

Just like tables and other database objects, queries have properties. To generate a list of properties for a specific query, use the `For Each...Next` looping structure to iterate through the `Properties` collection of the DAO `QueryDef` object. The procedure in Hands-On 11.37 demonstrates this.



#### Hands-On 11.37 Listing Query Properties

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the `List_QryProperties.DAO` procedure shown here:

```
Sub List_QryProperties_DAO()
    Dim db As DAO.Database
    Dim prp As DAO.Property
    Dim strDBName As String
    Dim strPath As String

    On Error Resume Next

    strPath = "C:\VBAAccess2021_ByExample\"
    strDBName = "Northwind 2007_Chap11.accdb"
    Set db = OpenDatabase(strPath & strDBName)
    For Each prp In db.QueryDefs _
        ("Invoice Data").Properties
        Debug.Print prp.Name & "=" & prp.Value
    Next prp
    Set db = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.
4. Activate the Immediate Window to view the procedure output.

### **Listing All Queries in a Database with DAO**

---

You can obtain the listing of all queries in a database by using the `For...Each` loop to enumerate the `QueryDefs` collection of the DAO `QueryDef` object. The following procedure writes to the Immediate window the names of all queries in the `Northwind 2007_Chap11.accdb` database.

```
Sub List_AllQueries_DAO()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim strDb As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\"
    strDb = "Northwind 2007_Chap11.accdb"
    Set db = OpenDatabase(strPath & strDb)
    For Each qdf In db.QueryDefs
        Debug.Print qdf.Name
    Next qdf

    Set qdf = Nothing
    db.Close
    Set db = Nothing
End Sub
```

## Deleting a Query from a Database

To remove a DAO `QueryDef` object from a `QueryDefs` collection, use the `Delete` method as shown in Hands-On 11.38. The `DeleteAQuery.DAO` procedure deletes the query that was created in Hands-On 11.31.



### Hands-On 11.38 Deleting a Query from a Database

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `DeleteAQuery.DAO` procedure:

```
Sub DeleteAQuery.DAO()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim strDb As String
    Dim strPath As String

    On Error GoTo ErrorHandler
    strPath = "C:\VBAAccess2021_ByExample\
    strDb = "Northwind_Chap11.mdb"
    Set db = OpenDatabase(strPath & strDb)
    db.QueryDefs.Delete "myQuery"

    ExitHere:
    db.Close
    Set db = Nothing
    Exit Sub

ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.  
After running the procedure in Hands-On 11.37, the query named `myQuery` is removed from the `Northwind_Chap11.mdb` database.

## Determining If a Query Is Updatable

When a query is updatable, you may edit the values in the result set of records and your changes are automatically reflected in the underlying tables. Access's online help lists situations in which query results can or cannot be updated (see Figure 11.18). The DAO `QueryDef` object has an `Updatable` property that you can use in your VBA code to find out if the query definition can be updated. However, to determine whether the resulting recordset can be updated, you

must use the Updatable property of the DAO Recordset object, as demonstrated in Hands-On 11.38. If the Recordset object cannot be edited, the value of the Updatable property is False.

<b>NOTE</b>	<i>The Updatable property of the DAO Snapshot-type and Forward-only-type Recordset objects is always False. The same is true if the Recordset object contains read-only fields. However, when one or more fields are updatable, the property's value is True. Because a recordset can contain fields that can't be updated, you may want to check the DataUpdatable property of each field in the Fields collection of the Recordset object before attempting to edit a record.</i>
-------------	---

#### Restrictions on fields that can be updated

An update query cannot be used to update data in the following types of fields:

- **Calculated fields** The values in calculated fields do not permanently reside in tables. They only exist in your computer's temporary memory after Access calculates them. Because calculated fields do not have a permanent storage location, you cannot update them.
- **Fields from a totals query or a crosstab query** The values in these types of query are calculated, and therefore cannot be updated by an update query.
- **AutoNumber fields** By design, the values in AutoNumber fields change only when you add a record to a table.
- **Fields in unique-values queries and unique-records queries** The values in such queries are summarized. Some of the values represent a single record, and others represent more than one record. The update operation is not possible because it is not possible to determine what records were excluded as duplicates, and therefore not possible to update all the necessary records. This restriction applies whether you use an update query or try to update data manually by entering values in a form or a datasheet.
- **Fields in a union query** You cannot update data from fields in a union query because each record that appears in two or more data sources only appears once in the union query result. Because some duplicate records are removed from the results, Access cannot update all the necessary records.
- **Fields that are primary keys** In some cases, such as if the primary key field is used in a table relationship, you cannot update the field by using a query unless you first set the relationship to automatically cascade updates.

**Note:** When you cascade updates, Access automatically updates foreign key values when you change a primary key value in a parent table.

**FIGURE 11.18** Records returned by a query may or may not be updatable.

For details, please see <http://office.microsoft.com/en-us/access-help/update-data-by-using-a-query-HA010076527.aspx>.

The procedure in Hands-On 11.39 checks whether records returned by two queries in the Northwind\_Chap11.mdb database can be edited.



### Hands-On 11.39 Determining if Records Returned by a Query Can Be Edited

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **IsQryUpdatable.DAO** procedure:

```
Sub IsQryUpdatable.DAO()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim fld As DAO.Field
    Dim strDb As String
    Dim strQryName1 As String
    Dim strQryName2 As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\
    strDb = "Northwind_Chap11.mdb"
    strQryName1 = "Order Subtotals"
    strQryName2 = "Invoices"

    Set db = OpenDatabase(strPath & strDb)

    Set rst = db.OpenRecordset(strQryName1)
    Debug.Print strQryName1 &
        ": Updatable=" & rst.Updatable
    Set rst = db.OpenRecordset(strQryName2)
    Debug.Print strQryName2 &
        ": Updatable=" & rst.Updatable
    For Each fld In rst.Fields
        If Not fld.DataUpdatable Then
            Debug.Print fld.Name & " cannot be edited."
        End If
    Next

    rst.Close
    Set rst = Nothing
    db.Close
    Set db = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

When you run this procedure, the `Updatable` property returns `True` for the `Invoices` query and `False` for the `Order Subtotals` query. The `OpenRecordset` method is used to open each of these queries. The `Order Subtotals` query is not updatable because its SQL statement contains a `GROUP BY` clause. While the `Invoices` query is updatable, not all fields in the resulting recordset can be edited (see Figure 11.19).

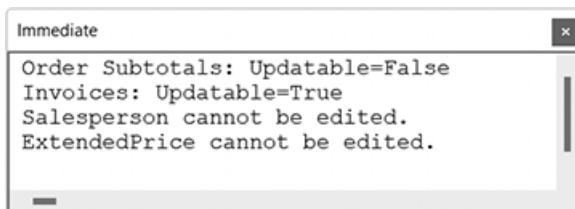


FIGURE 11.19 An updatable query can contain one or more fields that cannot be edited (see Hands-On 11.39).

## TRANSACTION PROCESSING

---

To improve your application's performance and to ensure that database activities can be recovered in case an unexpected hardware or software error occurs, consider grouping sets of database activities into a transaction. A *transaction* is a set of operations that are performed together as a single unit. If you use an automatic teller machine (ATM), you are already familiar with transaction processing. When you go to the bank to get cash, your account must be debited. In other words, the cash withdrawal must be deducted from your savings or checking account. A transaction is a two-sided operation. If anything goes wrong during the transaction, the entire transaction is canceled. If both operations succeed, that is, you get the cash and the bank debits your account, the transaction's work is saved (or committed).

Database transactions often involve modifications and additions of one or more records in a single table or in several tables. When a transaction has to be undone or canceled, the transaction is rolled back. Often, when you perform batch updates to database tables and an error occurs, updates to all tables must be canceled or the database could be left in an inconsistent state, resulting not only in loss of important information but also in a number of other headaches.

Transactions are extremely important for maintaining data integrity and consistency.

To work with transaction processing in DAO, use the transaction methods of the Workspace or DBEngine object: `BeginTrans`, `CommitTrans`, and `Rollback`. Within a Workspace transaction, you can perform operations on more than one connection or database.

### **Creating a Transaction**

---

The DAO Object Model supports transactions through the `BeginTrans`, `CommitTrans`, and `Rollback` methods of the Workspace and DBEngine objects. When you use these methods with the DBEngine object, the transaction is applied to the default workspace—`DBEngine.Workspaces(0)`. If you need to manage transactions or connections to multiple databases, use the Workspace object. A Workspace object represents a user's session. A transaction on a workspace will affect all data modifications made within the workspace. You can manage transactions independently across Database objects by creating additional Workspace objects.

Use the `BeginTrans` method to specify the beginning of a transaction, the `CommitTrans` method to save the changes, and `Rollback` to cancel the transaction. `BeginTrans` and `CommitTrans` are used in pairs. The data-modifying instructions you place between these keywords are stored in memory until VBA encounters the `CommitTrans` statement. After reaching `CommitTrans`, Access writes to the disk the changes that have occurred since the `BeginTrans` statement; therefore, any changes you've made in the tables become permanent.

If an error is generated during the transaction process, the `Rollback` statement placed further down in your procedure will undo all changes made since the `BeginTrans` statement, which ensures that the data is returned to the state it was in before you started the transaction.

Transaction processing should be used for archiving historical data. For instance, the procedure in Hands-On 11.40 selects all orders placed in 1997 and appends them to an archive table in another database (`Chap10.accdb`). Then the records are deleted from the source table in the current database (`Chap11.accdb`).

#### **Hands-On 11.40 Using a Database Transaction to Archive Records**

1. In the main Access window of the `Chap11.accdb` database, choose **External Data** | **New Data Source** | **From Database** | **Access**. In the File name box of the Get External Data dialog box, enter `C:\VBAAccess2021_ByExample\Northwind.mdb`, and then click **OK**. In the Import Objects window, select the

**Orders** table and click **OK**. Click **Close** to exit the Get External Data dialog box.

2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, enter the **OrdersArchive1997.DAO** procedure shown here:

```
Sub OrdersArchive1997.DAO()
    Dim db As DAO.Database
    Dim wrk As Workspace
    Dim blnTrans As Boolean
    Dim strSQL As String
    Dim strPath As String
    Dim strDb As String
    Dim strDateCriteria As String

    On Error GoTo ErrorHandler

    strPath = "C:\VBAAccess2021_ByExample\""
    strDb = "Chap10.accdb"
    strDateCriteria =
        "BETWEEN #1/1/1997# AND #12/31/1997#;""

    Set wrk = DBEngine(0)
    Set db = CurrentDb()

    'begin transaction

    blnTrans = True
    wrk.BeginTrans

    ' create an archive table on the fly
    ' and fill it with records

    strSQL = _
        "SELECT * INTO OrdersArchive1997 IN " & _
        Chr(34) & strPath & strDb & Chr(34) & _
        " FROM Orders WHERE Orders.OrderDate " & _
        strDateCriteria

    db.Execute strSQL, dbFailOnError

    ' delete records from the source table
    If db.RecordsAffected <> 0 Then
        strSQL = "DELETE FROM Orders " & _
            "WHERE Orders.OrderDate " & _
```

```

strDateCriteria

db.Execute strSQL, dbFailOnError

' ask user if OK to commit changes
If MsgBox("Click OK if you want to archive " _
& db.RecordsAffected & _
" records.", vbOKCancel + _
vbQuestion + vbDefaultButton2, _
"Proceed?") = vbOK Then
    DBEngine.CommitTrans
Else
    If blnTrans Then DBEngine.Rollback
End If
Else
    DBEngine.Rollback
    MsgBox "No records to archive " & _
    "with the specified criteria.", _
    vbInformation + vbOKOnly, _
    "Records not found"
End If
Cleanup:
Set db = Nothing
Exit Sub
ErrorHandler:
If Err.Number = 3010 Then
    ' hardcoding path and filename for
    ' demonstration only
    strSQL = "INSERT INTO OrdersArchive1997 IN " & _
    """C:\VBAAccess2021_ByExample\Chap10.accdb"" & _
    " SELECT * FROM Orders WHERE Orders.OrderDate " & _
    strDateCriteria
    Resume 0
Else
    If blnTrans Then DBEngine.Rollback
    MsgBox Err.Description
    Resume Cleanup
End If
End Sub

```

**4. Choose Run | Run Sub/UserForm** to execute the procedure.

Because transactions exist in a Workspace object, we define an object variable `wrk` and we set its value to `DBEngine(0)`, which stands for the default workspace. Before we begin the transaction with the `BeginTrans` method of the Workspace object we set the transaction flag to `True` (`blnTrans`) to indicate that the

transaction is active. We also set the Database object variable (`db`) to point to the current database. The first data operation in this transaction requires that we create a table in another Access database to store the selected records from the Orders table in the current database. In the Access user interface, we would simply create a Make-Table query; in VBA programming, we can use the SQL `SELECT...INTO` statement. The first part of this statement specifies the fields we want to select; in this case we use a wildcard (\*) to denote that all fields should be copied into the new table. This is followed by the `INTO` clause and the name of the table to be created. The path and database name must be surrounded by the quotation marks. You can use the `Chr(34)` function to prepend and append double quotes to the strings.

If the table already exists, then the `SELECT...INTO` statement will fail and VBA will respond with error 3010. We must set an error trap (see the `ErrorHandler` code). To add the data to the existing table, we must use the SQL `INSERT INTO` statement. The name of the table in the `SELECT...INTO` statement is followed by the `IN` clause and the name of the external database into which data is to be inserted. Again, you need to specify the full path to the target database file. Here, the path and database name are hardcoded for you to see another way of building an SQL insert statement string.

The name of the external database is followed by the `FROM` clause and the name of the existing table from which records are selected. You may select data from more than one table. You may also specify selection criteria following the `WHERE` clause. After creating the SQL statement, we execute it using the `Execute` method of the `Database` object.

Notice the use of the `dbFailOnError` option with the `Execute` method. If the statement fails, `dbFailOnError` will generate an error message we can trap. Without it, you are not notified of any errors, and the entire procedure may not produce the intended results. You can see how the error trap works by running the procedure more than once. If the `Execute` statement succeeds, we proceed to delete records from the source table. However, we don't want to execute the delete code if the `SELECT` statement returned no records. After the `Execute` command is run, we use the `RecordsAffected` property of the `Database` object to obtain the number of records affected by the most recent `Execute` command.

If we have more than one record, we specify the records to delete using the SQL `DELETE` statement, and then carry out the delete operation by calling the `Execute` method of the `Database` object. If `dbFailOnError` did not notify us of any errors, we assume that the `Execute` statement succeeded and we can

commit the transaction. Before carrying out this operation, we ask the user to confirm or cancel the transaction. If the user chooses not to go ahead with the changes, we roll back the transaction. We also withdraw changes to the records if there were no records to archive.

It is important to keep in mind that in case of an error you must roll back the transaction. Always check if the transaction is still active by using a flag. Rolling back the transaction will ensure that the transaction doesn't stay active after your VBA procedure has ended.

5. Run the procedure once again in step mode (using **F8**) to walk through the error code.

## SUMMARY

---

This chapter covered some basic and more advanced DAO material that you should find useful in developing professional applications in Access. You started by learning how to create and open databases using DAO methods. You learned how to create tables, add, and modify their properties, and work with various types of fields, keys, and indexes. Next, you spent a great deal of time learning and working with the `Recordset` object. With this object now pretty much mastered, you should have no trouble working with data. This chapter also showed you how you can create and run queries in your VBA procedures. Finally, you learned how transactions are used to ensure that certain database operations are always performed as a single unit.

In the next chapter, we will focus on writing VBA procedures that handle the same types of topics that we covered in this chapter by using the ActiveX Data Objects (ADO). We will also cover features that can only be accomplished by using ADO.

# Chapter 12 *CREATING AND MANIPULATING DATABASES WITH ADO*

In Chapter 11, we focused on performing various database tasks using the Data Access Objects library commonly referred to as DAO. In this chapter you learn how to create and manipulate databases using another library – ActiveX Data Objects or ADO.

## **CREATING AN ACCESS DATABASE WITH ADO**

To create a new Access database using ADO, you must use the ADOX Catalog object's `Create` method. The ADOX library is discussed in Chapter 18. The `Create` method creates and opens a new ADO connection to the data source. An error will occur if the provider does not support creating new catalogs.

The procedure in Hands-On 12.1 creates a new blank database named `TestADO.accdb` in your `C:\VBAAccess2021_ByExample` folder. The error trap ensures that the procedure works correctly even if the specified file already exists. This procedure uses the error handler to detect whether a database of the specified name already exists. When error `-2147217897` occurs, the procedure deletes the database file using the VBA `Kill` statement and returns to the statement that caused the error.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 12.1 Creating a Database

1. Open Access and create a new database called **Chap12.accdb** In your C:\VBAAccess2021\_ByExample folder.
2. In the Visual Basic Editor window, choose **Tools | References**. In the References dialog box, click the checkbox next to the **Microsoft ADO Ext. 6.0 for DDL and Security Object Library** and click **OK**.
3. Add a new standard module and enter the **CreateNewDB\_ADO** procedure shown here:

```
Sub CreateNewDB_ADO()

    ' you must make sure that a reference to
    ' Microsoft ADO Ext. 6.0 for DDL and Security
    ' Object Library is set in the References dialog box

    Dim cat As ADOX.Catalog
    Dim strDb As String

    Set cat = New ADOX.Catalog
    strDb = "C:\VBAAccess2021_ByExample\TestADO.accdb"

    On Error GoTo ErrorHandler
    cat.Create "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & strDb
    MsgBox "The database was created (" & strDb & ")."
    Set cat = Nothing
    Exit Sub

ErrorHandler:
    If Err.Number = -2147217897 Then
        Kill strDb
        Resume 0
    Else
        MsgBox Err.Number & ":" & Err.Description
    End If
End Sub
```

4. Choose **Run | Run Sub/UserForm** to execute the procedure.

## **Copying a Database**

---

At times you may want to duplicate your database programmatically. This can be easily done in DAO with the DBEngine object's `CompactDatabase` method. ADO does not have a special method for copying files. However, you can set up a reference to the File Scripting object (the Microsoft Scripting Runtime Library) to gain access to your computer filesystem, or use the `CreateObject` function to access this library without setting up a reference.

### ***Copying a Database with FileSystemObject***

---

You can use the `CopyFile` method of the `FileSystemObject` from the Microsoft Scripting Runtime Library to copy any file. This method allows you to copy one or more files and requires that you specify the source and destination. The source is the name of the file you want to copy or the file specification. For example, to copy all your databases located in a specific directory, you can include wildcard characters to specify the source like this: `C:\VBAAccess2021_ByExample\*.accdb`. The destination is the string specifying where the file or files are to be copied. You cannot use wildcard characters in the destination string. The third argument of the `CopyFile` method is optional. It indicates whether existing files in the destination are to be overwritten. If `True`, files are overwritten; if `False`, they are not. The default is `True`.

Hands-On 12.2 demonstrates how to copy a file from one directory to another using this method.



### **Hands-On 12.2 Copying a File Using FileSystem Object**

This hands-on exercise makes a copy of the `TestADO.accdb` database created in Hands-On 12.1.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **Copy\_AnyFile** procedure:

```
Sub Copy_AnyFile(strFileName As String)
    Dim fso As Object
    Dim strSourceFolder As String
    Dim strDestFolder As String
    Dim strDb As String

    On Error GoTo ErrorHandler
    strSourceFolder = "C:\VBAAccess2021_ByExample\
    strDestFolder = strSourceFolder & "TestFolder"
    strDb = strSourceFolder & strFileName
```

```
Set fso = CreateObject("Scripting.FileSystemObject")
fso.CreateFolder strDestFolder
fso.CopyFile strDb, strDestFolder & "\" & strFileName

Set fso = Nothing
Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
End Sub
```

3. To run the above procedure, type the following statement in the Immediate Window and press Enter:

```
Copy_AnyFile «TestADO.accdb»
```

This procedure uses the `CreateObject` method to return a reference to a `FileSystemObject` from the Microsoft Scripting Runtime Library. The `CreateFolder` method of the `FileSystemObject` is used to create a new folder named `TestFolder` in your `VBAAccess2021_ByExample` folder. The `CopyFile` method of the `FileSystemObject` is then used to copy the specified database to the newly created folder.

## DATABASE ERRORS

---

So far in this book you've seen several procedures that incorporated error handling. You already know that an *error handler* is a block of code that is executed when a runtime error occurs. The procedure execution is transferred to error-handling code via the `On Error GoTo <Label>` statement. Recall that there are three types of `On Error` statements:

- **On Error GoTo <Label>**—This statement tells VBA to jump to the specified label when an error occurs. A label is any unreserved word followed by a colon and is placed on a separate line in the same procedure as the `On Error` statement. The code between the line that caused the error and the line with the label is simply ignored. The execution of the procedure continues from the line following the label. The error-handling code is placed at the very bottom of the procedure. To ensure that the error handler is not executed if there are no errors, place an `Exit Sub` or `Exit Function` statement on a separate line just before the label.
- **On Error Resume Next**—This statement tells VBA to resume the procedure execution at the line following the statement that caused the error.

Place this statement in your code anywhere you think the error might occur. The runtime error will be trapped and stored in the VBA Err object. You should check the error number of the Err object immediately after that statement to determine how to handle the error.

- **On Error GoTo 0**—This statement disables the error handler in the current procedure. When an error occurs, VBA will display its standard runtime error message box in which you can click the End button to terminate the procedure or press Debug to enter the break mode for troubleshooting.
- In Chapter 9, you learned that VBA has a built-in `Err` object that has several properties useful for determining the type of error that occurred. You can use the `Err` object's `Number` property to determine the error number. The `Description` property contains the text description of the error. You can also find out the source of an error by using the `Source` property.
- When using ADO to access data, you can get information about the errors from both the VBA `Err` object and the ADO `Error` object. When an error occurs in an application that uses the ADO Object Model, an `Error` object is appended to the ADO `Errors` collection of the `Connection` object and you are advised about the error via a message box.
- While the VBA `Err` object holds information only about the most recent error, the ADO Errors collection can contain several entries regarding the last ADO error. You can count the errors caused by an invalid operation by using the `Count` property of the `Errors` collection. By checking the contents of the `Errors` collection you can learn more information about the nature of the error. The `Errors` collection is available only from the `Connection` object. Errors that occur in ADO itself are reported to the VBA `Err` object. Errors that are provider-specific are appended to the `Errors` collection of the ADO `Connection` object. These errors are reported by the specific OLE DB provider when ADO objects are being used to access data.

The DBError2 procedure in Hands-On 12.3 attempts to open a nonexistent database to demonstrate the capabilities of the VBA Err object and the ADO Errors collection.

### Hands-On 12.3 Using the VBA Err Object and ADO Errors Collection

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. Open **Tools | References** and set the reference to the **Microsoft ActiveX 6.1 Object Library**.

3. In the module's Code window, type the following **DBError2** procedure:

```
Sub DBError2()
    Dim conn As New ADODB.Connection
    Dim errADO As ADODB.Error

    On Error GoTo CheckErrors
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" _
        & "Data Source=C:\my.accdb"
    Debug.Print CurrentProject.Path

    CheckErrors:
    Debug.Print "Listed below is information " _
        & "regarding this error " & vbCrLf _
        & "contained in the ADO Errors collection."
    For Each errADO In conn.Errors
        Debug.Print vbTab & _
            "Error Number: " & errADO.Number
        Debug.Print vbTab & _
            "Error Description: " & errADO.Description
        Debug.Print vbTab & _
            "Jet Error Number: " & errADO.SQLState
        Debug.Print vbTab & _
            "Native Error Number: " & errADO.NativeError
        Debug.Print vbTab & _
            "Source: " & errADO.Source
        Debug.Print vbTab & _
            "Help Context: " & errADO.HelpContext
        Debug.Print vbTab & _
            "Help File: " & errADO.HelpFile
    Next
    MsgBox "Errors were written to the Immediate window."
End Sub
```

4. Choose **Run | Run Sub/UserForm** to execute the procedure.

In this procedure, an error is encountered when VBA attempts to open a database file that does not exist in the specified directory. The `On Error GoTo CheckErrors` statement tells VBA to jump to the line labeled `CheckErrors`. The line that prints the current project path is never executed. The `CheckErrors` handler reads the content of the VBA Err object and prints the error number and its description to the Immediate window. After that, we retrieve more information about the encountered errors by looping through the ADO Errors collection.

Here's the output from running the procedure in this Hands-On:  
Listed below is information regarding this error  
contained in the ADO Errors collection.

Error Number: -2147467259

Error Description: Could not find file 'C:\my.accdb'.

Jet Error Number: 3024

Native Error Number: -534578963

Source: Microsoft Access Database Engine

Help Context: 5003024

Help File:

**NOTE**

*To trace errors in your VBA procedures, don't forget to use the Step commands in the Visual Basic Debug menu (see Chapter 9 for more information).*

## OPENING A MICROSOFT JET DATABASE IN READ/WRITE MODE

---

You can use ADO to open an Access database for shared access (read/write). The names of common data providers used with ADO are listed in Chapter 10 (see Table 10.4).

To specify the data source name, use the Connection object's `ConnectionString` property. As you recall from earlier discussion, connection strings describe how to access data. Here's a code fragment that specifies the minimum required connection information:

```
With conn
    .Provider = "Microsoft.ACE.OLEDB.12.0;"
    .ConnectionString = "Data Source=" & CurrentProject.Path & _
        "\Northwind 2007.accdb"
End With
```

In the preceding example, the data source includes the full path to the database file you are going to open. Once you've specified the minimum connection information, you may proceed to open the database. Use the Connection object's `Open` method to open the connection to a data source:

```
conn.Open
```

ADO syntax is quite flexible. A connection to a database can also be opened like this:

```
conn.Open "Provider = Microsoft.ACE.OLEDB.12.0;Data Source=" &
CurrentProject.Path & "\Northwind 2007.accdb"
```

As you can see in the preceding code fragment, the Provider name and the data source (in this example, path to the database) are supplied as arguments when you call a Connection object's Open method.

Or you could open the database connection like this:

```
With conn
    .Provider = "Microsoft.ACE.OLEDB.12.0;"
    .Mode = adModeReadWrite
    .ConnectionString = "Data Source=" & CurrentProject.Path & _
        "\Northwind 2007.accdb"
    .Open
End With
```

By default, the Connection object's Open method opens a database for shared access. You can use the Connection object's Mode property to explicitly specify the type of access to a database. The Mode property must be set prior to opening the connection because it is read-only once the connection is open. Connections can be opened read-only, write-only, or read/write. You can also specify whether other applications should be prevented from opening a connection. The value for the Mode property can be one of the constants/values specified in Table 12.1.

**TABLE 12.1.** Intrinsic constants of the Connection object's Mode property

Constant Name	Value	Type of Permission
adModeUnknown	0	Permissions have not been set yet or cannot be determined. This is the default setting.
adModeRead	1	Read-only permissions.
adModeWrite	2	Write-only permissions.
adModeReadWrite	3	Read/write permissions.
adModeShareDenyRead	4	Prevents others from opening the connection with read permissions.
adModeShareDenyWrite	8	Prevents others from opening the connection with write permissions.
adModeShareExclusive	12	Prevents others from opening the connection.
adModeShareDenyNone	16	Prevents others from opening the connection with any permissions.

Hands-On 12.4 demonstrates how to use ADO to open an Access database for shared access (read/write).

### ① Hands-On 12.4 Opening a Database with ADO in Read/Write Mode

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new module to the currently open Chap12.accdb database.
2. In the module's Code window, type the following **openDB\_ADO** procedure:

```
Sub openDB_ADO()
    Dim conn As ADODB.Connection
    Dim strDb As String

    On Error GoTo ErrorHandler

    strDb = CurrentProject.Path & "\Northwind 2007.accdb"
    Set conn = New ADODB.Connection

    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0;"
        .Mode = adModeReadWrite
        .ConnectionString = "Data Source=" & strDb
        .Open
    End With

    If conn.State = adStateOpen Then
        MsgBox "Connection was opened."
    End If

    conn.Close
    Set conn = Nothing
    MsgBox "Connection was closed."
    Exit Sub

ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
End Sub
```

3. Position the insertion point anywhere within the code of the **openDB\_ADO** procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.

The ADO Connection object's **State** property returns a value that describes whether the connection is open, closed, connecting, executing, or retrieving data (see Table 12.2).

```
If conn.State = adStateOpen Then
```

```

    MsgBox "Database connection was established."
End If

```

**TABLE 12.2.** Intrinsic constants of the Connection object's State property

Constant	Value	Description
adStateClosed	0	Connection is closed.
adStateOpen	1	Connection is open.
adStateConnecting	2	Connection is connecting.
adStateExecuting	4	Connection is executing a command.
adStateFetching	8	Connection is retrieving data.

If an error occurs during the procedure execution (for example, when a database with the specified name or path cannot be found), the statement `On Error GoTo ErrorHandler` will pass the program control to the error-handling code located at the `ErrorHandler` label at the bottom of the procedure. Errors that occur in ADO are reported to the VBA Err object. You can find out the details about the error that occurred by using various properties of the Err object (Name, Description, Source, HelpFile, or HelpContext). The code in the error handler will execute only if an error occurs. If the procedure executes without an error, the `Exit Sub` statement will cause the procedure to finish without running the error code. You will find more information on database errors near the end of this chapter.

### **Connecting to the Current Access Database**

Microsoft Access provides a quick way to access the current DAO database by using the `CurrentDb` method. This method returns an object variable of type Database that represents the database currently open in the main Access window. In ADO, however, use the `CurrentProject.Connection` statement to access the currently open database. The `CurrentProject` object refers to the project for the current Access database. These statements work only in VBA procedures created in Access. If you'd like to reuse your VBA procedures in other Microsoft 365 Visual Basic applications, you will be better off creating a connection via an appropriate OLE DB provider.

The procedure in Hands-On 12.5 uses the `CurrentProject.Connection` statement to return a reference to the current database. Once the connection to the current database is established, the example procedure loops through the Properties collection of the Connection object to retrieve its property names and settings. The results are written both to the Immediate window and to a text file named `C:\VBAAccess2021_ByExample\Propfile.txt`.



### Hands-On 12.5 Establishing a Connection to the Current Access Database

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **Connect\_ToCurrentDB** procedure shown here:

```
Sub Connect_ToCurrentDB()
    Dim conn As ADODB.Connection
    Dim fs As Object
    Dim txtfile As Object
    Dim i As Integer
    Dim strFileName As String

    strFileName = "C:\VBAAccess2021_ByExample\Propfile.txt"

    Set conn = CurrentProject.Connection
    Set fs = CreateObject("Scripting.FileSystemObject")
    Set txtfile = fs.CreateTextFile(strFileName, True)

    For i = 0 To conn.Properties.Count - 1
        Debug.Print conn.Properties(i).Name & "=" & _
            conn.Properties(i).Value
        txtfile.WriteLine (conn.Properties(i).Name & _
            "=" & conn.Properties(i).Value)
    Next i
    MsgBox "Please check results in the " & _
        "Immediate window." & vbCrLf & _
        "The results have also been written to the " & _
        & Chr(13) & strFileName & " file."
    txtfile.Close

    Set fs = Nothing
    conn.Close
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.
- The **Connect\_ToCurrentDB** procedure uses the `CurrentProject.Connection` statement to get a reference to the currently open database. To create a text file from a VBA procedure, the `CreateObject` function is used to access the `Scripting.FileSystemObject`. This function returns the `FileSystemObject` (`fs`). The `CreateTextFile` method of the `FileSystemObject` creates the `TextStream` object that represents a text file (`txtfile`). The `WriteLine` method writes each

property and the corresponding setting to the newly created text file (C:\Propfile.txt). Finally, the `Close` method closes the text file.

## OPENING OTHER DATABASES, SPREADSHEETS, AND TEXT FILES FROM ACCESS

---

The Microsoft Access Jet/ACE database engine can be used to access other databases, spreadsheets, and text files. The following subsections of this chapter demonstrate how to connect to SQL Server, Excel spreadsheets, and text files.

### Connecting to an SQL Server Database

The ADO provides a number of ways of connecting to an SQL Server database. To access data residing on Microsoft SQL Server, use SQLOLEDB, which is the native Microsoft OLE DB provider for SQL.

You can also connect to an SQL database using the MSDASQL provider. This provider allows you to access any existing ODBC data sources. You can open a connection to the SQL Server by using an ODBC DSN (Data Source Name) or an ODBC DSN-less connection. Both of these connection types were discussed in Chapter 10. The following code snippet opens and then closes a connection with the SQL Server database based on a DSN named Pubs.

```
With conn
    .Open "Provider=MSDASQL;DSN=Pubs"
    .Close
End With
```

Recall that you can skip setting the Provider property because MSDASQL is the default provider for ODBC. All you really need to establish a connection in this case is a DSN.

Additional Code in the companion files.

Filename: **SQLOLEDB\_Provider.txt**

Description: Connecting to an SQL Server Database Using SQLOLEDB Provider

### Opening a Microsoft Excel Workbook

---

To open Excel 2007–2021 workbook files with the .xlsx file format using ADO, use the Microsoft ACE OLEDB 12.0 provider and use the Extended Properties of the ADO Connection object to pass the connection string like this:

```
Dim conn As ADODB.Connection
Set conn = New ADODB.Connection
conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
"Data Source=C:\VBAAccess2021_ByExample\" & _
"Report2021.xlsx;" & _
"Extended Properties=""Excel 12.0; HDR=YES"";"
```

To open workbook files created in Excel 2000–2003, use the Microsoft Jet OLE DB 4.0 provider and Excel 8.0 in the Extended Properties:

```
Dim conn As ADODB.Connection
conn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
"Data Source=C:\VBAAccess2021_ByExample\Report.xls;" & _
"Extended Properties=""Excel 8.0; HDR=YES"";"
```

You can also use ODBC to open an Excel workbook file. For example, the following code snippet establishes an ODBC DSN-less connection:

```
Dim conn As ADODB.Connection
Set conn = New ADODB.Connection

With conn
    .ConnectionString = "Driver={Microsoft Excel Driver " & _
    "(*.xls, *.xlsx, *.xlsm, *.xlsb)};" & _
    "DBQ=C:\VBAAccess2021_ByExample\Report2021.xlsx;" 
    .Open
End With
```

Hands-On 12.6 demonstrates how to open an Excel workbook with ADO and modify its data.



## Hands-On 12.6 Opening an Excel Workbook

- In the standard module, enter the following **Open\_Excel\_ADO** procedure:

```
Sub Open_Excel_ADO(strFileName As String)
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strFindWhat As String

    Set conn = New ADODB.Connection

    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0;"
        .ConnectionString = "Data Source=" & _
        CurrentProject.Path & "\" & strFileName & _
        ";Extended Properties=""Excel 12.0;HDR=Yes;IMEX=0"";"
```

```
.Open
End With

Set rst = New ADODB.Recordset
rst.Open "SELECT * FROM [Sheet1$]", conn, _
adOpenStatic, adLockOptimistic

strFindWhat = "[Excel Version] = 'Excel 2000'"
rst.Find strFindWhat
rst(1).Value = "500"
rst.Update
rst.Close
Set rst = Nothing
MsgBox "Excel workbook was opened and updated."

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

2. In the Visual Basic Editor window, press **Ctrl+G** to open the Immediate window or choose **View | Immediate Window**.
3. To run the `Open_Excel_ADO` procedure, type `Open_Excel_ADO "Report2021.xlsx"` in the Immediate window and press **Enter**.

Notice how the `Open_Excel_ADO` procedures passed the connection string to the ADO Connection object's `Open` method. The Provider is set to Microsoft ACE OLEDB 12.0 for the current version of an Excel file, and Extended Properties is set to use Excel 12.0. Notice, the IMEX option, which stands for Import Export mode, is set to zero (IMEX=0). This setting will allow the data in the worksheet to be updatable. When IMEX=1, the file becomes read-only, and you'll get an error on attempt to update the recordset. Once the connection to the workbook file is open, an ADO Recordset is opened. We instruct the procedure to select all data from the Sheet1 worksheet using the following SQL statement:

```
"SELECT * FROM [Sheet1$]"
```

Notice that in the `SELECT` statement, the sheet name must be enclosed in square brackets and have a dollar sign (\$) appended to it. The Recordset is opened using the open connection (`conn`). The procedure uses the ADO constants `adOpenStatic` (Cursor Type parameter) and `adLockOptimistic` (Lock Type parameter) to ensure that the Recordset is updatable. ADO Recordsets are discussed in detail later in this chapter.

Before you can modify data in a worksheet, you must find it. The search criteria string is defined in the `strFindWhat` variable. To find the data, the procedure uses the `Find` method of the `Recordset` object. Once the searched data is located, we simply assign a new value to the `Recordset` field using the `Value` property:

```
rst(1).Value = "500"
```

The ADO Recordset fields are counted beginning with zero (0). Therefore, the preceding statement sets the value in the second column in the worksheet. To save the changes to the file, call the `Update` method, like this:

```
rst.Update
```

The remaining code in this procedure performs the standard cleanup: closing the objects and releasing the memory used by the object variables (`rst`, `conn`).

## **Opening a Text File**

There are several ways to open text files programmatically. This section demonstrates how to gain access to a text file by using the Microsoft Text Driver. Notice that this is a DSN-less connection (as explained in Chapter 10). Hands-On 12.7 demonstrates how to open a Recordset based on a comma-separated file format and write the file contents to the Immediate window.



### **Hands-On 12.7 Opening a Text File**

1. Copy the `Employees.txt` file from the companion files to your `C:\VBAAccess2021_ByExample` folder, or prepare the text file from scratch by typing the following in Notepad and saving the file as `C:\VBAAccess2021_ByExample\Employees.txt`:

```
"Last Name", "First Name", "Birthdate", "Years Worked"  
"Krawiec", "Bogdan", #1963-01-02#, 3  
"Gorecka", "Jadwiga", #1948-05-12#, 1  
"Olszewski", "Stefan", #1957-04-07#, 0
```

2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following `Open_TextFile` procedure:

```
Sub Open_TextFile()  
    Dim conn As ADODB.Connection  
    Dim rst As ADODB.Recordset
```

```
Dim fld As ADODB.Field

Set conn = New ADODB.Connection
Debug.Print conn.ConnectionString
conn.Open "DRIVER={Microsoft Access Text Driver (*.txt, *.csv)};" & _
"DBQ=" & CurrentProject.Path & "\"

Set rst = New ADODB.Recordset
rst.Open "SELECT * FROM [Employees.txt]", conn, adOpenStatic,
adLockReadOnly, adCmdText
Do Until rst.EOF
    For Each fld In rst.Fields
        Debug.Print fld.Name & "=" & fld.Value
    Next fld
    rst.MoveNext
Loop
rst.Close
Set rst = Nothing

conn.Close
Set conn = Nothing
MsgBox "Open the Immediate window to view the data."
End Sub
```

4. Make sure that the C:\VBAAccess2021\_ByExample\Employees.txt file is closed and choose **Run | Run Sub/UserForm** to execute the procedure.
5. Open the Immediate window to view the procedure results.

**NOTE**

*If you are getting an error while running this procedure check that the driver is spelled out exactly as it appears on the Drivers tab in the ODBC Data Source Administrator (64-bit). If you are running on a 32-bit system, you should change the ODBC Driver name to: Microsoft Text Driver (\*.txt, \*.csv).*

If you worked through the previous exercises in this chapter, you should have no problem following the code of the Open\_TextFile procedure. Because you are only reading the records, you can open the Recordset using the `adOpenStatic` and `adLockReadOnly` ADO constants. Notice that the ADO constant `adCmdText` is used as the last parameter of the Recordset's `Open` method:

```
rst.Open "SELECT * FROM [Employees.txt]", conn, adOpenStatic,
adLockReadOnly, adCmdText
```

The last parameter in the preceding statement can be any valid option. You can indicate the type of source you are using with the `adCmdText` constant (for an SQL statement), `adCmdTable` (to retrieve all the rows in a table), or `adCmdStoredProc` (to get records via a stored procedure). If you do not specify the type of source, `adCmdUnknown` is used as the default.

## CREATING A MICROSOFT ACCESS TABLE AND SETTING FIELD PROPERTIES

---

You can get going with your database design by using Access objects contained in the ADOX library. The full name of this library is ActiveX Data Object Extensions for DDL and Security. To use ADOX in your VBA procedures, choose Tools | References from your Visual Basic Editor window and select Microsoft ADO Ext. 6.0 for DDL and Security. The ADOX Object Model is an extension of the ADODB library.

The most important ADOX object is called Catalog. It represents an entire database and contains database tables, columns, indexes, groups, users, procedures, and views. You will use the ADOX Catalog object in your VBA procedures to create a table.

The following steps outline the process of creating a new Access table:

1. Declare the variables representing the Connection, Catalog, and Table objects:

```
Dim conn As ADODB.Connection  
Dim cat As ADOX.Catalog  
Dim tbl As ADOX.Table
```

2. Open the connection to your database:

```
Set conn = New ADODB.Connection  
conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _  
    "Data Source=C:\VBAAccess2021_ByExample\Chap12b.mdb"
```

3. Supply the open connection to the ActiveConnection property of the ADOX Catalog object:

```
Set cat = New ADOX.Catalog  
Set cat.ActiveConnection = conn
```

4. Create a new Table object:

```
Set tbl = New ADOX.Table
```

**5.** Provide the name for your table:

```
tbl.Name = "tblAssets"
```

The Table object is a member of the Tables collection, which in turn is a member of the Catalog object. Each Table object has a Name property and a Type property. The Type property specifies whether a Table object is a standard Access table, a linked table, a system table, or a view.

**6.** Append the Table object to the Catalog object's Tables collection:

```
cat.Tables.Append tbl
```

At this point your table is empty.

**7.** Add new fields (columns) to your new table:

```
With tbl.Columns
    .Append "SiteID", adVarWChar, 10
    .Append "Category", adSmallInt
    .Append "InstallDate", adDate
End With
```

The preceding code fragment creates three fields named SiteID, Category, and InstallDate. You can create new fields in a table by passing the Column object's Name, Type, and DefinedSize properties as arguments of the Columns collection's Append method. Notice that ADOX uses different data types than those used in the Access user interface (see Table 12.3 for a comparison of the data types).

<b>NOTE</b>	<p><i>The Table object contains the Columns collection that contains Column objects. To add a new field to a table, you could create a Column object and write the code like this:</i></p> <pre>Dim col As ADOX.Column set col = New ADOX.Column With col     .Name = "SiteID"     .DefinedSize = 10 End With tbl.Columns.Append col</pre> <p><i>The last statement in the preceding example appends the new Column object (field) to the Columns collection of a table. The Name property specifies the name of the column. The DefinedSize property designates the maximum size of an entry in the column. To create another field, you would have to create a new Column object and set its properties. Creating fields in this manner takes longer and is less efficient than using the method demonstrated earlier.</i></p>
-------------	--

The complete procedure is shown here:

```

Sub CreateTableADO()
Dim conn As ADODB.Connection
Dim cat As ADOX.Catalog
Dim tbl As ADOX.Table

' make sure to set up a reference to
' the Microsoft ActiveX Data Objects 6.1 Library
' and ADO Ext. 6.0 for DDL and Security

' copy Chap12b.mdb from the companion files
' to your C:\VBAAccess2021_ByExample folder

Set conn = New ADODB.Connection
conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
"Data Source=C:\VBAAccess2021_ByExample\Chap12b.mdb"

Set cat = New ADOX.Catalog
Set cat.ActiveConnection = conn

Set tbl = New ADOX.Table
tbl.Name = "tblAssets"

cat.Tables.Append tbl

With tbl.Columns
    .Append "SiteID", adVarWChar, 10
    .Append "Category", adSmallInt
    .Append "InstallDate", adDate
End With

Set cat = Nothing
conn.Close
Set conn = Nothing
End Sub

```

**TABLE 12.3** ADO data types versus Access data types

ADO Data Type	Corresponding Data Type in Access
adBoolean	Yes/No
adUnsignedTinyInt	Number (FieldSize = Byte)
adSmallInt	Number (FieldSize = Integer)

(Contd.)

ADO Data Type	Corresponding Data Type in Access
adSingle	Number (FieldSize = Single)
adDouble	Number (FieldSize = Double)
adDecimal	Number (FieldSize = Decimal)
adInteger	Number (FieldSize = LongInteger) AutoNumber
adCurrency	Currency
adVarWChar	Text
adDate	Date/Time
adLongVarBinary	OLE object
adLongVarWChar	Memo
adLongVarWChar	Hyperlink

**NOTE** *ADO does not support the Attachment data type, multi select lookup fields, and the Append Only and Rich Text memo fields that were first introduced in Access 2007. To programmatically access these features in Access 2007–2021, you must rely on DAO (see example procedures in Chapter 11).*

## COPYING A TABLE

The procedure in Hands-On 12.8 uses the SQL `SELECT...INTO` statement to select all records from the Customers table in the Northwind database and place them into a new table called `CustomersCopy`. The `SELECT...INTO` statement is equivalent to a MakeTable query in the Access user interface. This statement creates a new table and inserts data from other tables. To copy a table, the SQL statement is passed as the first argument of the `Execute` method of the ADO Connection object. Note that the copied table will not have the indexes that may exist in the original table.



### Hands-On 12.8 Making a Copy of a Table

1. Backup the Northwind.mdb file to `C:\VBAAccess_ByExample\Northwind_Chap12.mdb`.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following `Copy_Table` procedure:

```
' make sure to set up a reference to
' the Microsoft ActiveX Data Objects 6.1 Library
```

```
Sub Copy_Table()
    Dim conn As ADODB.Connection
    Dim strTable As String
    Dim strSQL As String

    On Error GoTo ErrorHandler

    strTable = "Customers"

    strSQL = "SELECT " & strTable & ".* INTO "
    strSQL = strSQL & strTable & "Copy "
    strSQL = strSQL & "FROM " & strTable

    Debug.Print strSQL

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    conn.Execute strSQL
    conn.Close
    Set conn = Nothing
    MsgBox "The " & strTable & " table was copied."
    Exit Sub

ErrorHandler:
    If Err.Number = -2147217900 Then
        conn.Execute "DROP Table " & strTable
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
    End If
End Sub
```

**4. Choose Run | Run Sub/UserForm** to execute the procedure.

When you run this procedure, Access creates a copy of the Customers table named CustomersCopy in the Northwind\_Chap12.mdb database.

---

## DELETING A DATABASE TABLE

You can use ADO to delete a table programmatically by opening the ADOX Catalog object, accessing its Tables collection, and calling the Delete method.

The procedure in Hands-On 12.9 requires a parameter that specifies the name of the table you want to delete.



### Hands-On 12.9 Deleting a Table from a Database

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **Delete\_Table** procedure shown here:

```
Sub Delete_Table(strTblName As String)
    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog

    On Error GoTo ErrorHandler

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set cat = New ADOX.Catalog

    cat.ActiveConnection = conn

    cat.Tables.Delete strTblName
    Set cat = Nothing
    conn.Close
    Set conn = Nothing

    Exit Sub

ErrorHandler:
    MsgBox "Table '" & strTblName & _
        "' cannot be deleted " & vbCrLf & _
        "because it does not exist."
    Resume Next
End Sub
```

3. To run this procedure, type the following statement in the Immediate window and press **Enter**:

```
Delete_Table "CustomersCopy"
```

The CustomersCopy table was created by running the **Copy\_Table** procedure in Hands-On 12.8. When you press Enter, Visual Basic will delete the specified table from the Northwind\_Chap12.mdb database. If the table does not exist, an appropriate message is displayed.

## ADDING NEW FIELDS TO AN EXISTING TABLE

At times you may want to programmatically add a new field to an existing table. The procedure in Hands-On 12.10 adds a new text field called MyNewField to a table located in the Northwind\_Chap12.mdb database.



### Hands-On 12.10 Adding a New Field to a Table

The procedure demonstrated in this hands-on exercise uses the CustomersCopy table in the Northwind database.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **Add\_NewFields** procedure:

```
Sub Add_NewFields()
    Dim conn As ADODB.Connection
    Dim cat As New ADOX.Catalog

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set cat = New ADOX.Catalog
    cat.ActiveConnection = conn
    cat.Tables("CustomersCopy").Columns.Append _
        "MyNewField", adVarWChar, 15

    Set cat = Nothing
    conn.Close
    Set conn = Nothing
End Sub
```

3. Run the **Copy\_Table** procedure in Hands-On 12.9 to ensure that the CustomersCopy table exists in the Northwind database.
4. Choose **Run | Run Sub/UserForm** to run the Add\_NewFields procedure. Figure 12.1 shows a new field called MyNewField added to the CustomersCopy table.

ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax	MyNewField
Sales Represent	Obere Str. 57	Dresden		12209	Germany	030-0074321	030-0076545	
Owner	Avda. de la Con: México D.F.			05021	Mexico	(5) 555-4729	(5) 555-3745	
Owner	Mataderos 231 México D.F.			05023	Mexico	(5) 555-3932		
Sales Represent	120 Hanover Sq Dover			WA1 1DP	UK	(171) 555-7788	(171) 555-6750	
Order Administratör	Berguvsvägen 8 Luleå			S-958 22	Sweden	0921-12 34 65	0921-12 34 67	

FIGURE 12.1 The MyNewField in the CustomersCopy table was added in Hands-On 12.11.

If you run the previous procedure again, Microsoft Visual Basic will display a runtime error “Cannot define field more than once.” Therefore, before adding a new field, it is always a good idea to check whether the field already exists in the table.



### Hands-On 12.11 Checking for Field Existence

1. On your own, rewrite the procedure in Hands-On 12.10 to check for the existence of the specified field prior to adding it to the table.

The revised Add\_NewFields procedure can be found in the companion files (Chap12\_HandsOn\_12.11.txt).

## REMOVING A FIELD FROM A TABLE

---

You may remove any field from an existing table, whether or not this field contains data. However, you can't delete a field after you have created an index that references that field. You must first delete the index.

The procedure in Hands-On 12.12 illustrates how to access the ADOX Columns collection of a Table object and use the Columns collection's Delete method to remove a field from a table. This procedure will fail if the field you want to delete is part of an index.



### Hands-On 12.12 Removing a Field from a Table

This hands-on exercise requires that you created and executed the CopyTable and Add\_NewFields procedures in Hands-On 12.8 and 12.10.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **Delete\_Field** procedure shown here:

```
Sub Delete_Field()
    Dim conn As ADODB.Connection
    Dim cat As New ADOX.Catalog

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set cat = New ADOX.Catalog
    cat.ActiveConnection = conn
    cat.Tables("CustomersCopy").Columns.Delete _
        "MyNewField"
```

```
Set cat = Nothing
conn.Close
Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

## RETRIEVING TABLE PROPERTIES

---

You can set or retrieve table properties using the Properties collection of an ADOX Table object. The Properties collection exposes standard ADO properties as well as properties specific to the data provider. You can iterate through all of the properties of an object using the `For Each...Next` programming structure.

The procedure in the following hands-on exercise accesses the Customers-Copy table (see Hands-On 12.9) and lists its properties and their values in the Immediate window (see Figure 12.2).



### Hands-On 12.13 Listing Table Properties

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `List_TableProperties` procedure:

```
Sub List_TableProperties()
    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog
    Dim tbl As ADOX.Table
    Dim pr As ADOX.Property

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set cat = New ADOX.Catalog
    cat.ActiveConnection = conn

    Set tbl = cat.Tables("CustomersCopy")

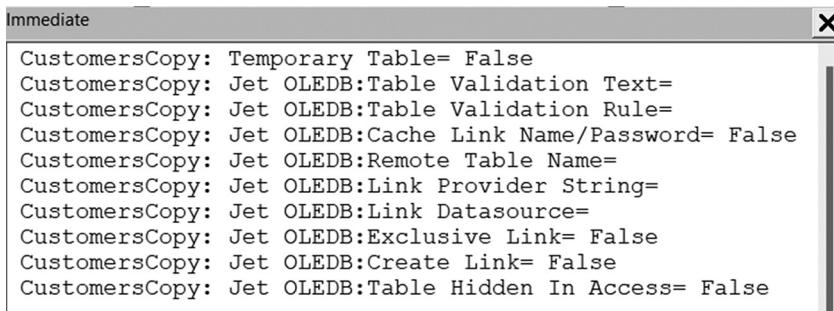
    ' retrieve table properties
    For Each pr In tbl.Properties
```

```

    Debug.Print tbl.Name & ":" & _
    pr.Name & "=" & pr.Value
Next
Set cat = Nothing
conn.Close
Set conn = Nothing
End Sub

```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.



The screenshot shows the Microsoft Access Immediate window. It contains a list of properties for a table named 'CustomersCopy'. Each property is followed by its value, separated by an equals sign (=). The properties listed are: Temporary Table= False, Jet OLEDB:Table Validation Text=, Jet OLEDB:Table Validation Rule=, Jet OLEDB:Cache Link Name/Password= False, Jet OLEDB:Remote Table Name=, Jet OLEDB:Link Provider String=, Jet OLEDB:Link Datasource=, Jet OLEDB:Exclusive Link= False, Jet OLEDB:Create Link= False, and Jet OLEDB:Table Hidden In Access= False.

**FIGURE 12.2** You can list the names of table properties and their values programmatically as shown in Hands-On 12.12.

## RETRIEVING FIELD PROPERTIES

---

The procedure in Hands-On 12.14 retrieves the field properties of the field named CustomerID located in the Customers table in the Northwind\_Chap12.mdb database and prints them to the Immediate window, as shown in Figure 12.3.



### Hands-On 12.14 Listing Field Properties

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **List\_FieldProperties** procedure shown here:

```

Sub List_FieldProperties()
    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog
    Dim col As ADOX.Column
    Dim pr As ADOX.Property

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _

```

```

    "Data Source=" & CurrentProject.Path & _
    "\Northwind_Chap12.mdb"

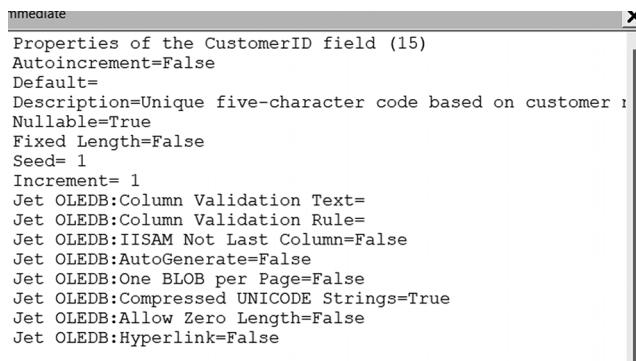
    Set cat = New ADOX.Catalog
    Set cat.ActiveConnection = conn
    Set col = New ADOX.Column
    Set col = cat.Tables("Customers"). _
        Columns("CustomerID")

    Debug.Print "Properties of the CustomerID " & _
        "field (" & col.Properties.Count & ")"
        ' retrieve Field properties
    For Each pr In col.Properties
        Debug.Print pr.Name & "="; pr.Value
    Next

    Set cat = Nothing
End Sub

```

**3. Choose Run | Run Sub/UserForm to execute the procedure.**



The screenshot shows the Microsoft Visual Studio Immediate window. The title bar says "Immediate". The window contains a list of properties for the "CustomerID" field, which has a count of 15 properties. The properties listed include Autoincrement=False, Default=, Description=Unique five-character code based on customer:, Nullable=True, Fixed Length=False, Seed=1, Increment=1, Jet OLEDB:Column Validation Text=, Jet OLEDB:Column Validation Rule=, Jet OLEDB:IISAM Not Last Column=False, Jet OLEDB:AutoGenerate=False, Jet OLEDB:One BLOB per Page=False, Jet OLEDB:Compressed UNICODE Strings=True, Jet OLEDB:Allow Zero Length=False, and Jet OLEDB:Hyperlink=False.

**FIGURE 12.3** Running the procedure in Hands-On 12.14 generates a list of field properties and their values in the Immediate window.

## LINKING A MICROSOFT ACCESS TABLE

---

To create a linked Access table, you must set the following table properties:

```

Jet OLEDB:LinkDatasource
Jet OLEDB:Remote Table Name
Jet OLEDB:CreateLink

```

The procedure in Hands-On 12.15 demonstrates how to establish a link to the Customers table located in the Northwind\_Chap12.mdb database.



### Hands-On 12.15 Linking a Microsoft Jet Table

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **Link\_JetTable** procedure as shown below.

```
Sub Link_JetTable()
    Dim cat As ADOX.Catalog
    Dim lnkTbl As ADOX.Table
    Dim strDb As String
    Dim strTable As String

    On Error GoTo ErrorHandler

    strDb = CurrentProject.Path & "\Northwind_Chap12.mdb"
    strTable = "Customers"
    Set cat = New ADOX.Catalog
    cat.ActiveConnection = CurrentProject.Connection

    Set lnkTbl = New ADOX.Table
    With lnkTbl
        ' Name the new Table and set its ParentCatalog property to the
        ' open Catalog to allow access to the Properties collection.

        .Name = strTable
        Set .ParentCatalog = cat

        ' Set the properties to create the link
        .Properties("Jet OLEDB:Create Link") = True
        .Properties("Jet OLEDB:Link Datasource") = strDb
        .Properties("Jet OLEDB:Remote Table Name") = strTable
    End With

    ' Append the table to the Tables collection
    cat.Tables.Append lnkTbl

    Set cat = Nothing
    MsgBox "The current database contains a linked " & _
           "table named " & strTable
    Exit Sub

ErrorHandler:
    MsgBox Err.Number & ": " & Err.Description
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

To access the linked Customers table after running this procedure, be sure to refresh the Access application window by pressing **Ctrl+F5**.

## LINKING A MICROSOFT EXCEL WORKSHEET

---

You can link an Excel worksheet to an Access database by using the `TransferSpreadsheet` method of the `DoCmd` object, as shown in Hands-On 12.16. Note, however, that neither the `DoCmd` object nor its `TransferSpreadsheet` method are members of the ADO Object Model. The `DoCmd` object is built into the Microsoft Access library.



### Hands-On 12.16 Linking an Excel Worksheet

This hands-on exercise uses the `Regions.xlsx` workbook file provided in the companion files. You can revise the procedure code to use any workbook file that you have available; however, you must match the name of the spreadsheet constant with the Excel version. Table 12.4 shows the constant names and values if you need a different format.

TABLE 12.4 Spreadsheet constants

Constant	Value	Description
<code>acSpreadsheetTypeExcel3</code>	0	Microsoft Excel 3.0 format
<code>acSpreadsheetTypeExcel4</code>	6	Microsoft Excel 4.0 format
<code>acSpreadsheetTypeExcel5</code>	5	Microsoft Excel 5.0 format
<code>acSpreadsheetTypeExcel7</code>	5	Microsoft Excel 95 format
<code>acSpreadsheetTypeExcel8</code>	8	Microsoft Excel 97 format
<code>acSpreadsheetTypeExcel9</code>	8	Microsoft Excel 2000–2003 format
<code>acSpreadsheetTypeExcel12</code>	9	Microsoft Excel 2007–2010 format (.xlsx)
<code>acSpreadsheetTypeExcel12Xml</code>	10	Microsoft Excel 2007–2021 format (.xml)

1. Copy the `Regions.xlsx` workbook from the **companion files** to your `C:\VBAAccess2021_ByExample` folder.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following `Link_ExcelSheet` procedure:

```
Sub Link_ExcelSheet()
    Dim rst As ADODB.Recordset
```

```

DoCmd.TransferSpreadsheet acLink, _
    acSpreadsheetTypeExcel12, _
    "mySheet", _
    CurrentProject.Path & "\Regions.xlsx", _
    -1, "Regions!A1:B15"

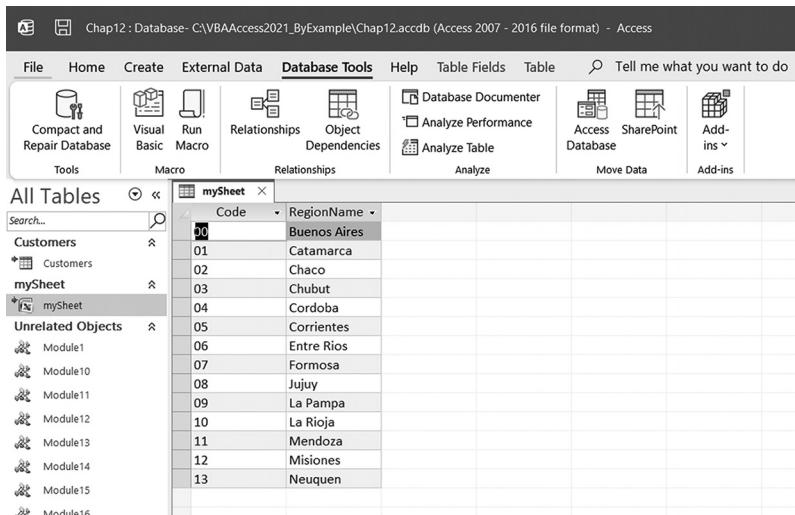
Set rst = New ADODB.Recordset
With rst
    .ActiveConnection = CurrentProject.Connection
    .CursorType = adOpenKeyset
    .LockType = adLockOptimistic
    .Open "mySheet", , , adCmdTable
End With

Do Until rst.EOF
    Debug.Print rst.Fields(0).Value, _
        rst.Fields(1).Value
    rst.MoveNext
Loop
rst.Close
Set rst = Nothing
End Sub

```

**4. Choose Run | Run Sub/UserForm to execute the procedure.**

The current database navigation pane should now show the mySheet linked table that displays data from an Excel worksheet (Figure 12.4).



**FIGURE 12.4** The Access Database Navigation pane with two linked tables created by running procedures in this chapter.

The `Link_ExcelSheet` procedure begins by creating a linked table named `mySheet` from the specified range of cells (A1:B15) in the `Regions` worksheet in the `Regions.xls` file. The first argument in the `DoCmd` statement indicates that the first row of the spreadsheet contains column headings. Next, the procedure uses the ADO Recordset object to retrieve the data from the `mySheet` table into the Immediate window. Notice that prior to opening the Recordset object, several properties of the Recordset object must be set:

- The `ActiveConnection` property sets the reference to the current database.
- The `CursorType` property specifies how the Recordset object should interact with the data source.

The `adOpenKeyset` setting tells Visual Basic that instead of retrieving all the records from the data source, only the keys are to be retrieved. The data for these keys is retrieved only as you scroll through the recordset. This guarantees better performance than retrieving big chunks of data at once.

- The `LockType` property determines how to lock the data while it is being manipulated.

The `adLockOptimistic` setting locks the record only when you attempt to save it.

- Opening the Recordset object also requires that you specify the data source. The data source in this procedure is the linked table named `mySheet`. The parameter passed depends on the source type used.

The `adCmdTable` setting indicates that all rows from the source table should be included.

You could also open the Recordset object by passing all the required parameters at once, as follows:

```
rst.Open "mySheet",  
        CurrentProject.Connection, adOpenKeyset,  
        adLockOptimistic, adCmdTable
```

## **LISTING DATABASE TABLES**

---

The procedure in Hands-On 12.17 generates a list of tables in the `Northwind_Chap12.mdb` database. It uses the ADOX Catalog object to gain access to the database, then iterates through the `Tables` collection to retrieve the names of Access tables, system tables, and views. The ADOX `Tables` collection stores various types of Table objects, as shown in Table 12.5.

**TABLE 12.5** Types of tables in the ADOX Tables collection

Name	Description
ACCESS TABLE	An Access system table
LINK	A linked table from a non-ODBC data source
PASS-THROUGH	A linked table from an ODBC data source
SYSTEM TABLE	A Microsoft Jet system table
TABLE	A Microsoft Access table
VIEW	A table from a row-returning, non-parameterized query



### Hands-On 12.17 Creating a List of Database Tables

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **ListTbls** procedure:

```
Sub ListTbls()
    Dim cat As ADOX.Catalog
    Dim tbl As ADOX.Table

    Set cat = New ADOX.Catalog
    cat.ActiveConnection =
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    For Each tbl In cat.Tables
        If tbl.Type <> "VIEW" And
            tbl.Type <> "SYSTEM TABLE" And
            tbl.Type <> "ACCESS TABLE" Then
            Debug.Print tbl.Name
        End If
    Next tbl
    Set cat = Nothing
    MsgBox "View the list of tables in " & _
        "the Immediate window."
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

## LISTING TABLES AND FIELDS

---

Earlier in this chapter you learned how to enumerate tables in the Northwind database by accessing the Tables collection of the ADOX Catalog object. The

procedures in Hands-On 12.18 and Hands-On 12.19 demonstrate how to use the `OpenSchema` method of the ADO Connection object to obtain more information about a database table and its fields.



### Hands-On 12.18 Using the OpenSchema Method to List Database Tables

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `ListTbls2` procedure:

```
Sub ListTbls2()
    ' This procedure lists database tables using
    ' the OpenSchema method
    Dim rst As ADODB.Recordset
    Set rst = CurrentProject.Connection.OpenSchema _
        (adSchemaTables)

    Do Until rst.EOF
        Debug.Print rst.Fields("TABLE_TYPE") & " ->" _
            & rst.Fields("TABLE_NAME")
        rst.MoveNext
    Loop
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

The output of the `ListTbls2` procedure is shown below:

LINK ->Customers  
ACCESS TABLE ->MSysAccessStorage  
SYSTEM TABLE ->MSysACEs  
SYSTEM TABLE ->MSysComplexColumns  
ACCESS TABLE ->MSysNameMap  
ACCESS TABLE ->MSysNavPaneGroupCategories  
ACCESS TABLE ->MSysNavPaneGroups  
ACCESS TABLE ->MSysNavPaneGroupToObjects  
ACCESS TABLE ->MSysNavPaneObjectIDs  
SYSTEM TABLE ->MSysObjects  
SYSTEM TABLE ->MSysQueries  
SYSTEM TABLE ->MSysRelationships  
ACCESS TABLE ->MSysResources  
LINK ->mySheet

Obtaining the names of fields requires that you use `adSchemaColumns` as the parameter for the `OpenSchema` method. The `ListTblsAndFields` procedure in Hands-On 12.18 retrieves the names of fields in each table of the `Northwind_Chap12.mdb` database.



### Hands-On 12.19 Listing Tables and Their Fields Using the OpenSchema Method

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **ListTblsAndFields** procedure:

```
Sub ListTblsAndFields()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim curTable As String
    Dim newTable As String
    Dim counter As Integer

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" _
        & "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set rst = conn.OpenSchema(adSchemaColumns)
    curTable = ""
    newTable = ""
    counter = 1
    Do Until rst.EOF
        curTable = rst!table_Name
        If (curTable <> newTable) Then
            newTable = rst!table_Name
            Debug.Print "Table: " & rst!table_Name
            counter = 1
        End If
        Debug.Print "Field" & counter & ":" & _
            rst!Column_Name
        counter = counter + 1
        rst.MoveNext
    Loop
    rst.Close
    conn.Close
    Set rst = Nothing
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

The partial output generated by running the ListTblsAndFields procedure is shown below:

Table: Alphabetical List of Products

Field1: CategoryID

Field2: CategoryName

Field3: Discontinued

Field4: ProductID

Field5: ProductName

Field6: QuantityPerUnit

Field7: ReorderLevel

Field8: SupplierID

Field9: UnitPrice

Field10: UnitsInStock

Field11: UnitsOnOrder

Table: Categories

Field1: CategoryID

## **LISTING DATA TYPES**

---

The ListDataTypes procedure in Hands-On 12.20 uses the `adSchemaProviderTypes` parameter of the ADO Connection object's `OpenSchema` method to list the data types supported by the Microsoft ACE OLE DB 12.0 provider.



### **Hands-On 12.20 Listing Data Types**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **ListDataTypes** procedure shown below.

```
Sub ListDataTypes()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn=New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" _
        & "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"
    Set rst=conn.OpenSchema(adSchemaProviderTypes)
```

```
Do Until rst.EOF
    Debug.Print rst!Type_Name & vbTab -
        & "Size: " & rst!Column_Size
    rst.MoveNext
Loop

rst.Close
conn.Close
Set rst = Nothing
Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.  
The ListDataTypes procedure generates the following output:

```
Short Size: 5
Long Size: 10
Single Size: 7
Double Size: 15
Currency Size: 19
DateTime Size: 8
Bit Size: 2
Byte Size: 3
GUID Size: 16
BigBinary Size: 4000
LongBinary Size: 1073741823
VarBinary Size: 510
LongText Size: 536870910
VarChar Size: 255
Decimal Size: 28
```

## CHANGING THE AUTONUMBER

---

When you create a table in an Access database, you can assign an AutoNumber data type to a primary key field manually using the Access user interface. The AutoNumber is a unique sequential number (incremented by 1) or a random number assigned by Access whenever a new record is added to a table.

The procedure in Hands-On 12.21 opens the ADO Recordset object based on the Shippers table in the Northwind\_Chap12.mdb database, retrieves the last used AutoNumber value, and determines the current step (increment) value in effect.



### Hands-On 12.21 Changing the Value of an AutoNumber

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **ChangeAutoNumber** procedure shown here:

```
Sub ChangeAutoNumber()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strSQL As String
    Dim beginNum As Integer
    Dim stepNum As Integer

    Set conn = New ADODB.Connection
    conn.Open "Provider = Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set rst = New ADODB.Recordset
    With rst
        .CursorType = adOpenKeyset
        .LockType = adLockReadOnly
        .Open "Shippers", conn
        .MoveLast
    End With
    beginNum = rst(0)
    rst.MovePrevious
    stepNum = beginNum - rst(0)

    MsgBox "Last Auto Number Value = " & _
        beginNum & vbCrLf & _
        "Current Step Value = " & stepNum, _
        vbInformation, _
        "AutoNumber"

    rst.Close
    conn.Close
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

After defining the fields for your tables, take the time to set up primary keys, indexes, and relationships between tables. The following sections of this chapter focus on the ADOX objects that are designed to work with these features.

## CREATING A PRIMARY KEY INDEX

---

Indexes determine the order in which records are accessed from database tables and whether or not duplicate records are accepted. While indexes can speed up access to specific records in large tables, too many indexes can also slow down updates to the database. Each table in your database should include a field (or set of fields) that uniquely identifies each individual record in a table. Such a field or set of fields is called a *primary key*. A primary key is an index with its Unique and Primary properties set to True. There can be only one primary key per table.

## CREATING INDEXES USING ADO

---

In ADO, indexes are created using the `Key` object from the ADOX library. The `Type` property of the `Key` object allows you to determine whether the key is primary, foreign, or unique. For example, to create a primary key, set the `Key` object's `Type` property to `adKeyPrimary`.

The procedure in Hands-On 12.22 demonstrates how to add a primary key to the `tblFilters` table.



### Hands-On 12.22 Creating a Primary Key

1. In the Access window, press **Alt+F11** to switch to the Visual Basic Editor window.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following `Create_PrimaryKey` procedure:

```
' make sure to set up a reference to
' the Microsoft ActiveX Data Objects 6.1
' and Microsoft ADO Ext. 6.0 for DDL and Security
```

```
Sub Create_PrimaryKey()
    Dim cat As ADOX.Catalog
    Dim tbl As ADOX.Table
    Dim pKey As ADOX.Key

    On Error GoTo ErrorHandler

    Set cat = New ADOX.Catalog
    cat.ActiveConnection = CurrentProject.Connection

    Set tbl = New ADOX.Table
    tbl.Name = "tblFilters"
```

```
cat.Tables.Append tbl

With tbl.Columns
    .Append "ID", adVarWChar, 10
    .Append "Description", adVarWChar, 255
    .Append "Type", adInteger
End With

SetKey:
Set pKey = New ADOX.Key
With pKey
    .Name = "PrimaryKey"
    .Type = adKeyPrimary
End With

pKey.Columns.Append "ID"
tbl.Keys.Append pKey

Set cat = Nothing
Exit Sub

ErrorHandler:
If Err.Number = -2147217856 Then
    MsgBox "The " & tbl.Name & " is open.", _
        vbCritical, "Please close the table"
ElseIf Err.Number = -2147217857 Then
    MsgBox Err.Description
    Set tbl = cat.Tables(tbl.Name)
    Resume SetKey
ElseIf Err.Number = -2147217767 Then
    tbl.Keys.Delete pKey.Name
    Resume
Else
    MsgBox Err.Number & ":" & Err.Description
End If
End Sub
```

**4. Choose Run | Run Sub/UserForm** to execute the procedure.

The Create\_PrimaryKey procedure begins by creating a table named `tblFilters` in the currently open database and proceeds to set the primary key index on the `ID` field. If the `tblFilters` table already exists, the error handler code displays the error message and sets an object variable (`tbl`) to point to this table. The `Resume SetKey` statement refers the procedure execution to the label `SetKey`. The code that follows that label defines the primary key using the `Name` and

Type properties of the `Key` object. Next, the procedure appends the ID column to the `Columns` collection of the `Key` object, and the `Key` object itself is appended to the `Keys` collection of the table. Because errors could occur if a table is open or it already contains the primary key, the error handler is included to ensure that the procedure runs as expected.

5. Run this procedure again using by stepping through the code line by line (press **F8**).

## CREATING A SINGLE-FIELD INDEX

---

In ADO, you can add an index to a table by using the ADOX Index object. Before creating an index, make sure the table is not open and that it does not already contain an index with the same name.

To define an index, perform the following:

1. Append one or more columns to the index by using the `Append` method.
2. Set the `Name` property of the `Index` object and define other index properties, if necessary.
3. Use the `Append` method to add the `Index` object to the table's `Indexes` collection.

You can use the `Unique` property of the `Index` object to specify whether the index keys must be unique. The default value of the `Unique` property is `False`. Another property, `IndexNulls`, lets you specify whether Null values are allowed in the index. This property can be set to one of the constants shown in Table 12.5.

**TABLE 12.5** Intrinsic constants for the `IndexNulls` property of the ADOX `Index` object (see the `AllowNullsEnum` in the ADOX Library)

Constant Name	Description
<code>adIndexNullsAllow</code>	You can create an index if there is a Null value in the index field (an error will not occur).
<code>adIndexNullsDisallow</code> (This is the default value)	You cannot create an index if there is a Null value in the index field for the column (an error will occur).
<code>adIndexNullsIgnore</code>	You can create an index if there is a Null value in the index field (an error will not occur). The <code>Ignore Nulls</code> property in the <code>Indexes</code> window in the user interface will be set to Yes.
<code>adIndexNullsIgnoreAny</code> (This value is not supported by the Microsoft Jet Provider)	You can create an index if there is a Null value in the index field. The <code>Ignore Nulls</code> property in the <code>Indexes</code> window in the user interface will be set to No.

The Add\_SingleFieldIndex procedure in Hands-On 12.23 demonstrates how to add a single-field index called idxDescription to the table tblFilters.



### Hands-On 12.23 Adding a Single-Field Index to an Existing Table

This procedure uses the tblFilters table created in Hands-On 12.22.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **Add\_SingleFieldIndex** procedure

```
Sub Add_SingleFieldIndex()
    Dim cat As New ADOX.Catalog
    Dim myTbl As New ADOX.Table
    Dim myIdx As New ADOX.Index
    Dim strTblName As String

    On Error GoTo ErrorHandler

    strTblName = "tblFilters"
    cat.ActiveConnection = CurrentProject.Connection
    Set myTbl = cat.Tables(strTblName)

    With myIdx
        .Name = "idxDescription"
        .Unique = False
        .IndexNulls = adIndexNullsIgnore
        .Columns.Append "Description"
        .Columns(0).SortOrder = adSortAscending
    End With
    myTbl.Indexes.Append myIdx

    Set cat = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = -2147217856 Then
        MsgBox strTblName & " will be closed.", _
            vbCritical, "Warning: Table is Open"
        DoCmd.Close acTable, strTblName, acSaveYes
        Resume
    ElseIf Err.Number = -2147217868 Then
        myTbl.Indexes.Delete myIdx.Name
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
    End If
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

After the index properties are set, the Description column is appended to the index, and the index sort order is set to the default (`adSortAscending`). To set the index field's sort order to descending, use the `adSortDescending` constant. Next, the index is appended to the Indexes collection of the Table object.

## LISTING INDEXES IN A TABLE

---

The ADO Indexes collection contains all Index objects of a table. You can retrieve all the index names from the Indexes collection. The procedure in the next hands-on exercise demonstrates how to list the names of indexes available in the Northwind\_Chap12.mdb database's Employees table in the Immediate window.



### Hands-On 12.24 Listing Indexes in a Table

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **List\_Indexes** procedure:

```
Sub List_Indexes()
    Dim conn As New ADODB.Connection
    Dim cat As New ADOX.Catalog
    Dim tbl As New ADOX.Table
    Dim idx As New ADOX.Index

    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Open "Data Source=" & CurrentProject.Path & _
            "\Northwind_Chap12.mdb"
    End With
    cat.ActiveConnection = conn
    Set tbl = cat.Tables("Employees")

    For Each idx In tbl.Indexes
        Debug.Print idx.Name
    Next idx

    conn.Close
    Set conn = Nothing
    MsgBox "Indexes are listed in the Immediate window."
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

## DELETING TABLE INDEXES

---

Although you can delete unwanted or obsolete indexes from the Indexes window in the Access user interface, it is much faster to remove them programmatically. The procedure in Hands-On 12.25 illustrates how to delete all but the primary key index from the Customers table located in the Northwind\_Chap12.mdb database.



### Hands-On 12.25 Deleting Indexes from a Table

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **Delete\_Indexes** procedure:

```
Sub Delete_Indexes()
    ' This procedure deletes all but the primary
    ' key index from the Customers table

    Dim conn As New ADODB.Connection
    Dim cat As New ADOX.Catalog
    Dim tbl As New ADOX.Table
    Dim idx As New ADOX.Index
    Dim count As Integer

    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Open "Data Source=" & CurrentProject.Path &
              "\Northwind_Chap12.mdb"
    End With

    cat.ActiveConnection = conn
    Setup:
    Set tbl = cat.Tables("Customers")

    Debug.Print tbl.Indexes.Count
    For Each idx In tbl.Indexes
        If idx.PrimaryKey <> True Then
            tbl.Indexes.Delete (idx.Name)
            GoTo Setup
        End If
    Next idx

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

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

Notice that each time you delete an index from the table's Indexes collection you must set the reference to the table because current settings are lost when an index is deleted. Hence, the `GoTo Setup` statement sends Visual Basic to the `Setup` label to get the new reference to the Table object.

## CREATING TABLE RELATIONSHIPS

---

This section demonstrates how you can relate two tables via VBA code. We will establish the most common relationship, known as a *parent-child relationship*. In database terms, this relationship is also called a *one-to-many relationship*. We will create a Publishers table as a parent table and a Titles table as a child table. Then we will link them by a parent-child relationship. In this type of relationship, a record in the parent table can have multiple child records in the other table. In other words, when the term *one-to-many* is used, the parent is the *one* (single record) and *many* represents the children (multiple child records) in the other table.

In ADO, to establish a one-to-many relationship between tables, you'll need to perform the following steps:

1. Use the ADOX Key object to create a foreign key and set the `Type` property of the `Key` object to `adKeyForeign`. A *foreign key* consists of one or more fields in a foreign table that uniquely identify all rows in a primary table.
2. Use the `RelatedTable` property to specify the name of the related table.
3. Use the `Append` method to add appropriate columns in the foreign table to the foreign key. A foreign table is usually located on the “many” side of a one-to-many relationship and provides a foreign key to another table in a database.
4. Set the `RelatedColumn` property to the name of the corresponding column in the primary table.
5. Use the `Append` method to add the foreign key to the `Keys` collection of the table containing the primary key.

The procedure in Hands-On 12.26 illustrates how to create a one-to-many relationship between two tables: Titles and Publishers.



### Hands-On 12.26 Creating a One-to-Many Relationship

1. In the current database (Chap12.accdb), create the Titles and Publishers tables and add the fields as shown in the following table:

Table Name	Field Name	Data Type	Size
Titles	TitleID	Number	
Titles	PubID	Number	
Titles	Title	Short Text	100
Titles	Price	Currency	
Publishers	PubID	Number	
Publishers	PubName	Short Text	40
Publishers	City	Short Text	25
Publishers	Country	Short Text	25

2. Make **TitleID** the primary key for the **Titles** table and **PubID** the primary key for the **Publishers** table.
3. In the Visual Basic Editor window, choose **Insert | Module**.
4. In the module's Code window, type the **CreateTblRelation** procedure shown here:

```

Sub CreateTblRelation()
    Dim cat As New ADOX.Catalog
    Dim fKey As New ADOX.Key

    On Error GoTo ErrorHandler

    cat.ActiveConnection = CurrentProject.Connection

    With fKey
        .Name = "fkPubID"
        .Type = adKeyForeign
        .RelatedTable = "Publishers"
        .Columns.Append "PubID"
        .Columns("PubID").RelatedColumn = "PubID"
    End With
    cat.Tables("Titles").Keys.Append fKey
    MsgBox "Relationship was created."

    Set cat = Nothing
    Exit Sub

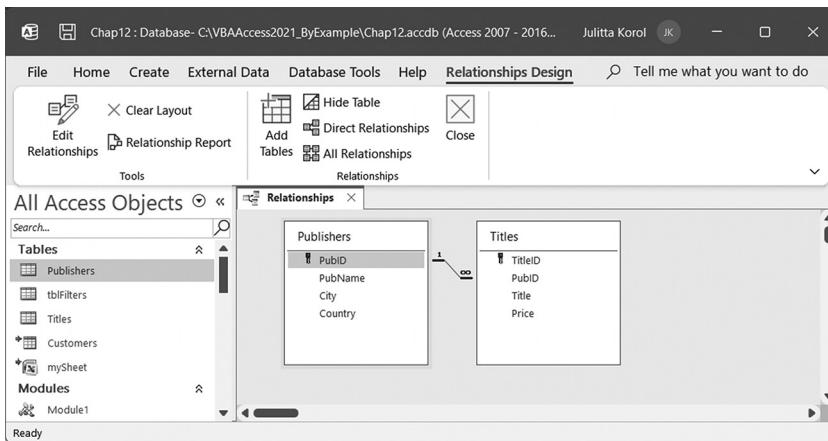
ErrorHandler:
    cat.Tables("Titles").Keys.Delete "fkPubID"
    Resume
End Sub

```

5. Choose **Run | Run Sub/UserForm** to execute the procedure.

If you receive an error while running this procedure, make sure that both tables are closed.

You can view the relationship between the Publishers and Titles tables that was created by the CreateTblRelation procedure in the Relationships window. To activate this window, switch to the Access application window and choose Database Tools | Relationships. You should see the Publishers and Titles tables in the Relationships window linked with a one-to-many relationship (see Figure 12.5).



**FIGURE 12.5** The one-to-many relationship between the Publishers and Titles tables was created programmatically by accessing objects in the ADOX library (see the code in the CreateTblRelation procedure in Hands-On 12.25).

## INTRODUCTION TO ADO RECORDSETS

---

The `Recordset` object is one of the three most-used ADO objects (the other two are `Connection` and `Command`). You can open an ADO Recordset by using the `Recordset` object's `Open` method. The information needed to open a recordset can be provided by first setting properties and then calling the `Open` method, or by using the `Open` method's parameters like this:

```
rst.Open [Source], [ActiveConnection], [CursorType], [LockType],
[CursorLocation], [Options]
```

Notice that all the parameters are optional (they appear in square brackets). If you decide that you don't want to pass parameters, then use a different syntax to open a recordset. For example, examine the following code block:

```
With rst
    .Source = strSQL
    .ActiveConnection = strConnect
    .CursorType = adOpenStatic
    .LockType = adLockOptimistic
    .CursorLocation = adUseClient
    .Open Options := adCmdText
End with
```

The preceding code segment opens a recordset by first setting properties of the Recordset object, then calling its `Open` method. Notice that the names of the required Recordset properties are equivalent to the parameter names listed earlier. The values assigned to each property are discussed later. You will become familiar with both methods of opening a recordset as you work with the example procedures that follow.

Let's return to the syntax of the recordset's `Open` method, which specifies the parameters. Needless to say, you need to know what each parameter is and how it is used. The `Source` parameter determines where you want your records to come from. The data source can be an SQL string, a table, a query, a stored procedure or view, a saved file, or a reference to a `Command` object. Later in this chapter you will learn how to open a recordset based on a table, a query, and an SQL statement.

The `ActiveConnection` parameter can be an SQL string that specifies the connection string or a reference to a `Connection` object. This parameter tells where to find the database as well as what security credentials to use.

Before we discuss the next three parameters, you need to know that the ADO recordsets are controlled by a cursor. The *cursor* determines whether the recordset is scrollable (backward and forward or forward only), whether it is read-only or updatable, and whether changes made to the data are visible to other users.

The ADO cursors have three functions specified by the following parameters:

- `CursorType`
- `LockType`
- `CursorLocation`

Before you choose the cursor, you need to think of how your application will use the data. Some cursors yield better performance than others. It's important to determine where the cursor will reside and whether changes made while the cursor is open need to be visible immediately. The following subsection should assist you in choosing the correct cursor.

## Cursor Types

The `CursorType` parameter specifies how the recordset interacts with the data source and what is allowed or not allowed when it comes to data changes or movement within the recordset. This parameter can take one of four constants: `adOpenForwardOnly` (0), `adOpenKeyset` (1), `adOpenDynamic` (2), and `adOpenStatic` (3).

You can find out what types of cursors are available by using the Object Browser. Before proceeding, check that the database database file you are working with has a reference to the ActiveX Data Objects library. Set this reference by switching to the Visual Basic Editor window and choosing Tools | References. Find and select Microsoft ActiveX Data Objects 6.1 Library in the References dialog box and click OK. Next, activate the Object Browser window by pressing F2 or choose View | Object Browser. Select ADODB from the Project/Library drop-down listbox and type `CursorType` in the Search text box, as shown in Figure 12.6.

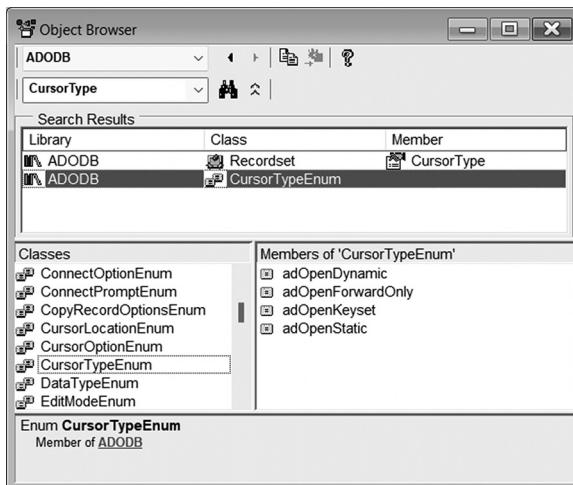


FIGURE 12.6. The Object Browser lists four predefined constants you can use to specify the cursor type to be retrieved.

- When the cursor type is dynamic (`adOpenDynamic`), users are allowed to view changes other users made to the database. The dynamic cursor is not supported by the Jet 4.0 engine in Microsoft Access. To use this cursor, you must use other OLE DB providers, such as MSDASQL or SQLOLEDB. When you use the dynamic cursor, you can move back and forth in the recordset.

- When the cursor type is forward-only (`adOpenForwardOnly`), additions, changes, or deletions made by other users are not visible. This is both the default and the fastest cursor because it only allows you to scroll forward in the recordset.
- When the cursor type is keyset driven (`adOpenKeyset`), you can scroll back and forth in the recordset; however, you cannot view records added or deleted by another user. Use the Recordset's `Requery` method to overcome this limitation.
- When the cursor type is static (`adOpenStatic`), all the data is retrieved as it was at a point in time. This cursor is desirable when you need to find data or generate a report. You can scroll back and forth within a recordset, but additions, changes, or deletions by other users are not visible. Use this cursor to retrieve an accurate record count.
- You must set the `CursorType` before opening the recordset with the `Open` method. Otherwise, Access will create a Forward-only recordset. You may use a constant name or its value in your VBA procedures.

### Lock Types

After you choose a cursor type, it is important to specify how the ADO should lock the row when you make a change. The `LockType` specifies whether the recordset is updatable. The default setting for `LockType` is read-only. The `LockType` predefined constants are listed in the Object Browser, as shown in Figure 12.7.

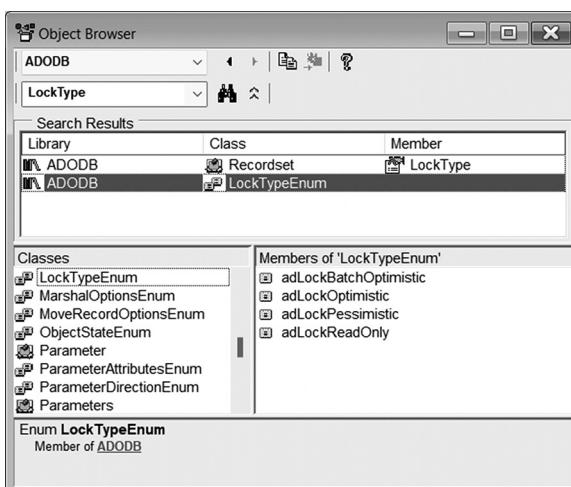


FIGURE 12.7 The Object Browser lists four predefined constants that you can use to specify what type of locking ADO should use when you make a change to the data.

When the `LockType` property is batch optimistic (`adLockBatchOptimistic`), batch updates made to the data are stored locally until the `UpdateBatch` method is called, during which all pending updates are committed all at once. Until the `UpdateBatch` method is called, no locks are placed on edited data. Batch optimistic locking eliminates network roundtrips that normally occur with optimistic locking (`adLockOptimistic`) when users make changes to one record and move to another. With batch optimistic locking, a user can make all the changes to all the records and then submit them as a single operation.

- When the `LockType` property is optimistic (`adLockOptimistic = 3`), no locks are placed on the data until you attempt to save a row. Records are locked only when you call the `Update` method, and the lock is released as soon as the Save operation is completed. Two users are allowed to update a record at the same time. Optimistic locking allows you to work with one row at a time. If you need to make multiple updates, it's better to save them all at once by using batch optimistic locking.
- When the `LockType` property is pessimistic (`adLockPessimistic = 2`), all the records are locked as soon as you begin editing a record. The record remains locked until the edit is committed or canceled. This type of lock guarantees that two users will not make changes to the same record. If you use pessimistic locking, ensure that your code does not require any input from the users. You certainly don't want a scenario where a user opens a record and makes a change, then leaves for lunch without saving the record. In that case, the record is locked until the user comes back and saves or discards the edit. In this situation, it is better to use optimistic locking.
- When the `LockType` property is read-only (`adLockReadOnly = 1`), you will not be able to alter any data. This is the default setting.

## Cursor Location

---

The `CursorPosition` parameter determines whether ADO or the SQL Server database engine manages the cursor. Cursors use temporary resources to hold the data. These resources can be memory, a disk paging file, temporary disk files, or even temporary storage in the database.

- When a cursor is created and managed by ADO, the recordset is said to be using a *client-side cursor* (`adUseClient`). With the client-side cursor, all the data is retrieved from the server in one operation and is placed on the client computer. Because all the requested data is available locally, the

connection to the database can be closed and reopened only when another set of data is needed. Since the entire result set has been downloaded to the client computer, browsing through the rows is very fast.

- When a cursor is managed by a database engine, the recordset is said to be using a *server-side cursor* (`adUseServer`). With the server-side cursor, all the data is stored on the server and only the requested data is sent over the network to the user's computer. This type of cursor can provide better performance than the client-side cursor when excessive network traffic is an issue. However, it's important to point out that a server-side cursor consumes server resources for every active client and, because it provides only single-row access to the data, it can be quite slow.

It is recommended that you use the server-side cursor when working with local Access databases, and the client-side cursor when working with remote Access databases or SQL Server databases.

The `CursorLocation` predefined constants are listed in the Object Browser, as shown in Figure 12.8.

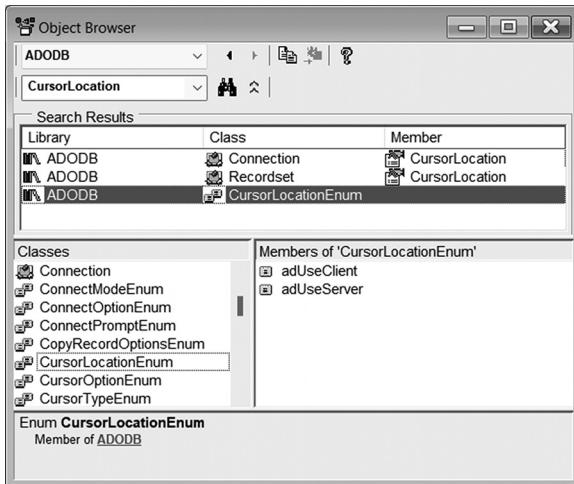


FIGURE 12.8. The `CursorLocation` parameter of the Recordset's Open method can be set by using the `adUseClient` or `adUseServer` constant.

### The Options Parameter

The `Options` parameter specifies the data source type being used. Similar to the parameters related to cursors, the `Options` parameter can take one of many values, as shown in Figure 12.9.

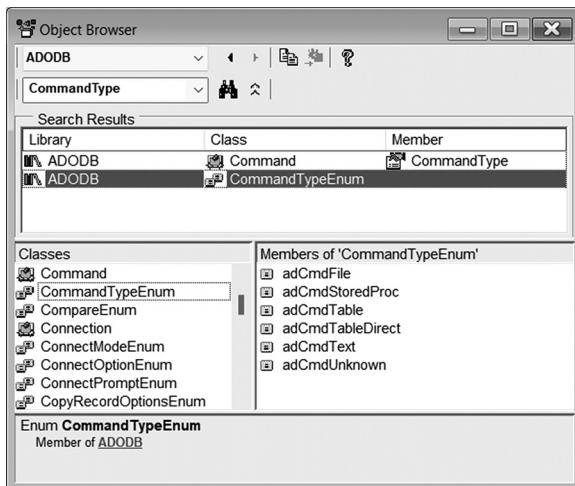


FIGURE 12.9 The Options parameter of the Recordset's Open method is supplied by the constant values listed under the CommandType property of the Command object.

- When the `Options` parameter is set to `adCmdFile` (256), it tells the ADO that the source of the recordset is a path or filename. ADO can open recordsets based on files in different formats.
- When the `Options` parameter is set to `adCmdStoredProc` (4), it tells the ADO that the source of the recordset is a stored procedure or parameterized query.
- When the `Options` parameter is set to `adCmdTable` (2), it tells the ADO that the source of the recordset is a table or view. The `adCmdTable` constant will cause the provider to generate an SQL query to return all rows from a table or view by prepending `SELECT * FROM` in front of the specified table or view name.
- When the `Options` parameter is set to `adCmdTableDirect` (512), it tells the ADO that the `Source` argument should be evaluated as a table name. How does this constant differ from `adCmdTable`? The `adCmdTableDirect` constant is used by OLE DB providers that support opening tables directly by name, using an interface called `IOpenRowset` instead of an ADO Command object. Since the `IOpenRowset` method does not need to build and execute a Command object, its use results in increased performance and functionality.
- When the `Options` parameter is set to `adCmdText` (1), it tells the ADO that you are using an SQL statement to open the recordset.

- When the `Options` parameter is set to `adCmdUnknown` (8), it tells the ADO that the command type in the `Source` argument is unknown. This is the default, which is used if you don't specify any other option. By using the `adCmdUnknown` constant, or not specifying any constant at all for the `Options` parameter, you force ADO to make an extra roundtrip to the server to determine the source type. As you would expect, this will decrease your VBA procedure's performance; therefore, you should use `adCmdUnknown` only if you don't know what type of information the `Source` parameter will contain.

**NOTE**

*Not all options are supported by all data providers. For example, Microsoft Jet OLE DB Provider does not support the `adCmdTableDirect` cursors.*

In addition to specifying the type of `CommandType` in the `Options` parameter (see Figure 12.9), you can pass additional information in the `Options` parameter. For example, you can tell ADO how to execute the command by specifying whether ADO should wait while all the records are being retrieved or should continue asynchronously.

**SIDE BAR** ***Asynchronous Record Fetching***

Asynchronous fetching is an ADO feature that allows some records to be downloaded to the client while the remaining records are still being fetched from the database. As soon as the user sees some records, he can begin paging through them. The user does not know that only a few records have been returned. As he pages through the rows backward and forward, a new connection is made to the server and more records are fetched and passed to the client's computer. Once all records have been returned, paging is very quick because all records are on the client. Asynchronous fetching makes it seem to the user that the data retrieval is pretty fast. The downside is that records cannot be sorted until they have all been downloaded.

Additional `Options` parameters are described in the following list. Note that only the first three constants (`adAsyncExecute`, `adAsyncFetch`, and `adAsyncFetchNonBlocking`) can be used with the Recordset's `Open` method. Other constants are used with the Command or Connection object's `Execute` method.

- `adAsyncExecute` (16)—This tells ADO to execute the command asynchronously, meaning that all requested rows are retrieved as soon as they

are available. Using `adAsyncExecute` enables the application to perform other tasks while waiting for the cursor to populate.

Note that the `adAsyncExecute` constant cannot be used with `adCmdTableDirect`.

- `adAsyncFetch` (32)—Using this constant requires that you specify a value greater than 1 for the recordset's `CacheSize` property. The `CacheSize` property is used to determine the number of records ADO will hold in local memory. For example, if the cache size is 100, the provider will retrieve the first 100 records after first opening the `Recordset` object. The `adAsyncFetch` constant tells ADO that the rows remaining after the initial quantity specified in the `CacheSize` property should be retrieved asynchronously.
- `adAsyncFetchNonBlocking` (64)—This option tells ADO that it should never wait for a row to be fetched. The application will continue execution while records are being continuously extracted from a very large data file. If the requested recordset row has not been retrieved yet, the current row automatically moves to the end of the file (causing the recordset's `EOF` property to become `True`). In other words, the data retrieval process will not block other processes.

Note that `adAsyncFetchNonBlocking` has no effect when the `adCmdTableDirect` option is used to open the recordset. Also, `adAsyncFetchNonBlocking` is not supported with a Server cursor (`adUseServer`) when you use the ODBC provider (MSDASQL).

- `adExecuteNoRecords` (128)—This option tells ADO not to expect any records when the command is executed. Use this option for commands that do not return records, such as `INSERT`, `UPDATE`, or `DELETE`. Use the `adExecuteNoRecords` constant with `adCmdText` to improve the performance of your application. When this option is specified, ADO does not create a `Recordset` object and does not set any cursor properties.

Note that `adExecuteNoRecords` can only be passed as an optional parameter to the `Command` or `Connection` object's `Execute` method and cannot be used when opening a recordset.

- `adExecuteStream` (256)—Indicates that the results of a `Command` execution should be returned as a stream. The `adExecuteStream` constant can only be passed as an optional parameter to the `Command` or `Connection` object's `Execute` method and it cannot be used when opening a

recordset.

- adExecuteRecord (512)—Indicates that the value of the CommandText property is a command or stored procedure that returns a single row as a Record object (a Record object represents one row of data).
- adOptionUnspecified (-1)—Indicates that the command is unspecified. This is the default option.

Note that similar to adExecuteNoRecords, adExecuteStream, and adExecuteRecord, this constant can only be passed as an optional parameter to the Command or Connection object's Execute method and cannot be used when opening a recordset.

## Opening a Recordset

---

ADO offers numerous ways of opening a Recordset object. To begin with, you can create ADO recordsets from scratch without going through any other object. Suppose you want to retrieve all the records from the Employees table. The code you need to write is very simple. Let's try this out in Hands-On 12.27.



### Hands-On 12.27 Opening a Recordset

1. Create a copy of the Northwind 2007.accdb database and save it as **Northwind 2007\_Chap12.accdb** in your C:\VBAAccess2021\_ByExample folder.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following OpenADORst procedure:

```
' make sure to set up a reference to
' the Microsoft ActiveX Data Objects 6.1 Library

Sub OpenADORst()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Open "Data Source=" & CurrentProject.Path & _
               "\Northwind 2007_Chap12.accdb"
    End With

    Set rst = New ADODB.Recordset
    With rst
        .Source = "SELECT * FROM Employees"
```

```

    .ActiveConnection = conn
    .Open
    Debug.Print rst.Fields.Count
    .Close
End With

Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub

```

**4. Choose Run | Run Sub/UserForm** to execute the procedure.

In the preceding code example, we first define and open a connection to the database. Next, we declare a Recordset object and create a new instance of it. The Recordset object's Source property specifies the data you want to retrieve. The source can be a table, query, stored procedure, view, saved file, or Command object. The SQL SELECT statement tells VBA to select all the data from the Employees table. Next, the ActiveConnection property specifies how to connect to the data. We set the ActiveConnection property to the object variable (`conn`) that holds the connection information. Finally, the `Open` method retrieves the specified records into the recordset. Before we close the recordset using the Recordset's `Close` method, we retrieve the number of fields in the open recordset by examining the Recordset's Fields collection and write the result to the Immediate window.

### Opening a Recordset Based on a Table or Query

A recordset can be based on a table, view, SQL statement, or command that returns rows. It can be opened via a `Connection` or `Command` object's `Execute` method or a `Recordset`'s `Open` method (see the following example procedures).

- Using the `Execute` method of the `Connection` object:

```

Sub ConnectAndExec()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim fld As Variant

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind 2007_Chap12.accdb"
    Set rst = conn.Execute("SELECT * FROM Employees")
    Debug.Print rst.Source

```

```

Do Until rst.EOF
    Debug.Print "||||||||||||||||||||||||"
    For Each fld In rst.Fields
        Debug.Print fld.Name & "=" & fld.Value
    Next
    'Debug.Print "---new record ---"
    rst.MoveNext
Loop
'Debug.Print rst.Fields(1).Value
rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing

End Sub

```

**NOTE**

*Once you open the recordset, you can perform the required operation on its data. In this example, we use the Recordset's Source property to write to the Immediate window the SQL command on which the recordset is based. Next, we loop through the recordset to retrieve the contents of each field in every record.*

- Using the `Execute` method of the Command object:

```

Sub CommandAndExec()
    Dim conn As ADODB.Connection
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    With conn
        .ConnectionString = _
            "Provider=Microsoft.ACE.OLEDB.12.0;" & _
            "Data Source=" & CurrentProject.Path & _
            "\Northwind 2007\Chap12.accdb"
        .Open
    End With

    Set cmd = New ADODB.Command
    With cmd
        .ActiveConnection = conn
        .CommandText = "SELECT * FROM Customers"
    End With

    Set rst = cmd.Execute

```

```
MsgBox rst.Fields(1).Value

rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

**NOTE**

*Once you open the recordset, you can perform the required operation on its data. In this example, we display a message with the name of the first customer.*

- Using the Open method of the Recordset object:

```
Sub RecSetOpen()
    Dim rst As ADODB.Recordset
    Dim strConnection As String

    strConnection =
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind 2007_Chap12.accdb"

    Set rst = New ADODB.Recordset
    With rst
        .Open "SELECT * FROM Customers", _
            strConnection, adOpenForwardOnly
        .Save CurrentProject.Path & "\MyRst.dat"
        .Close
    End With
    Set rst = Nothing
End Sub
```

**NOTE**

*Once you open the recordset, you can perform the required operation on its data. In this example, we save the entire recordset to a disk file named MyRst.dat. Later in this chapter you will learn how to work with records that have been saved in a file like that.*

The procedure in Hands-On 12.28 illustrates how to open a recordset based on a table or query.



### Hands-On 12.28 Opening a Recordset Based on a Table or Query

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **OpenRst\_TableOrQuery** procedure:

```
Sub OpenRst_TableOrQuery()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Open "Data Source=" & CurrentProject.Path & _
            "\Northwind 2007_Chap12.accdb"
    End With

    Set rst = New ADODB.Recordset
    rst.Open "Employees", conn

    Debug.Print "CursorType: " & _
        rst.CursorType & vbCrLf & _
        "LockType: " & rst.LockType & vbCrLf & _
        "Cursor Location: " & rst.CursorLocation

    Do Until rst.EOF
        Debug.Print rst.Fields(2)
        rst.MoveNext
    Loop

    rst.Close
    Set rst = Nothing
    conn.Close
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

After opening the recordset, it's a good idea to check what type of recordset was created. Notice that this procedure uses the `CursorType`, `LockType`, and `CursorLocation` properties to retrieve this information. After the procedure is run, the Immediate window displays the following:

```
CursorType: 0
LockType: 1
Cursor Location: 2
```

```
Freehafer
Cencini
Kotas
Sergienko
Thorpe
Neipper
Zare
Giussani
Hellung-Larsen
```

Notice that because you did not specify any parameters in the Recordset's `Open` method, you obtained a default recordset. This recordset is forward-only (0), read-only (1), and server-side (2).

To create a different type of recordset, pass the appropriate parameters to the Recordset's `Open` method. For example, if you open your recordset like this:

```
rst.Open "Employees", conn, adUseClient, adLockReadOnly
```

you will get the static (3), read-only (1), and client-side (3) recordset. In this recordset, you can easily find out the number of records by using the Recordset's `RecordCount` property:

```
Debug.Print rst.RecordCount
```

Next, this procedure uses the `MoveNext` method to iterate through all the records in the recordset until the end of file (EOF) is reached.

---

**SIDE BAR** *Counting Records*

Use the Recordset object's `RecordCount` property to determine the number of records in a recordset. If the number of records cannot be determined, this property will return -1. The `RecordCount` property setting depends on the cursor type and the capabilities of the provider. To get the actual count of records, open the recordset with the static (`adOpenStatic`) or dynamic (`adOpenDynamic`) cursor.

---

To quickly test the contents of the recordset, we write the employees' last names to the Immediate window. Since this recordset contains all the fields in the Employees table, you can add extra code to list the remaining field values.

---

**SIDE BAR** *Is This Recordset Empty?*

A recordset may be empty. To check whether your recordset has any records in it, use the Recordset object's `BOF` and `EOF` properties. The `BOF` property stands for "beginning of file," and `EOF` indicates "end of file."

- If you open a Recordset object that contains no records, the `BOF` and `EOF` properties are both set to `True`.
- If you open a Recordset object that contains at least one record, the `BOF` and `EOF` properties are `False` and the first record is the current record.

You can use the following conditional statement to test whether there are any records:

```
If rst.BOF And rst.EOF Then  
    MsgBox "This recordset contains no records"  
End If
```

To open a recordset based on a saved query, replace the table name with your query name.

---

### ***Opening a Recordset Based on an SQL Statement***

The procedure in Hands-On 12.29 demonstrates how to use the `Connection` object's `Execute` method to open a recordset based on an SQL statement that selects all the employees from the `Employees` table in the sample `Northwind 2007_Chap12.accdb` database. Only the name of the first employee is written to the Immediate window. As in the preceding example, the resulting recordset is forward-only and read-only.

#### **Hands-On 12.29   Opening a Recordset Based on an SQL Statement**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the `CreateRst_WithSQL` procedure shown here:

```
Sub CreateRst_WithSQL()  
    Dim conn As ADODB.Connection  
    Dim rst As ADODB.Recordset  
    Dim strConn As String  
  
    strConn = _  
        "Provider = Microsoft.ACE.OLEDB.12.0;" & _  
        "Data Source=" & CurrentProject.Path & _  
        "\Northwind 2007_Chap12.accdb"  
  
    Set conn = New ADODB.Connection  
    conn.Open strConn  
  
    Set rst = conn.Execute _
```

```
("SELECT * FROM Employees")
Debug.Print rst("Last Name") & _
", " & rst("First Name")

rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### ***Opening a Recordset Based on Criteria***

---

Instead of retrieving all the records from a specific table or query, you can use the SQL `WHERE` clause to get only those records that meet certain criteria. The procedure in Hands-On 12.30 calls the Recordset's `Open` method to create a forward-only and read-only recordset populated with employees who are sales representatives.



### **Hands-On 12.30   Opening a Recordset Based on Criteria**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **OpenRst\_WithCriteria** procedure:

```
Sub OpenRst_WithCriteria()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strConn As String

    strConn =
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind 2007_Chap12.accdb"

    Set conn = New ADODB.Connection
    conn.Open strConn

    Set rst = New ADODB.Recordset
    rst.Open "SELECT * FROM Employees WHERE " & _
        "[Job Title] = " & _
        "'Sales Representative'", _
        conn, adOpenForwardOnly, adLockReadOnly
```

```
Do While Not rst.EOF
    Debug.Print rst.Fields(2).Value
    rst.MoveNext
Loop

rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### ***Opening a Recordset Directly with ADO***

---

If you are planning to open just one recordset from a specific data source, you can take a shortcut and open it directly without first opening a Connection object. This method requires you to specify the source and connection information prior to calling the Recordset object's `Open` method, as shown in Hands-On 12.31.



#### **Hands-On 12.31    Opening a Recordset Directly**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the `OpenRst_Directly` procedure shown here:

```
Sub OpenRst_Directly()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Open "Data Source=" &
            CurrentProject.Path &
            "\Northwind 2007_Chap12.accdb"
    End With

    Set rst = New ADODB.Recordset
    With rst
        .Source = "SELECT * FROM Employees"
        .ActiveConnection = conn
        .Open
    End With
    MsgBox rst.Fields(2)
```

```
rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### Moving Around in a Recordset

---

You can navigate the ADO Recordset by using the following five methods: **MoveFirst**, **MoveLast**, **MoveNext**, **MovePrevious**, and **Move**. The procedure in Hands-On 12.32 demonstrates how to move around in a recordset and retrieve the names of fields and their contents for each record.



### Hands-On 12.32 Moving Around in a Recordset

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **MoveAround** procedure:

```
Sub MoveAround()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim fld As ADODB.Field
    Dim strConn As String

    strConn = _
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set conn = New ADODB.Connection
    conn.Open strConn

    Set rst = New ADODB.Recordset
    rst.Open "SELECT * FROM Customers WHERE " & _
        "ContactTitle = 'Owner'", _
        conn, adOpenForwardOnly, adLockReadOnly
    Do While Not rst.EOF
        Debug.Print "New Record -----"
        For Each fld In rst.Fields
            Debug.Print fld.Name & " = " & _
            fld.Value
        Next
    rst.MoveNext
```

```
Loop

    rst.Close
    Set rst = Nothing
    conn.Close
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### Finding the Record Position

---

Use the `AbsolutePosition` property of the `Recordset` object to determine the current record number. This property specifies the relative position of a record in an ADO Recordset. The procedure in Hands-On 12.33 opens a recordset filled with employee records from the `Employees` table in the Northwind database and uses the `AbsolutePosition` property to return the record number three times during the procedure execution.



### Hands-On 12.33 Finding the Record Position

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **FindRecordPosition** procedure:

```
Sub FindRecordPosition()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strConn As String

    strConn =
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set conn = New ADODB.Connection
    conn.Open strConn

    Set rst = New ADODB.Recordset
    With rst
        .Open "SELECT * FROM Employees", _
            conn, adOpenKeyset, _
            adLockOptimistic, adCmdText
        Debug.Print .AbsolutePosition
        .Move 3 ' move forward 3 records
    End With
End Sub
```

```
Debug.Print .AbsolutePosition  
.MoveLast ' move to the last record  
Debug.Print .AbsolutePosition  
Debug.Print .RecordCount  
.Close  
End With  
  
Set rst = Nothing  
conn.Close  
Set conn = Nothing  
End Sub
```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

Notice that at the beginning of the recordset, the record number is 1. Next, the `FindRecordPosition` procedure uses the `Move` method to move the cursor three rows ahead, after which the `AbsolutePosition` property returns 4 (1 + 3) as the current record position. Finally, the `MoveLast` method is used to move the cursor to the end of the recordset. The `AbsolutePosition` property now determines that this is the ninth record (9). The `RecordCount` property of the `Recordset` object returns the total number of records (9).

## Reading Data from a Field

---

Use the `Fields` collection of a `Recordset` object to retrieve the value of a specific field in an open recordset. The procedure in Hands-On 12.34 uses the `Do...While` loop to iterate through the recordset and prints the names of all the employees to the Immediate window.



### Hands-On 12.34 Retrieving Field Values

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **ReadField** procedure:

```
Sub ReadField()  
    Dim conn As ADODB.Connection  
    Dim rst As ADODB.Recordset  
  
    Set conn = New ADODB.Connection  
    With conn  
        .Provider = "Microsoft.ACE.OLEDB.12.0"  
        .Open "Data Source=" & _  
            CurrentProject.Path & _  
            "\Northwind 2007_Chap12.accdb"  
    End With
```

```
Set rst = New ADODB.Recordset
rst.Open "SELECT * FROM Employees", _
    conn, adOpenStatic

Do While Not rst.EOF
    Debug.Print rst.Fields("Last Name").Value
    rst.MoveNext
Loop

rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### **Returning a Recordset as a String**

---

Instead of using a loop to read the values of fields in all rows of the open recordset, you can use the Recordset object's `GetString` method to get the desired data in one step. The `GetString` method returns a recordset as a string-valued Variant. This method has the following syntax:

```
Variant = Recordset.GetString(StringFormat, NumRows, _
    ColumnDelimiter, RowDelimiter, NullExpr)
```

- The first argument (`StringFormat`) determines the format for representing the recordset as a string. Use the `adAddClipString` constant as the value for this argument.
- The second argument (`NumRows`) specifies the number of recordset rows to return. If blank, `GetString` will return all the rows.
- The third argument (`ColumnDelimiter`) specifies the delimiter for the columns within the row (the default column delimiter is tab (`vbTab`)).
- The fourth argument (`RowDelimiter`) specifies a row delimiter (the default is carriage return (`vbCrLf`)).
- The fifth argument (`NullExpr`) specifies an expression to represent Null values (the default is an empty string ("")).



### Hands-On 12.35 Converting the Recordset to a String

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **GetRecordsAsString** procedure shown here:

```
Sub GetRecordsAsString()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim varRst As Variant
    Dim fso As Object
    Dim myFile As Object
    Dim strSQL As String

    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Open "Data Source=" & _
            CurrentProject.Path & _
            "\Northwind_Chap12.mdb"
    End With

    Set rst = New ADODB.Recordset

    strSQL = "SELECT EmployeeId, "
    strSQL = strSQL & "LastName & "" "" & FirstName "
    strSQL = strSQL & "AS FullName "
    strSQL = strSQL & "From Employees"

    rst.Open strSQL, conn, adOpenForwardOnly, _
        adLockReadOnly, adCmdText

    If Not rst.EOF Then
        ' Return all rows as a formatted string with
        ' columns delimited by Tabs, and rows
        ' delimited by carriage returns

        varRst = rst.GetString(adClipString, , _
            vbTab, vbCrLf)
        Debug.Print varRst
    End If

    ' save the recordset string to a text file
    Set fso = CreateObject _
        ("Scripting.FileSystemObject")
```

```
Set myFile = fso.CreateTextFile _
    (CurrentProject.Path & _
    "\RstString.txt", True)
myFile.WriteLine varRst
myFile.Close

Set fso = Nothing
rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

The `GetRecords_AsString` procedure demonstrates how you can transform a recordset into a tab-delimited list of values using the `Recordset` object's `GetString` method. You can use any characters you want to separate columns and rows. This procedure uses the following statement to convert a recordset to a string:

```
varRst = rst.GetString(adClipString, , vbTab, vbCrLf)
```

Notice that the second argument is omitted. This indicates that we want to obtain all the records. To convert only three records to a string, you could write the following line of code:

```
varRst = rst.GetString(adClipString, 3, vbTab, vbCrLf)
```

The `vbTab` and `vbCrLf` arguments are VBA constants that denote the Tab and carriage return characters.

Because `adClipString`, `vbTab`, and `vbCrLf` are default values for the `GetString` method's arguments, you can skip them altogether. Therefore, to put all of the records in this recordset into a string, you can simply use the `GetString` method without arguments, like this:

```
varRst = rst.GetString
```

Sometimes you may want to save your recordset string to a file. To gain access to a computer's filesystem, the procedure uses the `CreateObject` function to access the `FileSystemObject` from the Microsoft Scripting Runtime Library. You can easily create a `File` object by using the `CreateTextFile` method of this object. Notice that the second argument of the `CreateTextFile` method (`True`) indicates that the file should be overwritten if it already exists. Once

you have defined your file, you can use the `WriteLine` method of the `File` object to write the text to the file. In this example, your text is the variable holding the contents of a recordset converted to a string.

### Finding Records Using the Find Method

---

The ADO Object Model provides you with two methods for locating records: `Find` and `Seek`. This section demonstrates how to use the ADO `Find` method to locate all the employee records based on a condition. ADO has a single `Find` method. The search always begins from the current record or an offset from it. The search direction and the offset from the current record are passed as parameters to the `Find` method. The `SearchDirection` parameter can be either `adSearchForward` or `adSearchBackward`.



#### Hands-On 12.36 Finding Records Using the Find Method

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `Find_WithFind` procedure:

```
Sub Find_WithFind()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    conn.Open _
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set rst = New ADODB.Recordset
    rst.Open "Employees", conn, _
        adOpenKeyset, adLockOptimistic

    ' find the first record matching
    ' the criteria
    rst.Find "TitleOfCourtesy ='Ms.'"
    Do Until rst.EOF
        Debug.Print rst.Fields("LastName").Value
        ' search forward starting from
        ' the next record
        rst.Find "TitleOfCourtesy ='Ms.'", _
            SkipRecords:=1, _
            SearchDirection:=adSearchForward
    Loop
```

```

rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub

```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

To find the last record, call the `MoveLast` method before using `Find`. If none of the records meets the criteria, the current record is positioned before the beginning of the recordset (if searching forward) or after the end of the recordset (if searching backward). You can use the `EOF` or `BOF` properties of the Recordset object to determine whether a matching record was found.

**NOTE**

*The ADO Find method does not support the Is operator. To locate a record that has a Null value, use the equal sign (=). For example:*

```

' find records that do not have
' an entry in the ReportsTo field

rst.Find "ReportsTo = Null"

' find records that have data
' in the ReportsTo field
rst.Find " ReportsTo <> Null"

```

To find records based on more than one condition, use the `Filter` property of the Recordset object, as shown later in this chapter.

### Finding Records Using the Seek Method

You can use the Recordset object's `Seek` method to locate a record based on an index. If you don't specify the index before searching, the primary key will be used. If the record is found, the current row position is changed to that row. The syntax of the `Seek` method looks like this:

```
recordset.Seek KeyValues, SeekOption
```

The first argument of the `Seek` method specifies the key values you want to find. The second argument specifies the type of comparison to be made between the columns of the index and the corresponding `KeyValues`.

The procedure in Hands-On 12.37 uses the `Seek` method to find the first company with an entry in the `Region` field equal to "SP":

```
rst.Seek "SP", adSeekFirstEQ
```

To find the last record that meets the same condition, use the following statement:

```
rst.Seek "SP", adSeekLastEQ
```

The type of `Seek` to execute is specified by the constants shown in Table 12.6.

**TABLE 12.6** Seek method constants

Constant	Value	Description
adSeekFirstEQ	1	Seeks the first key equal to <code>KeyValues</code>
adSeekLastEQ	2	Seeks the last key equal to <code>KeyValues</code>
adSeekAfterEQ	4	Seeks a key either equal to <code>KeyValues</code> or just after where that match would have occurred
adSeekAfter	8	Seeks a key just after where a match with <code>KeyValues</code> would have occurred
adSeekBeforeEQ	16	Seeks a key either equal to <code>KeyValues</code> or just before where that match would have occurred
adSeekBefore	32	Seeks a key just before where a match with <code>KeyValues</code> would have occurred

The `Seek` method is recognized only by the Microsoft Jet 4.0/ACE 12.0 databases. To determine whether the `Seek` method can be used to locate a row in a recordset, use the `Recordset` object's `Supports` method. This method determines whether a specified `Recordset` object supports a particular type of feature. The Boolean value of `True` indicates that the feature is supported; `False` indicates that it is not.

```
' find out if the recordset
' supports the Seek method
```

```
MsgBox rst.Supports(adSeek)
```



### Hands-On 12.37 Finding Records Using the Seek Method

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `Find_WithSeek` procedure:

```
Sub Find_WithSeek()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    conn.Open _
```

```
"Provider=Microsoft.ACE.OLEDB.12.0;" & _  
"Data Source=" & CurrentProject.Path & _  
"\Northwind.mdb"  
  
Set rst = New ADODB.Recordset  
With rst  
    .Index = "Region"  
    .Open "Customers", conn, adOpenKeyset, _  
        adLockOptimistic, adCmdTableDirect  
  
    ' find out if this recordset  
    ' supports the Seek method  
    MsgBox rst.Supports(adSeek)  
    .Seek "SP", adSeekFirstEQ  
End With  
  
If Not rst.EOF Then  
    Debug.Print rst.Fields _  
        ("CompanyName").Value  
End If  
  
rst.Close  
Set rst = Nothing  
conn.Close  
Set conn = Nothing  
End Sub
```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

If the Seek method is based on a multifield index, use the VBA `Array` function to specify values for the `KeyValues` parameter. For example, the Order Details table in the Northwind\_Chap12.mdb database uses a multifield index as the PrimaryKey. This index is a combination of the OrderID and ProductID fields. To find the order in which OrderID = 10295 and ProductID = 56, use the following statement:

```
rst.Seek Array(10295, 56), adSeekFirstEQ
```

## Finding a Record Based on Multiple Conditions

---

ADO's `Find` method does not allow you to find records based on more than one condition. The workaround is using the `Recordset` object's `Filter` property to create a view of the recordset that contains only those records that match the specified criteria. The procedure in Hands-On 12.38 uses the `Filter` property to find the female employees who live in the United States.



### Hands-On 12.38 Finding a Record Based on Multiple Criteria

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **Find\_WithFilter** procedure shown here:

```
Sub Find_WithFilter()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    conn.Open _
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set rst = New ADODB.Recordset
    rst.Open "Employees", conn, _
        adOpenKeyset, adLockOptimistic
    rst.Filter =
        "TitleOfCourtesy ='Ms.' and Country ='USA'"
    Do Until rst.EOF
        Debug.Print rst.Fields("LastName").Value
        rst.MoveNext
    Loop

    rst.Close
    Set rst = Nothing
    conn.Close
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### Using Bookmarks

When you work with database records, you must keep in mind that the actual number of records in a recordset can change at any time as new records are added or others are deleted. Therefore, you cannot save a record number to return to it later. Because records change all the time, the record numbers cannot be trusted. However, programmers often need to save the position of a record after they've moved to it or found it based on certain criteria. Instead of scrolling through every record in a recordset comparing the values, you can move directly to a specific record by using a bookmark. A *bookmark* is a value that uniquely identifies a row in a recordset.

Use the `Bookmark` property of the Recordset object to mark the record so you can return to it later. The `Bookmark` property is read/write, which means that you can get a bookmark for a record or set the current record in a Recordset object to the record identified by a valid bookmark. The Recordset's `Bookmark` property always represents the current row. Therefore, if you need to mark more than one row for later retrieval, you may want to use an array to store multiple bookmarks (see Hands-On 12.39).

A single bookmark can be stored in a Variant variable. For example, when you get to a particular row in a recordset and decide that you'd like to save its location, store the recordset's bookmark in a variable, like this:

```
varMyBkmrk = rst.Bookmark
```

`varMyBkmrk` is the name of a Variant variable declared with the following statement:

```
Dim varMyBkmrk As Variant
```

To retrieve the bookmark, move to another row, then use the saved bookmark to move back to the original row, like this:

```
rst.Bookmark = varMyBkmrk
```

Because not all ADO Recordsets support the `Bookmark` property, you should use the `Supports` method to determine if the recordset does. Here's how:

```
If rst.Supports(adBookmark) Then
    MsgBox "Bookmarks are supported."
Else
    MsgBox "Sorry, can't use bookmarks!"
End If
```

Recordsets defined with a Static or Keyset cursor always support bookmarks. If you remove the `adOpenKeyset` intrinsic constant from the code used in the next procedure (Hands-On 12.40), the default cursor (`adOpenForwardOnly`) will be used, and you'll get an error because this cursor does not support bookmarks.

Another precaution to keep in mind is that there is no valid bookmark when the current row is positioned at the new row in a recordset. For example, if you add a new record with the following statement:

```
rst.AddNew
```

and then attempt to mark this record with a bookmark:

```
varMyBkmrk = rst.Bookmark
```

you will get an error.

When you close the recordset, bookmarks you've saved become invalid. Also, bookmarks are unique to the recordset in which they were created. This means that you cannot use a bookmark created in one recordset to move to the same record in another recordset. However, if you clone a recordset (that is, you create a duplicate Recordset object), a `Bookmark` object from one Recordset object will refer to the same record in its clone.



### Hands-On 12.39 Marking Records with a Bookmark

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `TestBookmark` procedure:

```
Sub TestBookmark()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim varMyBkmrk As Variant

    Set conn = New ADODB.Connection
    conn.Open _
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set rst = New ADODB.Recordset
    rst.Open "Employees", conn, adOpenKeyset

    If Not rst.Supports(adBookmark) Then
        MsgBox "This recordset does not " & _
            "support bookmarks!"
        Exit Sub
    End If

    varMyBkmrk = rst.Bookmark
    Debug.Print rst.Fields(1).Value

    ' Move to the 7th row
    rst.AbsolutePosition = 7
    Debug.Print rst.Fields(1).Value

    ' move back to the first row
    ' using bookmark
    rst.Bookmark = varMyBkmrk
    Debug.Print rst.Fields(1).Value
    rst.Close
    Set rst = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

Notice that this procedure uses the `AbsolutePosition` property of the `Recordset` object. The absolute position isn't the same as the record number. This property can change if a record with a lower number is deleted.

### ***Using Bookmarks to Filter a Recordset***

---

Bookmarks provide the fastest way of moving through rows. You can also use them to filter a recordset as shown in Hands-On 12.40.



#### **Hands-On 12.40 Using Bookmarks to Filter Records**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **Filter\_WithBookmark** procedure:

```
Sub Filter_WithBookmark()
    Dim rst As ADODB.Recordset
    Dim varMyBkmrk() As Variant
    Dim strConn As String
    Dim i As Integer
    Dim strCountry As String
    Dim strCity As String

    i = 0
    strCountry = "France"
    strCity = "Paris"

    strConn = _
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set rst = New ADODB.Recordset
    rst.Open "Customers", strConn, adOpenKeyset

    If Not rst.Supports(adBookmark) Then
        MsgBox "This recordset does not " & _
            "support bookmarks!"
        Exit Sub
    End If

    Do While Not rst.EOF
        If rst.Fields("Country") = strCountry And _
            rst.Fields("City") = strCity Then
```

```

        ReDim Preserve varMyBkmrk(i)
        varMyBkmrk(i) = rst.Bookmark
        i = i + 1
    End If
    rst.MoveNext
Loop

rst.Filter = varMyBkmrk()

rst.MoveFirst
Do While Not rst.EOF
    Debug.Print rst("CustomerId") & _
        " - " & rst("CompanyName") -
    rst.MoveNext
Loop
rst.Close
Set rst = Nothing
End Sub

```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### **Using the GetRows Method to Fill the Recordset**

---

To retrieve multiple rows from a recordset, use the `GetRows` method, which returns a two-dimensional array. Recall that using arrays in VBA procedures was the main focus of Chapter 7. To find out how many rows were retrieved, use VBA's `UBound` function, as illustrated in Hands-On 12.41. Because arrays are zero-based by default, you must add one (1) to the result of the `UBound` function to get the correct record count.



#### **Hands-On 12.41 Counting the Number of Returned Records**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `CountRecords` procedure:

```

Sub CountRecords()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim myarray As Variant
    Dim returnedRows As Integer
    Dim r As Integer ' record counter
    Dim f As Integer ' field counter

    Set conn = New ADODB.Connection
    conn.Open _
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _

```

```
"Data Source=" & CurrentProject.Path & _
"\Northwind_Chap12.mdb"

Set rst = New ADODB.Recordset
rst.Open "SELECT * FROM Employees", _
conn, adOpenForwardOnly, _
adLockReadOnly, _
adCmdText

' Return all rows into array
myarray = rst.GetRows()
returnedRows = UBound(myarray, 2) + 1

MsgBox "Total number of records: " & _
returnedRows

' Find upper bound of second dimension
For r = 0 To UBound(myarray, 2)
    Debug.Print "Record " & r + 1
    ' Find upper bound of first dimension
    For f = 0 To UBound(myarray, 1)
        ' Print data from each row in array
        Debug.Print Tab;
        rst.Fields(f).Name & " = " & myarray(f, r)
    Next f
Next r

rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

Notice how the CountRecords procedure prints the contents of the array to the Immediate window by using a nested loop.

---

## WORKING WITH RECORDS IN ADO

Now that you've familiarized yourself with various methods of opening, moving around in, and finding records, and reading the contents of a recordset, let's look at ADO techniques for adding, modifying, copying, deleting, and sorting records.

## Adding a New Record

To add a new record, use the ADO Recordset's `AddNew` method. Use the `Update` method if you are not going to add any more records. In ADO, it is not necessary to call the `Update` method if you are moving to the next record. Calling the `Move` method implicitly calls the `Update` method before moving to the new record. Look at the following statements:

```
rst![Last Name] = "Roberts"  
rst.MoveNext
```

In this code fragment, the `Update` method is automatically called when you move to the next record. The procedure in Hands-On 12.42 demonstrates how to add a new record to the Employees table.



### Hands-On 12.42 Adding a New Record to a Table

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `AddNewRec_ADO` procedure:

```
' Use the References dialog box  
' to set up a reference to  
' the Microsoft ActiveX Data 6.1 Object Library  
  
Sub AddNewRec_ADO()  
    Dim conn As ADODB.Connection  
    Dim rst As ADODB.Recordset  
    Dim strConn As String  
  
    strConn = "Provider=Microsoft.ACE.OLEDB.12.0;" & _  
        "Data Source=" & CurrentProject.Path & _  
        "\Northwind 2007_Chap12.accdb"  
  
    Set rst = New ADODB.Recordset  
    With rst  
        .Open "SELECT * FROM Employees", _  
            strConn, adOpenKeyset, adLockOptimistic  
  
        ' Add a record and specify some field values  
        .AddNew  
        ! [Company] = "Northwind Traders"  
        ! [Last Name] = "Roberts"  
        ! [First Name] = "Paul"
```

```
! [Job Title] = "Sales Representative"
! [E-mail Address] = "paul@northwindtraders.com"

' Retrieve the Employee ID for the current record
Debug.Print !ID.Value

' Move to the first record
.MoveFirst
Debug.Print !ID.Value
.Close
End With

Set rst = Nothing
Set conn = Nothing
End Sub
```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

When adding or modifying records, you can set the record's field values in one of the following ways:

```
rst.Fields("First Name").value = "Paul"
```

or

```
rst![First Name] = "Paul"
```

As mentioned earlier, when you use the `AddNew` method to add a new record and then use the `Move` method, the newly added record is automatically saved without explicitly having to call the `Update` method. In the preceding example procedure, we used the `MoveFirst` method to move to the first record; however, you can call any of the other move methods (`Move`, `MoveNext`, `MovePrevious`) to have ADO implicitly call the `Update` method. After calling the `AddNew` method, the new record becomes the current record.

## Modifying a Record

---

To modify data in a specific field, find the record and set the `Value` property of the required field to a new value. Always call the `Update` method if you are not planning to edit any more records. If you modify a row and then try to close the recordset without calling the `Update` method first, ADO will trigger a runtime error.

The procedure in Hands-On 12.43 modifies an employee record.



### Hands-On 12.43 Modifying a Record

This hands-on exercise requires the completion of Hands-On 12.42.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **ModifyRecord\_ADO** procedure shown here:

```
Sub ModifyRecord_ADO()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strConn As String

    strConn = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
              "Data Source=" & CurrentProject.Path & _
              "\Northwind 2007_Chap12.accdb"

    Set rst = New ADODB.Recordset

    With rst
        .Open "SELECT * FROM Employees WHERE " &
              "[Last Name] = 'Roberts'", _
              strConn, adOpenKeyset, adLockOptimistic
        .Fields("City").Value = "Redmond"
        .Fields("State/Province").Value = "WA"
        .Fields("Country/Region").Value = "USA"
        .Update
        .Close
    End With

    Set rst = Nothing
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

This procedure modifies a table record by first accessing the desired fields. You can modify several fields in a specific record by calling the `Update` method and passing it two arrays. The first array should specify the field names, and the second one should list the new values to be entered. For example, the following statement updates the data in the City, State/Province, and Country/Region fields with the corresponding values:

```
rst.Update Array("City", "State/Province", "Country/Region"),
Array("Redmond", "WA", "USA")
```

You can use the same technique with the `AddNew` method.

## Editing Multiple Records

ADO has the ability to perform batch updates. This means that you can edit multiple records and send them to the OLE DB provider in a single operation. To take advantage of batch updates, you must use the Keyset or Static cursor (see the Introduction to ADO Recordsets earlier in this chapter for more information about cursors).

The procedure in Hands-On 12.44 finds all records in the Employees table where Title is “Sales Representative” and changes it to “Sales Rep.” The changes are then committed to the database in a single Update operation.



### Hands-On 12.44 Performing Batch Updates

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module’s Code window, type the following **BatchUpdate\_Records\_ADO** procedure:

```
Sub BatchUpdate_Records_ADO()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strConn As String
    Dim strCriteria As String

    strConn = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind 2007_Chap12.accdb"

    strCriteria = "[Job Title] = 'Sales Representative'"

    Set conn = New ADODB.Connection
    conn.Open strConn

    Set rst = New ADODB.Recordset

    With rst
        Set .ActiveConnection = conn
        .Source = "Employees"
        .CursorLocation = adUseClient
        .LockType = adLockBatchOptimistic
        .CursorType = adOpenKeyset
        .Open
        .Find strCriteria
        Do While Not .EOF
            .Fields("Job Title") = "Sales Rep"
        Loop
    End With
End Sub
```

```
.Find strCriteria, 1
Loop
.UpdateBatch
End With

rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

The BatchUpdate\_Records\_ADO procedure uses the ADO `Find` method to locate all the records that need to be modified. Once the first record is located, it is changed in memory and the find operation goes on to search for the next record and so on until the end of the recordset is reached. Notice that the following statement is issued to search past the current record:

```
.Find strCriteria, 1
```

Once all the records have been located and changed, the changes are all committed to the database in a single operation by issuing the `UpdateBatch` statement.

---

#### SIDE BAR *Updating Data: Differences between ADO and DAO*

ADO differs from DAO in the way update and delete operations are performed. In DAO, you are required to use the `Edit` method of the Recordset object prior to making any changes to your data. ADO does not require you to do this; consequently, there is no `Edit` method in ADO. Also, in ADO, your changes are automatically saved when you modify a record. In DAO, leaving a row without first calling the `Update` method of the Recordset object will automatically discard your changes.

---

### Deleting a Record

To delete a record, find the record you want to delete and call the `Delete` method. After you delete a record, it's still the current record. You must use the `MoveNext` method to move to the next row if you are planning to perform additional operations with your records. An attempt to do anything with the row that has just been deleted will generate a runtime error. The procedure in Hands-On 12.45 deletes a record from the Employees table.



### Hands-On 12.45 Deleting a Record

This hands-on exercise requires the completion of Hands-On 12.42.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **Delete\_Record\_ADO** procedure shown here:

```
Sub Delete_Record_ADO()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strConn As String

    ' call procedure from Hands-On 12.42 to ensure
    ' that we have a record to delete
    AddNewRec_ADO

    strConn = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
              "Data Source=" & CurrentProject.Path & _
              "\Northwind 2007_Chap12.accdb"

    Set conn = New ADODB.Connection
    Set rst = New ADODB.Recordset

    With rst
        .Open "SELECT * FROM Employees WHERE " &
              & "[Last Name] ='Roberts'", _
              strConn, adOpenKeyset, adLockOptimistic
        .Delete
        .Close
    End With
    Set rst = Nothing
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

Because we don't want to delete any original rows in the Employees table, the procedure makes a call to the AddNewRec\_ADO procedure that we created in Hands-On 12.42 to ensure that we have a custom row to delete.

### Copying Records to a Word Document

There are several techniques for placing Microsoft Access data in a Microsoft Word document. The procedure in Hands-On 12.46 demonstrates how to use

the Recordset's `GetString` method to insert data from the Invoice Data table into a newly created Word document.



### Hands-On 12.46 Copying Records to a Word Document

1. Choose **Tools | References** in the Visual Basic Editor window. Scroll down to locate the **Microsoft Word 16 Object Library**, click the checkbox next to it, and then click **OK** to exit.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following **SendToWord\_ADO** procedure:

```
' be sure to select Microsoft Word 16 Object Library
' in the References dialog box

Public myWord As Word.Application

Sub SendToWord_ADO()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim doc As Word.Document
    Dim strSQL As String
    Dim varRst As Variant
    Dim f As Variant
    Dim strHead As String

    Set conn = New ADODB.Connection
    Set rst = New ADODB.Recordset

    conn.Provider = "Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind 2007_Chap12.accdb"

    strSQL = "SELECT [Order ID] AS OrderID,"
    strSQL = strSQL & "[Ship Name], "
    strSQL = strSQL & "[Ship City] FROM [Invoice Data]"

    conn.Open
    rst.Open strSQL, conn, adOpenForwardOnly, _
        adLockReadOnly, adCmdText

    ' retrieve data and table headings
    ' into variables
    If Not rst.EOF Then
```

```
varRst = rst.GetString(adClipString, , _
vbTab, vbCrLf)
For Each f In rst.Fields
    strHead = strHead & f.Name & vbTab
Next
End If

' notice that Word application is declared
' at the top of the module
Set myWord = New Word.Application

' create a new Word document
Set doc = myWord.Documents.Add
myWord.Visible = True

' paste contents of variables into
' Word document
doc.Paragraphs(1).Range.Text = strHead & vbCrLf
doc.Paragraphs(2).Range.Text = varRst

On Error GoTo ErrorHandler
doc.Close SaveChanges:=wdPromptToSaveChanges
EndProc:
myWord.Quit
Set myWord = Nothing
Exit Sub
ErrorHandler:
If Err = 4198 Then
    MsgBox "You refused to save this document."
End If
Resume EndProc
End Sub
```

**4. Choose Run | Run Sub/UserForm** to execute the procedure.

This procedure uses the Recordset object's `GetString` method to return recordset data as a string-valued Variant (see “Returning a Recordset as a String” earlier in this chapter). Prior to running this procedure you must set a reference to the Microsoft Word 16 Object Library (or its lower version if you do not have Word 2016/2021 installed on the computer). This reference allows the procedure to access the Word application objects, properties, and methods via its own library. The top of the module contains the declaration of the `myWord` object variable that will point to the Word application. Notice that this variable is declared with the `Public` scope; therefore it can be accessed by other procedures in the current VBA project.

To launch Word and create a new document, we set the Application object to a new instance of `Word.Application` using the `New` keyword:

```
Set myWord = New Word.Application
```

To work with a Word document, the `Add` method of the Word Documents collection is used to create a blank document. We store the reference to this document in the `doc` object variable. To enable the user to see what's going on while the procedure is running, the `Visible` property of the Word application is set to `True`. Next, the contents of the Recordset and the field names that we previously saved in the string variables are written to the Word document using the Document object's `Paragraphs` property. The procedure ends by prompting the user to save changes to the Word document. If the user does not opt to save the document, error 4198 is triggered.

### **Copying Records to a Text File**

---

The procedure in Hands-On 12.47 demonstrates how to write the records from the Order Details table in the Northwind.mdb database to a text file named TestFile. Figure 12.10 shows the generated text file after it has been opened in Notepad.



#### **Hands-On 12.47 Copying Records to a Text File**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **WriteToFile** procedure:

```
Sub WriteToFile()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim f As ADODB.Field
    Dim fso As Object
    Dim txtfile As Object
    Dim strFileName As String

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
    "Data Source=" & CurrentProject.Path & _
    "\Northwind_Chap12.mdb"

    strFileName = CurrentProject.Path & "\TestFile.txt"
```

```
Set fso = CreateObject("Scripting.FileSystemObject")
Set txtfile = fso.CreateTextFile(strFileName, True)

Set rst = New ADODB.Recordset
rst.Open "[Order Details]", conn

With rst
    For Each f In .Fields
        ' Write field name to the text file
        txtfile.Write (f.Name)
        txtfile.Write Chr(9)
    Next

    ' move to a new line
    txtfile.WriteLine

    ' write out all the records to the text file
    txtfile.Write rst.GetString(adClipString)

    .Close
End With

txtfile.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

This procedure uses the `CreateObject` function to access the `FileSystemObject`. The `File` object is created using the `FileSystemObject`'s `CreateTextFile` method. The first argument of this method specifies the name of the file to create, and the second argument (`True`) indicates that the file should be overwritten if it already exists. Next, the procedure iterates through the recordset based on the Order Details table and writes field names to the text file using the `Write` method of the `File` object. The data from the recordset is converted into a string using the `GetString` method of the `Recordset` object and then written to the text file using the `File` object's `Write` method. The text file is then closed with the `Close` method.

OrderID	ProductID	UnitPrice	Quantity	Discount
10248	11	14	12	0
10248	42	9.8	10	0
10248	72	34.8	5	0
10249	14	18.6	9	0
10249	51	42.4	40	0
10250	41	7.7	10	0
10250	51	42.4	35	0.15
10250	65	16.8	15	0.15
10251	22	16.8	6	0.05
10251	57	15.6	15	0.05
10251	65	16.8	20	0
10252	20	64.8	40	0.05
10252	33	2	25	0.05
10252	60	27.2	40	0
10253	31	10	20	0

FIGURE 12.10 After running the WriteToFile procedure in Hands-On 12.47, the records from the Order Details table are placed in a text file.

## Filtering Records

The procedure in Hands-On 12.48 opens a recordset that contains only records having the value of Null in the Region field or an entry of “Mrs.” in the TitleOfCourtesy field.



### Hands-On 12.48 Filtering Records with the SQL WHERE Clause

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module’s Code window, type the following **FilterWithSQLWhere\_ADO** procedure:

```
Sub FilterWithSQLWhere_ADO()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strSQL As String

    strSQL = "SELECT * FROM Employees " & _
        "WHEREIsNull(Region)" & _
        " or TitleOfCourtesy = 'Mrs.' "

    Set conn = New ADODB.Connection
    conn.Open _
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"
```

```
Set rst = New ADODB.Recordset
rst.Open strSQL, conn, adOpenKeyset, _
    adLockOptimistic
MsgBox "Selected " & rst.RecordCount & _
    " records."

rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

Use the `Filter` property as a workaround to the ADO `Find` method whenever you need to find records that meet more than one condition. If the specific set of records you want to obtain is located on the SQL Server, you should use stored procedures instead of the `Filter` property.

The next procedure creates a filtered view of customers listed in the Northwind database who are located in Madrid, Spain.



### Hands-On 12.49 Filtering Records Using the Filter Property

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `FilterRecords_ADO` procedure:

```
Sub FilterRecords_ADO()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    conn.Open _
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set rst = New ADODB.Recordset
    With rst
        .Open "Customers", conn, _
            adOpenKeyset, adLockOptimistic
        .Filter = "City='Madrid' and Country='Spain'"
        MsgBox .RecordCount & _
            " records meet the criteria.", _
            vbInformation, "Customers in Madrid (Spain)"
    End With
```

```
Do Until rst.EOF
    Debug.Print rst.Fields(1).Value
    rst.MoveNext
Loop

rst.Filter = adFilterNone
MsgBox "Filter was removed. " & vbCrLf _
& "The table contains " & _
rst.RecordCount & " records."

rst.Close
Set rst = Nothing
conn.Close
Set conn = Nothing
End Sub
```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

This procedure defines the filter on the Customers table and displays the filtered records. Then the filter is removed by setting the Filter property to adFilterNone.

## Sorting Records

---

You can use the Recordset object's Sort property to change the order in which records are displayed. The Sort property does not physically rearrange the records; it merely displays the records in the order specified by the index. If you are sorting on non-indexed fields, a temporary index is created for each field specified in the index. This index is removed automatically when you set the Sort property to an empty string. In ADO you can only use Sort on client-side cursors. If you use the server-side cursor, you will receive this error: "The operation requested by the application is not supported by the provider."

The default sort order is ascending. To order a recordset by country in ascending order, then by city in descending order, you would use the following statement:

```
rst.Sort = "Country ASC, City DESC"
```

Although you can use the Sort property to sort your data, you will most likely get better performance by specifying an SQL ORDER BY clause in the SQL statement or query used to open the recordset. The procedure in Hands-On 12.50 displays customer records from the Northwind database in ascending order by country.

## Hands-On 12.50 Sorting Records

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **SortRecords\_ADO** procedure:

```
Sub SortRecords_ADO()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset

    Set conn = New ADODB.Connection
    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & CurrentProject.Path & _
        "\Northwind_Chap12.mdb"

    Set rst = New ADODB.Recordset

    ' sort on nonindexed field
    With rst
        .CursorLocation = adUseClient
        .Open "Customers", conn, adOpenKeyset, _
            adLockOptimistic
        .Sort = "Country"
        Do Until rst.EOF
            Debug.Print rst.Fields _
                ("CompanyName").Value & ":" & _
                rst.Fields("Country").Value
            .MoveNext
        Loop

        Debug.Print _
            "--original sort order --"; .Sort = ""
        Do Until .EOF
            Debug.Print rst.Fields _
                ("CompanyName").Value & ":" & _
                rst.Fields("Country").Value
            .MoveNext
        Loop
        .Close
    End With

    Set rst = Nothing
    conn.Close
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

In this procedure, after sorting records in the specified order, the `Sort` property is set to an empty string and records are displayed in the order in which they physically appear in the table.

## CREATING AND RUNNING QUERIES WITH ADO

---

Now that you're familiar with various methods to handling ADO recordsets, let's focus on creating and running Access queries from your VBA procedures.

### Creating a Select Query with ADO

In ADO, queries, SQL statements, views, and stored procedures are represented by the `Command` object. This object is part of the ADOX Object Model. The `Command` object has many properties and methods that will allow you to return records or execute changes to your data (inserts, updates, and deletes). In the following sections of this chapter, you will become acquainted with the properties of the `Command` object, including `ActiveConnection`, `CommandText`, and  `CommandType`. These properties will be discussed as they appear in the example procedure code. You will also learn how to use the `Command` object's `Execute` method to run your queries.

The procedure in Hands-On 12.51 demonstrates how to create and save a Select query using ActiveX Data Objects (ADO).



### **Hands-On 12.51 Creating a Select Query**

1. In the Visual Basic Editor window of the Chap15.accdb database, choose **Insert | Module**.
2. Ensure that the following object libraries are selected in the References dialog box:  
**ADO Ext. 6.0 for DDL and Security Object Library**  
**Microsoft ActiveX Data Objects 6.1 Object Library**
3. In the module's Code window, type the following `Create_SelectQuery_ADO` procedure:

```
Sub Create_SelectQuery_ADO()
    Dim cat As ADOX.Catalog
    Dim cmd As ADODB.Command
    Dim strPath As String
```

```
Dim strSQL As String
Dim strQryName As String

On Error GoTo ErrorHandler

' assign values to string variables
strPath = CurrentProject.Path & _
    "\Northwind 2007_Chap12.accdb"
strSQL = "SELECT Employees.* "
strSQL = strSQL & "FROM Employees WHERE "
strSQL = strSQL & "Employees.City='Redmond';"

strQryName = "Redmond Employees"

' open the Catalog
Set cat = New ADOX.Catalog
cat.ActiveConnection =
    "Provider=Microsoft.ACE.OLEDB.12.0;" & _
    "Data Source=" & strPath

' create a query based on the specified
' SELECT statement
Set cmd = New ADODB.Command
cmd.CommandText = strSQL

' add the new query to the database
cat.Views.Append strQryName, cmd

MsgBox "Completed successfully.", _
    vbInformation, "Create Select Query"
ExitHere:
    Set cmd = Nothing
    Set cat = Nothing
    Exit Sub

ErrorHandler:
    If InStr(Err.Description, _
        "already exists") Then
        cat.Views.Delete strQryName
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
        Resume ExitHere
    End If
End Sub
```

4. Choose **Run | Run Sub/UserForm** to execute the procedure.

The `Create_SelectQuery_ADO` procedure opens the `Catalog` object and sets its `ActiveConnection` property to the Northwind 2007.accdb database:

```
Set cat = New ADOX.Catalog  
cat.ActiveConnection="Provider=Microsoft.ACE.OLEDB.12.0;" & _  
    "Data Source=" & strPath
```

As you may recall from an earlier discussion, the `Catalog` object represents an entire database. It contains objects that represent all the elements of the database: tables, stored procedures, views, columns of tables, and indexes. The `ActiveConnection` property of the `Catalog` object indicates the ADO Connection object the Catalog belongs to. The value of this property can be a reference to the `Connection` object or a connection string containing the definition for a connection. Next, the procedure defines a `Command` object and uses its `CommandText` property to set the SQL statement for the query:

```
Set cmd = New ADODB.Command  
cmd.CommandText = strSQL
```

The `CommandText` property contains the text of a command you want to issue against a provider. In this procedure, we assigned the string variable's value (`strSQL`) to the `CommandText` property.

The ADO `Command` object always creates a temporary query. So, to create a stored (saved) query in a database, the procedure must append the `Command` object to the ADOX `Views` collection, like this:

```
cat.Views.Append strQryName, cmd
```

When you open the Northwind 2007\_Chap12.accdb database after running this procedure, you will find the Redmond Employees query in the Access window.

---

**SIDE BAR** *Row-returning, Non-parameterized Queries*

Queries that return records, such as Select queries, are known as row-returning, *non-parameterized* queries.

In ADO, use the `View` object to work with queries that return records and do not take parameters. All `View` objects are contained in the `Views` collection of the ADOX `Catalog` object. To save these queries in a database, append the ADO `Command` object to the ADOX `Views` collection as shown in Hands-On 12.51.

---

## Executing an Existing Select Query with ADO

There's more than one way of executing a row-returning query with ADO. This section demonstrates two procedures that run the Products by Category query located in the Northwind\_Chap12.mdb database.

The procedure in Hands-On 12.52 uses the Command and Recordset objects to perform this task.



### Hands-On 12.52 Executing a Select Query

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **Execute\_SelectQuery\_ADO** procedure shown here:

```
Sub Execute_SelectQuery_ADO()
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset
    Dim strPath As String

    strPath = CurrentProject.Path & _
              "\Northwind_Chap12.mdb"

    Set cmd = New ADODB.Command
    With cmd
        .ActiveConnection =
            "Provider=Microsoft.ACE.OLEDB.12.0;" & _
            "Data Source=" & strPath
        .CommandText = "[Products by Category]"
        . CommandType = adCmdTable
    End With

    Set rst = New ADODB.Recordset
    Set rst = cmd.Execute

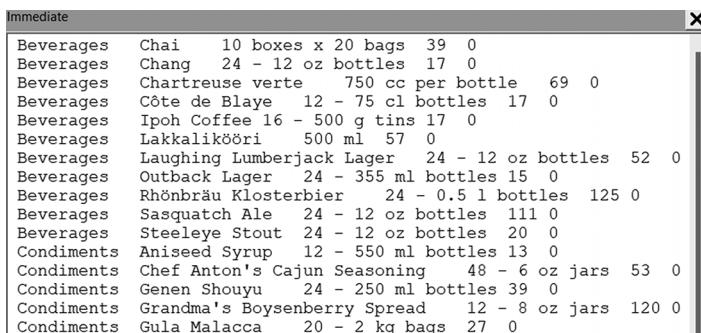
    Debug.Print rst.GetString

    rst.Close
    Set rst = Nothing
    Set cmd = Nothing
    MsgBox "View results in the Immediate window."
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

In the **Execute\_SelectQuery\_ADO** procedure, the connection to the database is opened by setting the **ActiveConnection** property of the **Command** object.

Next, the `Command` object's `CommandText` property specifies the name of the query you want to run. Notice that you need to place square brackets around the query's name when it contains spaces. The query type is determined by setting the  `CommandType` property of the `Command` object. Use the `adCmdTable` or `adCmdStoredProc` constants if the query string in the `CommandText` property is a query name. Finally, the `Execute` method of the `Command` object executes the query. Notice that the resulting recordset is passed to the `Recordset` object variable so that you can access the records retrieved by the query. Instead of looping through the records to read the returned records, the procedure uses the `Recordset` object's `GetString` method to print all the recordset rows to the Immediate window. The `GetString` method returns the recordset as a string. Figure 12.11 shows the output of the `Execute_Select Query_ADO` procedure.



The screenshot shows the Microsoft Visual Studio Immediate window. The title bar says "Immediate". The window displays a list of product names and their details, such as Chai, Chang, Chartreuse verte, Côte de Blaye, Ipoh Coffee, Lakkalikööri, Laughing Lumberjack Lager, Outback Lager, Rhönbräu Klosterbier, Sasquatch Ale, Steeleye Stout, Aniseed Syrup, Chef Anton's Cajun Seasoning, Genen Shouyu, Grandma's Boysenberry Spread, and Gula Malacca. Each entry includes the category (Beverages or Condiments), product name, description, unit, quantity, and price.

Beverages	Chai	10 boxes x 20 bags	39	0	
Beverages	Chang	24 - 12 oz bottles	17	0	
Beverages	Chartreuse verte	750 cc per bottle	69	0	
Beverages	Côte de Blaye	12 - 75 cl bottles	17	0	
Beverages	Ipoh Coffee	16 - 500 g tins	17	0	
Beverages	Lakkalikööri	500 ml	57	0	
Beverages	Laughing Lumberjack Lager	24 - 12 oz bottles	52	0	
Beverages	Outback Lager	24 - 355 ml bottles	15	0	
Beverages	Rhönbräu Klosterbier	24 - 0.5 l bottles	125	0	
Beverages	Sasquatch Ale	24 - 12 oz bottles	111	0	
Beverages	Steeleye Stout	24 - 12 oz bottles	20	0	
Condiments	Aniseed Syrup	12 - 550 ml bottles	13	0	
Condiments	Chef Anton's Cajun Seasoning	48 - 6 oz jars	53	0	
Condiments	Genen Shouyu	24 - 250 ml bottles	39	0	
Condiments	Grandma's Boysenberry Spread	12 - 8 oz jars	120	0	
Condiments	Gula Malacca	20 - 2 kg bags	27	0	

FIGURE 12.11 This is a sample result of records that were generated by executing the Select query in Hands-On 12.52.

The example procedure in Hands-On 12.53 demonstrates another method of running a row-returning query with ADO. Notice that in addition to the ADO `Command` and `Recordset` objects, this procedure uses the ADOX `Catalog` object. The connection to the database is established by setting the `ActiveConnection` property of the `Catalog` object and not the `Command` object, as was the case in Hands-On 12.52.

### Hands-On 12.53 Executing a Select Query with an ADO Catalog Object

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `Execute_SelectQuery2_ADO` procedure:

```
Sub Execute_SelectQuery2_ADO()
    Dim cat As ADOX.Catalog
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset
    Dim strPath As String

    strPath = CurrentProject.Path & _
              "\Northwind.mdb"

    Set cat = New ADOX.Catalog
    cat.ActiveConnection =
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & strPath

    Set cmd = New ADODB.Command
    Set cmd = cat.Views("Products by Category").Command

    Set rst = New ADODB.Recordset
    rst.Open cmd, , adOpenStatic, _
              adLockReadOnly, adCmdTable

    Debug.Print rst.GetString
    MsgBox "The query returned " & _
           rst.RecordCount & vbCrLf & _
           " records to the Immediate window."
    rst.Close
    Set rst = Nothing
    Set cmd = Nothing
    Set cat = Nothing
End Sub
```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

In this procedure, the following line of code is used to indicate the name of the query to be executed:

```
Set cmd = cat.Views("Products by Category").Command
```

This statement sets the `cmd` object variable to the desired query stored in the `Views` collection of the `ADOX Catalog` object. Next, the `Open` method of the `Recordset` object is used to open the recordset based on the specified query:

```
rst.Open cmd, , adOpenStatic, adLockReadOnly, adCmdTable
```

Notice that several optional arguments of the `Open` method are used to specify the data source: `cmd`, `ActiveConnection` (a comma appears in this spot because the existing connection is being used), `CursorType` (`adOpenStatic`),

LockType (adLockReadOnly), and Options (adCmdTable). Refer to an earlier section of this chapter for information about using these ADO constants. Next, the procedure dumps the contents of the records into the Immediate window (just as the procedure in Hands-On 12.52 did) by using the Recordset's GetString method. The MsgBox function contains a string that includes the information about the number of records retrieved. The RecordCount property of the Recordset object is used to get the record count. To get the correct record count, you must set the CursorType argument of the Recordset's Open method to adOpenStatic. If you set this argument to adOpenDynamic or adOpenForwardOnly, the RecordCount property will return -1.

## Modifying an Existing Query

---

If you'd like to modify an existing query, follow these steps:

1. Retrieve the query from the Views or Procedures collection of the Catalog object.
2. Set the CommandText property of the Command object to the new SQL statement.
3. Save the changes by setting the Procedure or View object's Command property to the modified Command object.

Earlier in this chapter you learned how to create a Select query named Redmond Employees by using ADO. The following hands-on exercise modifies this query so that employee records are ordered by last name.



### Hands-On 12.54 Modifying a Select Query

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **Modify\_Query\_ADO** procedure:

```
Sub Modify_Query_ADO()
    Dim cat As ADOX.Catalog
    Dim cmd As ADODB.Command
    Dim strPath As String
    Dim newStrSQL As String
    Dim oldStrSQL As String
    Dim strQryName As String

    strPath = CurrentProject.Path &
              "\Northwind 2007_Chap12.accdb"
```

```
newStrSQL = "SELECT Employees.* FROM " &
    "Employees WHERE Employees.City='Redmond'" & _
    " ORDER BY [Last Name];"

strQryName = "Redmond Employees"

Set cat = New ADOX.Catalog
cat.ActiveConnection =
    "Provider=Microsoft.ACE.OLEDB.12.0;" & _
    "Data Source=" & strPath

Set cmd = New ADODB.Command
Set cmd = cat.Views(strQryName).Command

    ' get the current SQL statement for this query
oldStrSQL = cmd.CommandText

MsgBox oldStrSQL, vbInformation, _
    "Current SQL Statement"

    ' now update the query's SQL statement
cmd.CommandText = newStrSQL
MsgBox newStrSQL, vbInformation, _
    "New SQL Statement"

    ' save the modified query
Set cat.Views(strQryName).Command = cmd

Set cmd = Nothing
Set cat = Nothing
End Sub
```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

When you run this procedure the Redmond Employees query created in Hands-On 12.51 is modified from the following SQL statement:

```
SELECT Employees.*
FROM Employees
WHERE Employees.City='Redmond';
```

to:

```
SELECT Employees.*
FROM Employees
WHERE Employees.City='Redmond' ORDER BY [Last Name];
```

## Creating and Running a Parameter Query

In ADO, to create a row-returning, parameterized query, simply add the parameters to the query's SQL string. The parameters must be defined by using the PARAMETERS keyword, as in the following:

```
strSQL = "PARAMETERS [Country Name] Text;" & _  
    "SELECT Customers.* FROM Customers WHERE " _  
    & "Customers.Country=[Type Country Name];"
```

The preceding SQL statement begins by defining one parameter called `Country Name`. This parameter will be able to accept text entries. The second part of the SQL statement selects all the records from the `Customers` table that have an entry in the `Country` field equal to the provided parameter value. The complete procedure is shown in Hands-On 12.55.



### Hands-On 12.55 Creating a Parameter Query

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following `Create_ParameterQuery_ADO` procedure:

```
Sub Create_ParameterQuery_ADO()  
    Dim cat As ADOX.Catalog  
    Dim cmd As ADODB.Command  
    Dim strPath As String  
    Dim strSQL As String  
    Dim strQryName As String  
  
    On Error GoTo ErrorHandler  
  
    strPath = CurrentProject.Path & "\Northwind_Chap12.mdb"  
  
    strSQL = "PARAMETERS [Country Name] Text;" & _  
        "SELECT Customers.* FROM Customers WHERE " _  
        & "Customers.Country=[Country Name];"  
  
    strQryName = "Customers by Country"  
  
    Set cat = New ADOX.Catalog  
    cat.ActiveConnection = _  
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _  
        "Data Source=" & strPath
```

```
Set cmd = New ADODB.Command
cmd.CommandText = strSQL

cat.Procedures.Append strQryName, cmd
Set cmd = Nothing
Set cat = Nothing

MsgBox "The procedure completed.", _
vbInformation, "Create Parameter Query"
Exit Sub

ErrorHandler:
If InStr(Err.Description, "already exists") Then
    cat.Procedures.Delete strQryName
    Resume
Else
    MsgBox Err.Number & ":" & Err.Description
End If
End Sub
```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

This procedure creates a simple Parameter query with one parameter. Because the ADO Command object always creates a temporary query, you must append the Command object to the ADOX Procedures collection in order to save a parameterized query in a database.

---

**SIDE BAR** *Row-Returning, Parameterized Queries*

Queries that return records and take parameters are known as row-returning, parameterized queries.

In ADO, use the ADOX Procedure object to work with queries that return records and take parameters. All Procedure objects are contained in the Procedures collection of the ADOX Catalog object. To save these queries in a database, append the ADO Command object to the ADOX Procedures collection.

---

To execute a Parameter query you must specify the parameter value using the Parameters collection of the Command object, like this:

```
cmd.Parameters("Country Name") = "France"
```

The procedure in Hands-On 12.56 shows how to run the Parameter query created by the procedure in Hands-On 12.55.



### Hands-On 12.56 Executing a Parameter Query

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **Execute\_ParamQuery\_ADO** procedure:

```
Sub Execute_ParamQuery_ADO(strCountry As String)
    Dim cat As ADOX.Catalog
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset
    Dim strQryName As String
    Dim strPath As String

    strQryName = "Customers by Country"
    strPath = CurrentProject.Path & "\Northwind_Chap12.mdb"

    Set cat = New ADOX.Catalog
    cat.ActiveConnection =
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & strPath

    Set cmd = New ADODB.Command
    Set cmd = cat.Procedures(strQryName).Command

    ' specify a parameter value
    cmd.Parameters("[Country Name]") = strCountry

    ' use the Execute method of the Command
    ' object to open the recordset
    Set rst = cmd.Execute

    ' return company names to the Immediate window
    Do Until rst.EOF
        Debug.Print rst(1)
        rst.MoveNext
    Loop

    rst.Close
    Set rst = Nothing
    Set cmd = Nothing
    Set cat = Nothing
End Sub
```

3. Execute this procedure from the Immediate window by typing the following statement and pressing **Enter**:

```
Execute_ParamQuery_ADO "Argentina"
```

The Execute\_ParamQuery\_ADO procedure establishes the connection to the Northwind database. Next, the name of the query is supplied in the following statement:

```
Set cmd = cat.Procedures(strQryName).Command
```

Because this is a Parameter query, the parameter value is specified by using the Parameters collection of the Command object, like this:

```
cmd.Parameters("[Country Name]") = strCountry
```

Then, the Recordset object is opened by using the Execute method of the Command object:

```
Set rst = cmd.Execute
```

Finally, the procedure loops through the recordset to retrieve the company names and print them to the Immediate window. After running this procedure, the following lines are returned to the Immediate window for the specified country:

```
Cactus Comidas para llevar  
Océano Atlántico Ltda.  
Rancho grande
```

**NOTE**

*Instead of specifying the parameter values before the recordset is open, you can use the Parameters argument of the Command object's Execute method to pass the parameter value, as follows:*

```
Set rst = cmd.Execute(Parameters:=strCountry)
```

## Executing an Update Query

Executing bulk queries that update data is quite easy with ADO. You can use the Execute method of the Connection or Command object. The procedure in Hands-On 12.57 uses the Connection object's Execute method to update records in the Products table of the Northwind.mdb database where CategoryId is equal to 8. The UnitPrice of the records that match this condition will be increased by one dollar. Note that the number of updated records is returned by the Execute method in the NumOfRec variable.



### Hands-On 12.57 Executing an Update Query

1. In the Visual Basic Editor window, choose **Insert | Module**.

2. In the module's Code window, type the following **Execute\_UpdateQuery\_ADO** procedure:

```
Sub Execute_UpdateQuery_ADO()
    Dim conn As ADODB.Connection
    Dim NumOfRec As Integer
    Dim strPath As String

    strPath = CurrentProject.Path & "\Northwind_Chap12.mdb"

    Set conn = New ADODB.Connection

    conn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
              "Data Source=" & strPath

    conn.Execute "UPDATE Products " & _
                "SET UnitPrice = UnitPrice + 1" & _
                " WHERE CategoryId = 8", _
                NumOfRec, adExecuteNoRecords

    MsgBox NumOfRec & " records were updated."
    conn.Close
    Set conn = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

This procedure uses the Data Manipulation Language (DML) UPDATE statement to make a change in the UnitPrice field of the Products table. The `Execute` method of the Connection object allows the provider to return the number of records that were affected via the `RecordsAffected` parameter. This parameter applies only to Action queries or stored procedures. To get the number of records returned by a result-returning query or stored procedure, you must use the `RecordCount` property. In the `Execute_UpdateQuery_ADO` procedure, we store the number of records affected in the string variable `NumOfRec`. Note that when a command does not return a recordset, you should include the constant `adExecuteNoRecords`. The `adExecuteNoRecords` constant can only be passed as an optional parameter to the `Command` or `Connection` object's `Execute` method.

The procedure in Hands-On 12.58 demonstrates how to execute an Update query by using the ADO Command object instead of the Connection object used in the preceding example. After running the following example, the Unit-Price of all the records in the Products table will increase by 10 percent.



### Hands-On 12.58 Executing an Update Query Using the Command Object

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the `Execute_UpdateQuery2_ADO` procedure shown here:

```
Sub Execute_UpdateQuery2_ADO()
    Dim cmd As ADODB.Command
    Dim NumOfRec As Integer
    Dim strPath As String

    strPath = CurrentProject.Path & "\Northwind_Chap12.mdb"

    Set cmd = New ADODB.Command
    With cmd
        .ActiveConnection =
            "Provider=Microsoft.ACE.OLEDB.12.0;" & _
            "Data Source=" & strPath
        .CommandText = "Update Products " & _
            "Set UnitPrice = UnitPrice *1.1"
        .Execute NumOfRec, adExecuteNoRecords
    End With
    MsgBox NumOfRec
    Set cmd = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

#### SIDE BAR *Non-Row-Returning Queries*

Queries that do not return records, such as Action queries or Data Definition Language (DDL) queries, are known as non-row-returning queries.

- Action queries are Data Manipulation Language (DML) queries that perform bulk operations on a set of records. They allow you to add, update, or delete records.
- DDL queries are used for creating database objects and altering the structure of a database.
- Use the ADOX Procedure object to work with queries that don't return records. All Procedure objects are contained in the Procedures collection of the ADOX Catalog object. To save these types of queries in a database, append the ADO Command object to the ADOX Procedures collection.

## Creating and Executing a Pass-Through Query

As mentioned earlier, SQL Pass-Through queries are SQL statements that are sent directly to the database server for processing. In earlier versions of Access, Pass-Through queries were used with Data Access Objects (DAO) to increase performance when accessing external ODBC data sources. In ADO, you can use the Microsoft OLE DB Provider for SQL Server to directly access the SQL Server. For this reason, you do not need to create Pass-Through queries. However, since it is possible to create a Pass-Through query using ADOX and Microsoft Jet Provider, the next hands-on exercise demonstrates how to do this.



### Hands-On 12.59 Creating a Pass-Through Query

This hands-on exercise requires that you have access to an SQL Server Northwind database and that you make changes in the connection string to point to your server.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the **Create\_PassThroughQuery** procedure shown here:

```
Sub Create_PassThroughQuery()
    Dim cat As ADOX.Catalog
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset
    Dim strPath As String
    Dim strSQL As String
    Dim strQryName As String
    Dim strDBCCConnect As String

    On Error GoTo ErrorHandler

    strSQL = "SELECT Customers.* FROM " & _
        "Customers WHERE " & _
        "Customers.Country='France';"

    strQryName = "French Customers"

    ' modify the following string to connect
    ' to your SQL Server
    strDBCCConnect = "ODBC;Driver=SQL Server;" & _
        "Server=PROD15;" & _
        "Database=Northwind;" & _
        "UID=" & _
        "PWD="
```

```

'strODBCConnect = "ODBC;DSN=ODBCNorth;UID=sa;PWD=;"'

Set cat = New ADOX.Catalog
cat.ActiveConnection = CurrentProject.Connection

Set cmd = New ADODB.Command
With cmd
    .ActiveConnection = cat.ActiveConnection
    .CommandText = strSQL
    .Properties _ 
        ("Jet OLEDB:ODBC Pass-Through Statement") = True
    .Properties _ 
        ("Jet OLEDB:Pass-Through Query Connect String") _ 
        = strODBCConnect
End With

cat.Procedures.Append strQryName, cmd

Set cmd = Nothing
Set cat = Nothing
MsgBox "The procedure completed successfully.", _
    vbInformation, "Create Pass-Through Query"
Exit Sub

ErrorHandler:
    If InStr(Err.Description, "already exists") Then
        cat.Procedures.Delete strQryName
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
    End If
End Sub

```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

This procedure creates a Pass-Through query named French Customers in the current database. Notice that to connect to the SQL Server database, the following string is built and later assigned to the Jet OLEDB:Pass-Through Query Connect String property of the Command object:

```

strODBCConnect = "ODBC;Driver=SQL Server;" & _
    "Server=PROD15;" & _
    "Database=Northwind;" & _
    "UID=;" & _
    "PWD="

```

To try this procedure, you must have access to a remote data source (such as an SQL Server database) and you'll need to modify the preceding string to point to your server. This string allows you to connect via the DSN-less connection. If you prefer, you may build your connection string to the remote data source using the DSN that you define in the Control Panel via Administrative Tools (ODBC). Your connection string could then look like this:

```
strODBCConnect = "ODBC;DSN=myDSN;UID=sa;PWD=;"
```

To create a Pass-Through query, you must also set two provider-specific properties of the Command object: Jet OLEDB:ODBC Pass-Through Statement and Jet OLEDB:Pass-Through Query Connect String.

To permanently store the Pass-Through query in your database, you need to append it to the Catalog object's Procedures collection, like this:

```
cat.Procedures.Append strQryName, cmd
```

After you run the `Create_PassThroughQuery` procedure, the query can be viewed and accessed from the navigation pane in the Access window.

In Hands-On 12.59, you learned how to create a Pass-Through query in VBA with ADO. This query retrieved the list of French customers from the Northwind database located on the SQL Server. The Pass-Through query was named French Customers and was saved permanently in the Chap12.accdb database. Let's find out how you can execute this query from a VBA procedure.



### Hands-On 12.60 Executing a Pass-Through Query Saved in Access

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the `Execute_PassThroughQuery_ADO` procedure shown here:

```
Sub Execute_PassThroughQuery_ADO()
    Dim cat As ADOX.Catalog
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset
    Dim strConnect As String

    ' modify the connection string to connect
    ' to your SQL Server Northwind database
    strConnect = "Provider=SQLOLEDB;" & _
        "Data Source=PROD15;" & _
        "Initial Catalog=Northwind;" & _
```

```
"User Id=sa;" & _  
"Password="  
  
Set cat = New ADOX.Catalog  
cat.ActiveConnection = CurrentProject.Connection  
  
Set cmd = New ADODB.Command  
Set cmd = cat.Procedures("French Customers").Command  
Set rst = cmd.Execute  
  
Debug.Print "--French Customers Only--" & vbCrLf _  
& rst.GetString  
  
Set rst = Nothing  
Set cmd = Nothing  
Set cat = Nothing  
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

The procedure begins by building a connection string to the SQL Server database. This is a standard connection that uses the native OLE DB SQL Server Provider (SQLOLEDB). This connection requires that you also provide the name of the SQL Server (Data Source), the name of the database from which to retrieve records (Initial Catalog), and the security context with which to log in (User Id, Password). If you connect to your SQL Server database using the NT integrated security, your connection string will look like this:

```
strConnect = "Provider=SQLOLEDB;" & _  
"Data Source=yourServerName;" & _  
"Integrated Security=SSPI;" & _  
"Initial Catalog=Northwind"
```

Because the Pass-Through query you want to execute has been saved in the Access database, you need to open the ADOX Catalog object to access its Procedures collection. The following line of code specifies the name of the query you want to execute and assigns it to the Command object:

```
Set cmd = cat.Procedures("French Customers").Command
```

To execute a Pass-Through query that returns records, you need to use the Recordset object in addition to the Command object. The following statement executes the Pass-Through query:

```
Set rst = cmd.Execute
```

The Pass-Through query executes on the server. To quickly view data on the client machine, we retrieve the contents of the recordset by using the `GetString` method:

```
Debug.Print "--French Customers Only--" & vbCrLf _  
& rst.GetString
```

### **Listing Queries in a Database**

---

The procedure in Hands-On 12.61 retrieves the names of all saved queries in the `Northwind_Chap12.mdb` database by iterating through the `View` objects stored in the ADOX Catalog object's `Views` collection.



#### **Hands-On 12.61 Listing Queries in a Database**

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the `List_AllQueries_ADO` procedure shown here:

```
Sub List_AllQueries_ADO()  
    Dim cat As New ADOX.Catalog  
    Dim v As ADOX.View  
    Dim strPath As String  
  
    strPath = CurrentProject.Path & "\Northwind_Chap12.mdb"  
    cat.ActiveConnection =  
        "Provider=Microsoft.ACE.OLEDB.12.0;" &  
        "Data Source= " & strPath  
  
    For Each v In cat.Views  
        Debug.Print v.Name  
    Next  
  
    Set cat = Nothing  
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.
4. After running this procedure, open the Immediate window to view the list of all saved queries in the `Northwind_Chap12.mdb` database.

### **Deleting a Query**

---

To delete a query in ADO, use the `Delete` method of the Procedures or Views collection. By running the procedure in Hands-On 12.62, you can quickly delete the Redmond Employees query created in Hands-On 12.51.



### Hands-On 12.62 Deleting a Query from a Database

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **DeleteAQuery\_ADO** procedure:

```
Sub DeleteAQuery_ADO()
    Dim cat As New ADOX.Catalog
    Dim strPath As String

    On Error GoTo ErrorHandler

    strPath = _
        CurrentProject.Path & "\Northwind 2007_Chap12.accdb"
    cat.ActiveConnection =
        "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source= " & strPath

    cat.Views.Delete "Redmond Employees"

    ExitHere:
    Set cat = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = 3265 Then
        MsgBox "Query does not exist."
    Else
        MsgBox Err.Number & ":" & Err.Description
    End If
    Resume ExitHere
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

After running the procedure in Hands-On 12.62, the query named Redmond Employees is removed from the Northwind 2007\_Chap12.accdb database.

In the preceding sections of this chapter, you learned how to use ADO objects to perform the most frequent database operations: create, run, and modify various types of queries. In the remaining sections of this chapter, we focus on using more advanced features of the ADO Object Model.

## USING ADVANCED ADO FEATURES

---

At this point you should feel comfortable using ADO in most of your Access programming endeavors. By using the knowledge you've acquired in this chapter, you can basically switch to any other Microsoft 365 application (Excel, Word, PowerPoint, or Outlook) and start programming. Because you already know the ADO methods of accessing databases and manipulating records, all you need to learn is the object model that the specific application is using. Learning a new type library is not very difficult. Recall that VBA offers the Object Browser that lists all the application's objects, properties, methods, and intrinsic constants that you may need for writing code. However, if you'd like to accomplish more with ADO, the following sections of this chapter will introduce you to a couple of more advanced ADO features that will set you apart from beginning programmers. You will learn about fabricating, persisting, disconnecting, cloning, and shaping recordsets. You will also learn how to process data modifications and additions by using ADO transactions.

### Fabricating a Recordset

---

So far you've worked with recordsets that were created from data that came from an Access database, a text or a dBASE file, an Excel spreadsheet, or a Word document. You may have also practiced working with a recordset generated from an SQL Server database. In each of these circumstances, to get the necessary data you needed to establish a connection to the appropriate data source. In other words, you worked with recordsets that had a live connection to the data source. These connected recordsets obtained their structure and data from a query to a data source to which they were connected. But what if you need to create a recordset with data that does not come from a data source? As you may recall from Chapter 10, "Data Access Technologies in Microsoft Access," the ADO Object Model allows you to work with both relational and non-relational data stores.

To store non-relational data in an ADO Recordset, you can create your recordset from scratch. This recordset will be defined programmatically in memory and will not be connected to any data source. For example, you can easily fabricate a custom recordset that holds non-relational data, such as the information about the files located in one of your hard drive's directories.

When you create your own recordset from scratch, you define the types of fields in the recordset and then populate the recordset with the information you want. The fields are defined using the Fields collection's `Append` method.

You must specify the field name and the data type. The syntax for the `Append` method looks like this:

```
Fields.Append Name, DataType[, FieldSize], [Attribute]
```

Arguments in square brackets are optional. `FieldSize` specifies the size in characters or bytes. `Attribute` specifies characteristics such as whether the field enables Null values or whether it is a primary key or an identity column.

Once you have defined the structure of your recordset, simply open it and populate it with the desired data. You can add data to your custom recordset in the same way you add data to a connected recordset: by using the `Recordset` object's `AddNew` method.

The procedures in Hands-On 12.63 demonstrate how to create an empty recordset containing three fields (Name, Size, and Modified), populate it with files located in a user-specified file folder, and then create a text file showing the directory listing.



### Hands-On 12.63 Creating a Custom Recordset

1. In the database window, press **Alt+F11** to switch to the Visual Basic Editor window.
2. In the Visual Basic Editor window, choose **Insert | Module**.
3. In the module's Code window, type the following **Custom\_Recordset** procedure:

```
Sub Custom_Recordset()
    Dim rst As ADODB.Recordset
    Dim strFile As String
    Dim strPath As String
    Dim strFolder As String
    Dim heading As Variant
    Dim listing As String

    Const MyFolder = "C:\VBAAccess2021_ByExample"

    heading = Array("Name", "Size", "Modified")

    strPath = InputBox("Enter pathname, e.g., " & MyFolder, _
        "Enter the Folder Name", MyFolder)

    If Right(strPath, 1) <> "\" Then strPath = strPath & "\"
    strFolder = strPath
```

```
strFile = Dir(strPath & "*.*")

If strFile = "" Then
    MsgBox "This folder does not contain files."
    Exit Sub
End If

Set rst = New ADODB.Recordset
' Create an empty recordset with 3 fields
With rst
    Set .ActiveConnection = Nothing
    .CursorLocation = adUseClient
    With .Fields
        .Append "Name", adVarChar, 255
        .Append "Size", adDouble
        .Append "Modified", adDBTimeStamp
    End With
    .Open
    Do While strFile <> ""
        If strFile = "" Then Exit Do
        ' Add a new record to the recordset
        .AddNew heading,
        Array(strFile, FileLen(strFolder & strFile), _
        FileDateTime(strFolder & strFile))
        ' get the name of the next file in folder
        strFile = Dir
    Loop
    .MoveFirst
    listing = rst.GetString(adClipString)
    .Close
End With

Set rst = Nothing
'create a text file with directory listing
CreateTxtFile heading, listing

End Sub

Sub CreateTxtFile(heading As Variant, rst As String)
    Dim fso As Object
    Dim txtfile As Object
    Dim strFileName As String

    strFileName = CurrentProject.Path & "\DirectoryListing.txt"
```

```
Set fso = CreateObject("Scripting.FileSystemObject")
Set txtfile = fso.CreateTextFile(strFileName, True)

With txtfile
    ' Write field names to the text file
    .Write heading(0)
    .Write Chr(9)    ' Tab
    .Write heading(1)
    .Write Chr(9)
    .Write heading(2)

    ' move to a new line
    .WriteLine

    ' write out all the records to the text file
    .Write rst

    ' close the text file
    .Close
End With

End Sub
```

**4. Choose Run | Run Sub/UserForm** to execute the procedure.

In the Custom\_Recordset procedure, we start by creating a Recordset object variable. To tell ADO that your recordset is not connected to any database, we set the ActiveConnection property of the Recordset object to Nothing. We also set the CursorLocation property to adUseClient to indicate that the processing will occur on the client machine as opposed to the database server. Next, we determine what columns the recordset should contain and add these columns to the Recordset's Fields collection by using the Append method. Once the structure of your recordset is defined, you can call the Open method to actually open your custom recordset. Now you can populate the recordset with the data you want. We obtain the data by looping through the folder the user specified in the input box and reading the information about each file. The VBA Dir function is used to obtain the filename in the specified path. The FileLen function is used to retrieve the size of a file in bytes. Another VBA function, FileDateTime, is used to retrieve the date and time a file was last modified. To retrieve the date and time separately, use the FileDateTime function as an argument of the DateValue or TimeValue functions.

Check the following statements in the Immediate window while stepping through the Custom\_Recordset procedure:

```
? DateValue(FileDateTime(myFolder & myFile))
? TimeValue(FileDateTime(myFolder & myFile))
```

For each found file we use the Recordset `AddNew` method to create a new record. The syntax of the `AddNew` method is shown below:

```
Recordset.AddNew FieldList, Values
```

The `FieldList` can be a single name, or an array of names. In our procedure we use the heading array to specify the field names. Because the `FieldList` is specified as an array, the `Values` must also be specified as an array with the same number of members. The order of fields must match the order of filed values in each array. Here's how we do it:

```
.AddNew heading, _
    Array(strFile, FileLen(strFolder & strFile), _
        FileDateTime(strFolder & strFile))
```

The `strFile` will provide the value for the `Name` column, the `FileLen` function specifies the value for the `Size` column, and the `FileDateTime` function will populate the `Modified Date` column.

Once the recordset is fabricated and populated with the required data, you can save it to a string variable using the `GetString` function and push the data to another application. In this case, we call the `CreateTxtFile` procedure that creates a text file with directory listing based on the passed parameters.

## Disconnected Recordsets

---

In the previous section, you learned how to create a recordset from scratch. This recordset had a structure custom-defined by you and was populated with data that did not come from a database. In other words, it was a disconnected recordset that was defined on the fly. A *disconnected recordset* is a recordset that is not connected to a data source. A disconnected recordset can be defined programmatically (as you saw in Hands-On 12.63) or it can get its information from the data source (as shown in Hands-On 12.64).

Using disconnected recordsets allows you to connect to a database, retrieve some records, return the records to the client, and then disconnect from the database. By keeping your connection to a database open just long enough to obtain the required data, you can help conserve valuable server resources. You can work with the disconnected recordset offline and then connect to the database again to add your changes.

To get started using disconnected recordsets, perform Hands-On 12.64. The example procedure retrieves some data from the `Orders` table in the Northwind

database, then disconnects from the database. While disconnected from the database, you can manipulate and examine the content of the retrieved recordset.



### Hands-On 12.64 Creating a Disconnected Recordset

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, type the following **Rst\_Disconnected** procedure:

```
Sub Rst_Disconnected()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strConn As String
    Dim strSQL As String
    Dim strRst As String
    Dim strFilePath As String
    Dim strFile As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_ByExample\
    strFile = "Northwind_Chap12.mdb"
    strSQL = "SELECT * FROM Orders WHERE " & _
              "CustomerID = 'VINET'"
    strFilePath = strPath & strFile

    strConn = "Provider=Microsoft.ACE.OLEDB.12.0;" 
    strConn = strConn & "Data Source = " & strFilePath
    Set conn = New ADODB.Connection
    conn.ConnectionString = strConn
    conn.Open

    Set rst = New ADODB.Recordset
    Set rst.ActiveConnection = conn

    ' retrieve the data
    rst.CursorLocation = adUseClient
    rst.LockType = adLockBatchOptimistic
    rst.CursorType = adOpenStatic
    rst.Open strSQL, , , adCmdText

    ' disconnect the recordset
    Set rst.ActiveConnection = Nothing

    ' change the CustomerID in the first
    ' record to 'OCEAN'
    rst.MoveFirst
```

```
Debug.Print rst.Fields(0) & " was previously: " _  
    & rst.Fields(1)  
rst.Fields("CustomerID").Value = "OCEAN"  
rst.Update  
  
' stream out the recordset as  
' a comma-delimited string  
strRst = rst.GetString(adClipString, , ",")  
Debug.Print strRst  
End Sub
```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

Notice that to create a disconnected recordset that gets its data from a data source, you need to set the `CursorLocation`, `LockType`, and `CursorType` properties of the `Recordset` object. `CursorLocation` should be set to `adUseClient`. This setting indicates that the cursor will reside on the client computer that is creating the recordset. Set `LockType` to `adLockBatchOptimistic` to enable multiple records to be updated. Finally, set `CursorType` to `adOpenStatic` to retrieve the snapshot of the data.

To disconnect a recordset, you must set the `Recordset` object's `ActiveConnection` property to `Nothing` after you've called the `Recordset`'s `Open` method.

When the recordset is disconnected from the database, you can freely manipulate its data or pass it to another application or process. In this example procedure, we manipulate our recordset by changing the value of the `CustomerID` field in the first retrieved record from `VINET` to `OCEAN`. Then we create a comma-delimited string using the `Recordset` object's `GetString` method. The content of the disconnected recordset is then printed out to the Immediate window, as shown here:

```
10274 was previously: VINET  
10274,OCEAN,6,8/6/1996,9/3/1996,8/16/1996,1,6.01,Vins et alcools  
Chevalier,59 rue de l'Abbaye,Reims,,51100,France  
10295,VINET,2,9/2/1996,9/30/1996,9/10/1996,2,1.15,Vins et alcools  
Chevalier,59 rue de l'Abbaye,Reims,,51100,France  
10737,VINET,2,11/11/1997,12/9/1997,11/18/1997,2,7.79,Vins et  
alcools Chevalier,59 rue de l'Abbaye,Reims,,51100,France  
10739,VINET,3,11/12/1997,12/10/1997,11/17/1997,3,11.08,Vins et  
alcools Chevalier,59 rue de l'Abbaye,Reims,,51100,France
```

## Saving a Recordset to Disk

The ADO has a `Save` method that allows you to save a recordset to disk and work with it from your VBA application. This method takes two parameters. You must specify a filename and one of the following two data formats:

- `adPersistADTG`—Advanced Data TableGram
- `adPersistXML`—Extensible Markup Language

A *saved (or persisted) recordset* is a recordset that is saved to a file. This file can later be reopened without an active connection.

In this section, you will persist a recordset into a file using the `adPersistADTG` format. You will work with the `adPersistXML` format in Chapter 26, “XML Features in Access 2021.”

To save a recordset in a file, you must first open it. When you have applied a filter to a recordset and then decide to save that recordset, only the filtered records will be saved. Using the `Save` method does not close the recordset. You can continue to work with the recordset after it has been saved. However, always remember to close the recordset when you are done working with it.

The procedure in Hands-On 12.65 opens the recordset based on the Customers table. Once the recordset is open, the `Save` method is called to persist the customer records into a file.



### Hands-On 12.65 Saving Records to a Disk File

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module’s Code window, type the following **SaveRecordsToDisk** procedure:

```
Sub SaveRecordsToDisk()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strFileName As String
    Dim strNorthPath As String

    strFileName = CurrentProject.Path & "\Companies.rst"
    strNorthPath = CurrentProject.Path & "\Northwind_Chap12.mdb"

    On Error GoTo ErrorHandler

    Set conn = New ADODB.Connection

    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
```

```
.ConnectionString = "Data Source = " & strNorthPath
.Mode = adModeReadWrite
.Open
End With

Set rst = New ADODB.Recordset
With rst
    .CursorLocation = adUseClient
    ' Retrieve the data
    .Open "Customers", conn,
        _adOpenKeyset, adLockBatchOptimistic, adCmdTable

    ' Disconnect the recordset
    .ActiveConnection = Nothing

    ' Save the recordset to disk
    .Save strFileName, adPersistADTG
    .Close
End With

MsgBox "Records were saved in " & strFileName & "."

ExitHere:
    ' Cleanup
    Set rst = Nothing
Exit Sub

ErrorHandler:
    If Not IsEmpty(Dir(strFileName)) Then
        Kill strFileName
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
        Resume ExitHere
    End If
End Sub
```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

This procedure saves all the data located in the Customers table to a file with an .rst extension. We named this file Companies.rst, but you are free to choose any filename and extension while saving your recordset.

Persisted recordsets are very useful for populating combo boxes or listboxes, especially when the data is located on a server and does not change too often. You can update your data as needed by running a procedure that creates a new dump of the required records and deletes the old disk file. This way, your Access

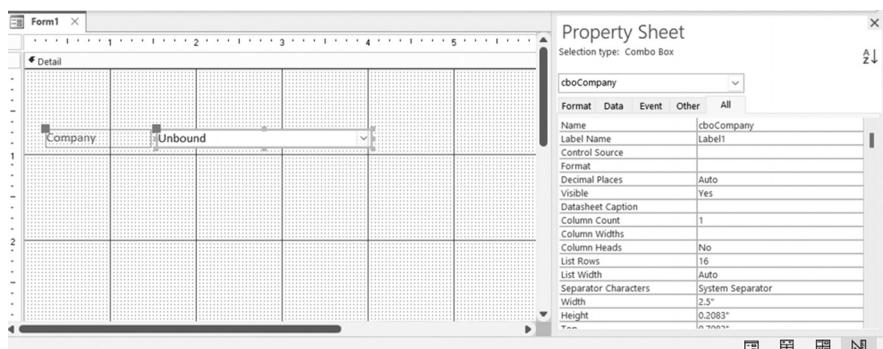
application can display the most recent data in its combo boxes or listboxes without having to connect to a database. Let's look at how you can fill a combo box with a saved recordset by working with Custom Project 12.1.



### Custom Project 12.1 Filling a Combo Box with a Disconnected Recordset

This custom project requires that you complete Hands-On 12.65.

1. In the Chap12.accdb database, choose **Create | Form Design** to create an Access form as shown in Figure 12.12. Use the Ribbon to locate the Combo box control and place it on the form. Cancel out from the Combo box wizard. Click the Property Sheet button on the Ribbon. Change the Name property of the unbound **Combo0** control to **cboCompany**. Click the **Combo0** label on the form and in the Property Sheet, set the **Caption** property of the label control to **Company**.



**FIGURE 12.12** This custom form is used to demonstrate how you can fill the combo box control with a disconnected recordset.

2. In the Property Sheet choose Form from the drop-down box and set the form's **Caption** property to **Disconnected combo**.
3. Set the Form's **Record Selectors** property to **No**.
4. Save the form as **frmFillCombo**.
5. In the form's Property Sheet, activate the **Event** tab, click next to the **On Load** event name, and click the ellipsis button. In the Choose Builder dialog box, select **Code Builder** and click **OK**.
6. Complete the **Form\_Load** procedure as shown here:

```
Private Sub Form_Load()
    Dim rst As ADODB.Recordset
    Dim strRowSource As String
```

```
Dim strName As String

strName = CurrentProject.Path & "\Companies.rst"

Set rst = New ADODB.Recordset
With rst
    .CursorLocation = adUseClient
    .Open strName, , , , adCmdFile
    Do Until .EOF
        strRowSource = strRowSource & rst!CompanyName & ";" 
        .MoveNext
    Loop
    With Me.cboCompany
        .RowSourceType = "Value List"
        .RowSource = strRowSource
        .SetFocus
    End With

    .Close
End With
Set rst = Nothing
End Sub
```

7. Open the frmFillCombo form in Form view.

To populate a combo box with values, the code in the Form\_Load procedure changes the RowSourceType property of the combo box control to Value List and sets the RowSource property to the string obtained by iterating through the recordset. When the form opens, its caption is changed to Disconnected combo, as shown in Figure 12.13.



FIGURE 12.13 After opening the form prepared in Custom Project 12.1, the combo box is filled with the names of companies obtained via a persisted recordset.

**8. Close the Disconnected combo (frmFillCombo) form.**

Persisted recordsets are especially handy when you need to support disconnected users or when you want to take data on the road with you. You can save the required set of records to a disk file, send it to your users in remote locations, or take it with you. While disconnected from the database, you or your users can view or modify the records. The next time you connect to the database you can update the original data with your changes using the BatchUpdate method. Custom Project 12.2 demonstrates this scenario.



### **Custom Project 12.2 Taking Persisted Data on the Road**

This custom project requires that you complete Hands-On 12.65.

---

#### ***Part 1: Saving a Recordset to Disk***

---

Before you can take a recordset on the road with you, you must save the records to a disk file. To create the data for this project, prepare and run the procedure in Hands-On 12.65. You should have the Companies.rst file available in your C:\VBAAccess2021\_ByExample folder before you proceed to Part 2.

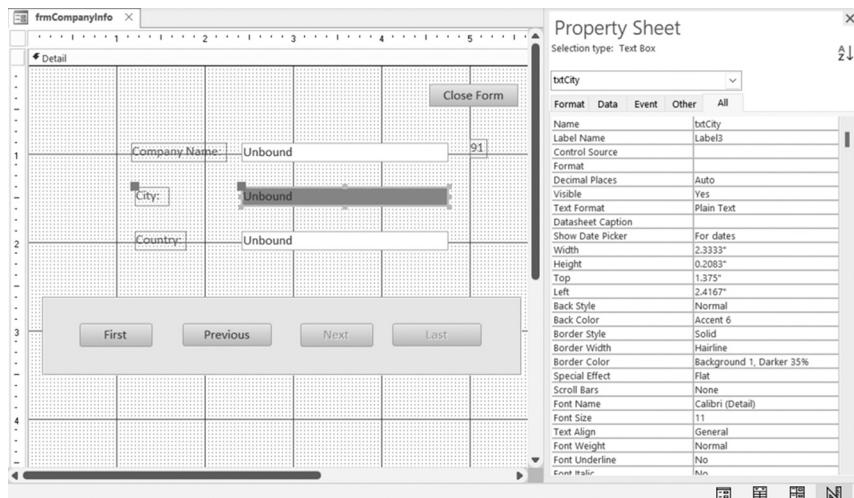
---

#### ***Part 2: Creating an Unbound Access Form to View and Modify Data***

---

Once you've saved the recordset to a disk file, the recordset becomes portable. You can take the file with you on the road or send it to someone else. Before either one of you can view the data and modify it, however, you need some sort of a user interface. In this part, like in the Custom Project 12.1, you will create an unbound Access form that will enable you to work with the file that contains the saved recordset.

1. Create a form as shown in Figure 12.14. Notice that this form contains only a couple of fields from the Customers table. This form serves only as an example. You can use as many fields as you have saved in the disk file.
2. Add three text boxes and five command buttons to the form.



**FIGURE 12.14** This custom form is used to demonstrate the use of the saved recordset in an unbound form.

**3.** Set the following properties for the form's text boxes controls:

Object	Property	Setting
1 <sup>st</sup> Unbound Text box label	Caption	Company Name:
1 <sup>st</sup> Unbound Text box	Name	txtCompany
2 <sup>nd</sup> Unbound Text box label	Caption	City:
2 <sup>nd</sup> Unbound Text box	Name Back Color	txtCity Accent 6
3 <sup>rd</sup> Unbound Text box label	Caption	Country:
3 <sup>rd</sup> Unbound Text box	Name	txtCountry
Command button 1	Name Caption	cmdFirst First
Command button 2	Name Caption	cmdPrevious Previous
Command button 3	Name Caption	cmdNext Next
Command button 4	Name Caption	cmdLast Last
Command button 5	Name Caption	cmdClose Close Form

**NOTE**

*We have set the Back Color property of the txtCity text box in the example application to visually indicate that the user can update only this field's data. You can select any color you like.*

4. To visually match the form in Figure 12.14, draw a rectangle control over the command buttons and set its **Back Color** property to any color you like. Select the rectangle and choose **Arrange | Send to Back** to move the rectangle behind the command buttons.
5. Add a label control to the right of the first text box (txtCompany) and set its Name property to **lblRecordNo** and its Caption property to **90**.
6. In the property sheet, select **Form** from the drop-down list and activate the **Format** tab. Set the following properties for the form:

Property Name	Setting
Scroll Bars	Neither
Record Selectors	No
Navigation Buttons	No

7. Save the form as **frmCompanyInfo**.

### **Part 3: Writing Procedures to Control the Form and Its Data**

---

Now that you've designed the form for your data, you need to write a couple of VBA procedures. The first procedure you'll write is an event procedure for the **Form\_Load** event. This procedure will load the form with data from the persisted file. You will start by declaring a module-level Recordset object variable called `rst` and a module-level Integer variable called `counter`. You will also write Click procedures for all the command buttons and a procedure to fill the text boxes with the data from the current record in the recordset. Let's get started!

1. In the form's property sheet ensure that the Form is selected from the drop-down. Activate the **Event** tab and click next to **On Load** event name. Click the Ellipsis button and, in the Choose Builder dialog box, select **Code Builder** and click **OK**.
2. Enter the code for the **Form\_Load** event procedure as shown here, starting with the declaration of module-level variables above the procedure:

```
Dim rst As ADODB.Recordset
Dim counter As Integer

Private Sub Form_Load()
```

```

Dim strFileName As String

strFileName = CurrentProject.Path & "\Companies.rst"
On Error GoTo ErrorHandler

Set rst = New ADODB.Recordset
With rst
    .CursorLocation = adUseClient
    .Open strFileName, , adOpenKeyset, _
        adLockBatchOptimistic, adCmdFile
End With

counter = 1
Call FillTxtBoxes(rst, Me)

With Me
    .txtCompany.SetFocus
    .cmdFirst.Enabled = False
    .cmdPrevious.Enabled = False
    .cmdLast.Enabled = True
    .cmdNext.Enabled = True
    .lbRecordNo.Caption = counter
End With

ExitHere:
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ": " & Err.Description
    If InStr(1, Err.Description, "cannot be found") > 0 Then
        Call SaveRecordsToDisk
    Resume 0
    End If
    Resume ExitHere
End Sub

```

The Form\_Load event procedure loads Companies.rst from a disk file. To fill the text boxes with the data from the current record in the recordset, you need to write the following code:

```

With Me
    .txtCompany = rst!CompanyName
    .txtCity = rst!City
    .txtCountry = rst!Country
End With

```

Because the preceding code will need to be entered in several procedures in this application, you can save yourself a great deal of typing by placing this code in a subroutine and calling it like this:

```
Call FillTxtBoxes(rst, Me)
```

This statement calls the subroutine named FillTxtBoxes and passes it two arguments: the Recordset object variable and the reference to the current form. The FillTxtBoxes procedure (see Step 3) is entered in a standard module and contains the code shown in the next step.

The counter variable, which was declared at the module level, is initialized to the value of 1. We will use this variable to control the display of command buttons on the form. The Form\_Load event procedure ends by setting the focus to the first text box (txtCompanyName) and disabling the first two command buttons. These buttons will not be required when the form first opens on the first record.

3. In the Visual Basic Editor Code window, choose **Insert | Module** and type the code of the following **FillTxtBoxes** procedure:

```
Sub FillTxtBoxes(ByVal rst As ADODB.Recordset, frm As Form)
    With frm
        .txtCompany = rst!CompanyName
        .txtCity = rst!City
        .txtCountry = rst!Country
    End With
End Sub
```

This procedure fills the three text boxes on the form with the data from the current record in the recordset. This procedure is called from the Form\_Load event procedure and the Click event procedures for each command button.

4. In the Form\_frmCompanyInfo Code window, type the following **Click** event procedure for the First command button:

```
Private Sub cmdFirst_Click()
    On Error GoTo Err_cmdFirst_Click

    If rst.Fields("City").OriginalValue <> Me.txtCity Then
        rst.Update "City", Me.txtCity
    End If
    rst.MoveFirst

    Call FillTxtBoxes(rst, Me)

    With Me
```

```
.txtCompany.SetFocus
.cmdFirst.Enabled = False
.cmdLast.Enabled = True
.cmdPrevious.Enabled = False
.cmdNext.Enabled = True
counter = 1
.lbRecordNo.Caption = counter
End With
Exit_cmdFirst_Click:
Exit Sub
Err_cmdFirst_Click:
MsgBox Err.Description
Resume Exit_cmdFirst_Click
End Sub
```

5. In the Form\_frmCompanyInfo Code window, type the following **Click** event procedure for the Next command button:

```
Private Sub cmdNext_Click()
On Error GoTo Err_cmdNext_Click

If rst.Fields("City").OriginalValue <> Me.txtCity Then
    rst.Update "City", Me.txtCity
End If
rst.MoveNext
counter = counter + 1

Me.cmdFirst.Enabled = True

Call FillTxtBoxes(rst, Me)

Me.cmdPrevious.Enabled = True
Me.lbRecordNo.Caption = counter
Me.txtCompany.SetFocus
If counter = rst.RecordCount Then
    Me.cmdNext.Enabled = False
    Me.cmdLast.Enabled = False
End If

Exit_cmdNext_Click:
Exit Sub
Err_cmdNext_Click:
MsgBox Err.Description
Resume Exit_cmdNext_Click
End Sub
```

6. In the Form\_frmCompanyInfo Code window, type the following **Click** event procedure for the Previous command button:

```
Private Sub cmdPrevious_Click()
    On Error GoTo Err_cmdPrevious_Click

    If rst.Fields("City").OriginalValue <> Me.txtCity Then
        rst.Update "City", Me.txtCity
    End If
    rst.MovePrevious
    counter = counter - 1

    Call FillTxtBoxes(rst, Me)

    With Me
        .txtCompany.SetFocus
        .cmdLast.Enabled = True
        .cmdNext.Enabled = True
        .lbRecordNo.Caption = counter
    End With
    If counter = 1 Then
        Me.cmdFirst.Enabled = False
        Me.cmdPrevious.Enabled = False
    End If

    Exit_cmdPrevious_Click:
    Exit Sub
Err_cmdPrevious_Click:
    MsgBox Err.Description
    Resume Exit_cmdPrevious_Click
End Sub
```

7. In the Form\_frmCompanyInfo Code window, type the following **Click** event procedure for the Last command button:

```
Private Sub cmdLast_Click()
    On Error GoTo Err_cmdLast_Click

    If rst.Fields("City").OriginalValue <> Me.txtCity Then
        rst.Update "City", Me.txtCity
    End If
    rst.MoveLast

    Call FillTxtBoxes(rst, Me)

    With Me
```

```

.txtCompany.SetFocus
.cmdFirst.Enabled = True
.cmdPrevious.Enabled = True
.cmdLast.Enabled = False
.cmdNext.Enabled = False
End With
counter = rst.RecordCount
Me.lbRecordNo.Caption = counter
Exit_cmdLast_Click:
Exit Sub
Err_cmdLast_Click:
MsgBox Err.Description
Resume Exit_cmdLast_Click
End Sub

```

- 8.** In the Form\_frmCompanyInfo Code window, type the following **Click** event procedure for the Close Form command button:

```

Private Sub cmdClose_Click()
    DoCmd.Close acForm, "frmCompanyInfo", acSaveYes
End Sub

```

When the Close Form button is clicked Access will execute the code in the cmdClose\_Click procedure and proceed to call the code in the Form\_Unload event procedure shown in Step 9.

Notice that all the Click event procedures you prepared in Steps 4–7 contain the following statement:

```

If rst.Fields("City").OriginalValue <> Me.txtCity Then
    rst.Update "City", Me.txtCity
End If

```

This statement updates the value of the City field in the recordset with the current value found in the txtCity text box on the form whenever you make a change to the City field as you move through the records. Although the user can enter data in other text boxes, all modifications are ignored as there is no code in the Click event procedures that will allow changes to fields other than City. Of course, you can easily change this behavior by adding the necessary lines of code. Depending on which button was clicked, certain command buttons are disabled, and others are enabled. This gives the user a visual clue of what actions are allowed at a particular moment.

To make the form work, we need to write one more event procedure. Before closing the form, we must make sure that the changes to the City field in the current record are saved and all changes in the City field we made while

working with the form data are written back to the disk file. In other words, we must replace the Companies.rst disk file with a new file. This is done in the Form\_Unload event procedure as shown in the next Step.

9. In the Form\_frmCompanyInfo Code window, type the code of the **Form\_Unload** event procedure as shown here:

```
Private Sub Form_Unload(Cancel As Integer)
    If rst.Fields("City").OriginalValue <> Me.txtCity Then
        rst.Update "City", Me.txtCity
    End If
    rst.Save CurrentProject.Path & "\Companies.rst", _
        adPersistADTG
End Sub
```

ADO Recordsets have a special property called `OriginalValue`, which is used for storing original values that were retrieved from a database. These original values are left unchanged while you edit the recordset offline. Any changes to the data made locally are recorded using the `Value` property of the Recordset object. The `OriginalValue` property is updated with the values changed locally when you reconnect to the database and perform an `UpdateBatch` operation (see Part 5 in this custom project).

The `Form_Unload` event occurs when you attempt to close a form but before the form is removed from the screen. This is a good place to perform those operations that must be executed before the form is closed. In the `Form_Unload` procedure, we use the Recordset's `OriginalValue` property to check whether changes were made to the content of the `City` field in the current record. If `OriginalValue` is different from the value found in the current record's `txtCity` text box, we want to save the record by using the `Update` method of the recordset. Next, we save the current recordset to a file with the same name.

#### ***Part 4: Viewing and Editing Data Offline***

---

Now that you've written all the procedures for the custom application, let's begin using the form to view and edit the data.

1. Open the **frmCompanyInfo** form in Form view.
2. In the first record, type another German city name.
3. Click the **Last** button, and enter another Polish city name.
4. Click the **First** button and notice that the value of `City` matches what you entered in step 2.
5. Use the **Next** button to move to the fourth record and enter another name for a city in UK.

6. Click the Close Form button to close the form.
7. Reopen the form and check if the values in the City text box in the first, fourth, and last records match your previous entries.
8. Close the frmCompanyInfo form.

#### ***Part 5: Connecting to a Database to Update the Original Data***

---

After you've made changes to the data by using the custom form, you can send the file with the modified recordset to any of your database administrators so that they can update the underlying database with your changes. Let's write a procedure that will take care of the update.

<b>NOTE</b>	<i>The procedure that you are about to write will modify the Customers table in the Northwind_Chap12.mdb database. I recommend that you take a few minutes now and create a copy of this database so that you can restore the original data later if necessary.</i>
-------------	---

1. In the Visual Basic Editor window of Chap12.accdb database, choose **Insert | Module**.
2. In the module's Code window, type the following **UpdateDb** procedure:

```
Sub UpdateDb()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strNorthPath As String
    Dim strRecStat As String

    On Error GoTo ErrorHandler
    strNorthPath = CurrentProject.Path & "\Northwind_Chap12.mdb"

    ' Open the connection to the database
    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .ConnectionString = "Data Source = " & strNorthPath
        .Mode = adModeReadWrite
        .Open
    End With

    ' Open the recordset from the local file
    ' that was persisted to the hard drive
    ' and update the data source with the changes
```

```
Set rst = New ADODB.Recordset
With rst
    .CursorLocation = adUseClient
    .Open CurrentProject.Path & "\Companies.rst", conn, _
        adOpenKeyset, adLockBatchOptimistic, adCmdFile
    .UpdateBatch adAffectAll

    ' Check if there were records with conflicts
    ' during the update
    .Filter = adFilterAffectedRecords
    Do Until .EOF

        If !city.Status = adRecOK Then
            Debug.Print !CustomerID & _
                " city was successfully updated to " & _
                !city & "." & vbCrLf
        End If

        strRecStat = strRecStat & !CustomerID & "->" & _
            !city & ":" & !city.Status & vbCrLf
        .MoveNext
    Loop
    .Close
    If strRecStat <> "" Then
        Debug.Print strRecStat
    Else
        Debug.Print "The Recordset was not changed."
    End If
End With

ExitHere:
Set rst = Nothing
Set conn = Nothing
    ' Generate a new Companies.rst file with
    ' the updated recordset so it is available
    ' the next time you want to work with the form
Call SaveRecordsToDisk

    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ": " & Err.Description
    Resume ExitHere
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

In the `UpdateDb` procedure, we used the `UpdateBatch` method of the ADO Recordset object to update the underlying database with the changes we made to the data while working with it offline using the `frmCompanyInfo` form. The `UpdateBatch` method takes an optional parameter that determines how many records will be affected by the update. This parameter can be one of the constants shown in Table 12.7.

TABLE 12.7 Enumerated constants used with the `UpdateBatch` method

Constant	Value	Description
<code>adAffectCurrent</code>	1	Pending changes will be written only for the current record.
<code>adAffectGroup</code>	2	Pending changes will be written for the records that satisfy the current filter.
<code>adAffectAll</code>	3	Pending changes will be written for all the records in the recordset. This is the default.

When you update the data, your changes are compared with values that are currently in the database table. The update will fail if the record was deleted or updated in the underlying database since the recordset was saved to disk. Therefore, after calling the `UpdateBatch` method, you should check the status of the records to locate records with conflicts. To do this, we must filter the recordset to see only the affected records:

```
rst.Filter = adFilterAffectedRecords
```

Next, we loop through the recordset and check the `Status` property of each record. This property can return different values, as shown in Table 12.8. You can locate these values in the Object Browser by typing `RecordStatusEnum` in the Search box.

TABLE 12.8 `RecordStatusEnum` constants returned by the `Status` property

Constant	Value	Description
<code>adRecCanceled</code>	256	The record was not saved because the operation was canceled.
<code>adRecCantRelease</code>	1024	The new record was not saved because the existing record was locked.
<code>adRecConcurrencyViolation</code>	2048	The record was not saved because optimistic concurrency was in use.
<code>adRecDBDeleted</code>	262144	The record has already been deleted from the data source.

Constant	Value	Description
adRecDeleted	4	The record was deleted.
adRecIntegrityViolation	4096	The record was not saved because the user violated integrity constraints.
adRecInvalid	16	The record was not saved because its bookmark is invalid.
adRecMaxChangesExceeded	8192	The record was not saved because there were too many pending changes.
adRecModified	2	The record was modified.
adRecMultipleChanges	64	The record was not saved because it would have affected multiple records.
adRecNew	1	The record is new.
adRecObjectOpen	16384	The record was not saved because of a conflict with an open storage object.
adRecOK	0	The record was successfully updated.
adRecOutOfMemory	32768	The record was not saved because the computer has run out of memory.
adRecPendingChanges	128	The record was not saved because it refers to a pending insert.
adRecPermissionDenied	65536	The record was not saved because the user has insufficient permissions.
adRecSchemaViolation	131072	The record was not saved because it violates the structure of the underlying database.
adRecUnmodified	8	The record was not modified.

If the record was successfully updated then its Status property is adRecOK (see Table 12.8 above) and we print a message to the Immediate window. We also collect all the information about the updates in the strRecStat variable and print it to the Immediate Window.

While iterating through the recordset you can add additional code to resolve any encountered conflicts or check, for example, the original value and the updated value of the fields in updated records. As mentioned earlier, the `OriginalValue` property returns the field value that existed prior to any changes (since the last `Update` method was called). You can cancel all pending updates by using the `CancelBatch` method.

When you execute the `UpdateDb` procedure, your changes are written to the database. At the end of the procedure, we call the `SaveRecordsToDisk` procedure to generate the updated disconnected recordset.

4. Open the Northwind\_Chap12.mdb database and review the content of the City field in the Customers table. You should see the changes you made in the first, fourth, and last records. The same changes should be visible in the linked Customers table in the current Chap12.accdb database.
5. Close the Northwind\_Chap12.mdb database and the Access window in which it was displayed. Do not close the Chap12.accdb database.

This completes Custom Project 16.2 in which you learned how to:

- Save the recordset to disk with the `Save` method
- Create a custom form to view and edit the recordset data in the disk file
- Open the recordset from disk with the `Open` method
- Work with the recordset offline (view and edit data)
- Reopen the connection to the original database and write your changes with the `UpdateBatch` method

**NOTE**

*Refer to Chapter 26, “XML Features in Access 2021” to find out how you can save a recordset in XML file using the `adPersistXML` format.*

### Cloning a Recordset

Sometimes you may want to manipulate a recordset without losing the current position in the recordset. You can do this by *cloning* your original recordset. Use the ADO `Clone` method to create a recordset that is a copy of another recordset. You can create a recordset clone like this:

```
Dim rstOrg As ADODB.Recordset ' your original recordset
Dim rstClone As ADODB.Recordset      ' cloned recordset

Set rstClone = rstOrg.Clone
```

As you can see from the assignment statement, the `rstClone` object variable contains a reference to the original recordset. After you've used the `Clone` method, you end up with two copies of the recordset that contain the same records but can be filtered and manipulated separately. You can create more than one clone of the original recordset.

Use the `Clone` method when you want to perform an operation on a recordset that requires multiple current records. The `Clone` object and the original `Recordset` object each have their own current records; therefore, the record pointers in the original and cloned recordsets can move independently of one another. And, because the `clone` points to the same set of data as the original,

any changes made using either the original recordset or any of its clones will be visible in the original and its clones. However, the original recordset and its clones can get out of sync if you requery the original recordset against the database. When you close the original recordset, the clones remain open until you close them. Closing any of the clones does not close the original recordset.

Because the `Clone` method does not create another copy of the data (it only points to the data), cloning a recordset is faster and more efficient than opening a second recordset based on the same criteria. A recordset created by a method other than cloning will have a different set of bookmarks than the original recordset, even when it is based on the same SQL statement.

You can make a clone read-only by using an optional parameter like this:

```
Set rstClone = rstOrg.Clone(adLockReadOnly)
```

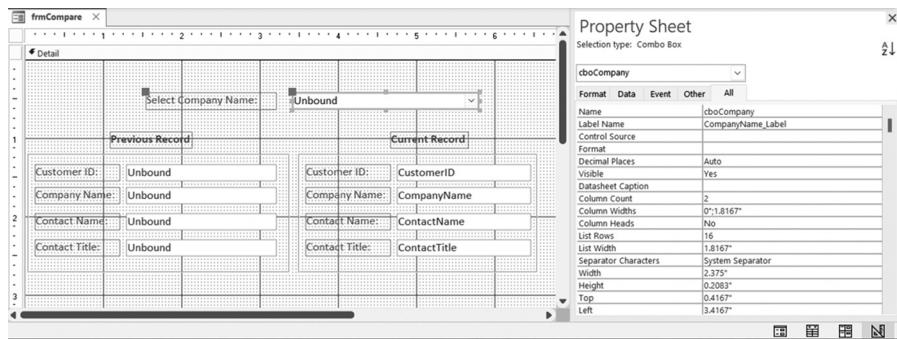
It's worth mentioning that you can only clone bookmarkable recordsets. Use the Recordset object's `Supports` method to find out if the recordset supports bookmarks (see the "Using Bookmarks" section in this chapter). If you try to clone a non-bookmarkable recordset, you will receive a runtime error. The clone and the original recordset have the same bookmarks, which you can share. A bookmark reference from one Recordset object refers to the same record in any of its clones.

Custom Project 12.3 demonstrates how the `Clone` method can be used to create a single form for displaying the current and previous records side by side (see Figure 12.15).



### Custom Project 12.3 Displaying the Contents of the Current and Previous Record by Using the Clone Method

1. In the main Access window of the Chap12.accdb database make sure that you have a table called **Customers**. If this table is missing, choose **External Data | Access**. In the File name box of the Get External Data dialog box, enter **C:\VBAAccess2021\_ByExample\Northwind\_Chap12.mdb**, and then click **OK**. In the Import Objects window, select the **Customers** table and click **OK**. Click **Close** to exit the Get External Data dialog box.
2. Choose **Create | Form Design** and create a form like the one depicted in Figure 12.15. The following steps will help you set up the form and various control properties.



**FIGURE 12.15** This custom form is used to demonstrate how the recordset cloning is used to read the contents of the previous record.

3. In the Controls area of the Form Design tab, click the **Combo Box** control and click inside the form area to position it at the upper right as shown in Figure 12.15. In the Combo Box Wizard's first screen, choose the option button labeled **I want the combo box to look up the values in a table or query**. Click **Next**. Make sure the **Customers** table is selected and click **Next**. The fields available in the Customers table should appear. Move **CustomerID** and **CompanyName** from the Available Fields box to the Selected Fields box, and then click **Next**. Specify **CompanyName** as the Ascending sort order for your combo box, and then click **Next**. In the next wizard dialog, adjust the width of the combo box column to fit the longest company name and click **Finish**. Now you should see the combo box placed on your form. Choose the Property Sheet in the Tools group of the Form Design Ribbon and set the **Caption** property of the **CompanyName\_Label** to **Select Company Name**. Set the **Tag** property of this label control to **cbo**. Set the **Name** property of the Company combo (unbound) to **cboCompany** and its **Tag** property to **cbo**.
4. Place two label controls on the form. Set the **Caption** property of the first one to **Previous Record** and set the **Caption** property of the second one to **Current Record**. Set the **Tag** property of the Previous Record label to **PrevRec**. Adjust the size of both label controls so the captions are fully visible.
5. Place four text boxes below the Previous Record label and set their properties as shown below:

Object	Property	Setting
1 <sup>st</sup> Label in front of Text box 1	Caption Tag	Customer ID: PrevRec

Object	Property	Setting
Text box 1	Name Tag Control Source	CustIDPrev PrevRec should be blank
2 <sup>nd</sup> Label in front of Text box 2	Caption Tag	Company Name: PrevRec
Text box 2	Name Tag Control Source	CompanyPrev PrevRec should be blank
3 <sup>rd</sup> Label in front of Text box 3	Caption Tag	Contact Name: PrevRec
Text box 3	Name Tag Control Source	ContactPrev PrevRec should be blank
4 <sup>th</sup> Label in front of Text box 4	Caption Tag	Contact Title: PrevRec
Text box 4	Name Tag Control Source	TitlePrev PrevRec should be blank

6. Place a rectangle control over the set of four text boxes that you have created. Set the **Tag** property of the rectangle to **PrevRec**.
7. Copy all the controls that appear below Previous Record label and paste them below the Current Record label. You can quickly select all the controls by dragging the mouse in the ruler area. Press **Ctrl+Z** if you need to cancel the action and try again.
8. Set the properties of the pasted text box controls as shown below:

Customer ID text box	Name Control Source	CustomerID CustomerID
Company Name text box	Name Control Source	CompanyName CompanyName
Contact Name text box	Name Control Source	ContactName ContactName
Contact Title text box	Name Control Source	ContactTitle ContactTitle

9. In the property sheet, select **Form** from the drop-down box and set the following form properties:

Property Name	Setting
Record Source	Customers
Caption	Record Comparison
Scroll Bars	Neither
Record Selectors	No
Navigation Buttons	No

Notice that the Record Source property of the Form is set to the Customers table. If the Record Source is left empty, you will get an error 91: Object variable or With block variable not set.

10. Save the form as **frmCompare**.

11. Click the combo box control on the form to select it. Activate the **Event** tab in the property sheet and click to the right of the **AfterUpdate** event name. Select **[Event Procedure]** from the drop-down box, and then click the **Build** button (...) to activate the Code window. Complete the **cboCompany\_AfterUpdate** procedure shown here:

```
Private Sub cboCompany_AfterUpdate()
    ' Find the record that matches the control.
    Dim rs As Object
    Dim c As Control

    On Error GoTo ErrHandle

    Set rs = Me.Recordset.Clone
    rs.FindFirst "[CustomerID] = '" & Me! [cboCompany] & "'"
    If Not rs.EOF Then Me.Bookmark = rs.Bookmark
        ' Move to the previous record in the clone
        ' so that we can load the previous records'
        ' data in the form's text boxes
        rs.MovePrevious
    If Not rs.BOF Then
        For Each c In Me.Controls
            c.Visible = True
        Next
        With Me
            .CustIdPrev = rs.Fields(0).Value
            .CompanyPrev = rs.Fields(1)
            .ContactPrev = rs.Fields(2)
            .TitlePrev = rs.Fields(3)
        End With
    End If
End Sub
```

```

        End With
    Else
        For Each c In Me.Controls
            If c.Tag = "PrevRec" Then
                c.Visible = False
            End If
        Next
    End If
    ExitHere:
    Exit Sub
ErrHandle:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub

```

Notice that this event procedure begins by creating a clone of the form's recordset. Next, the `FindFirst` method is used to locate the customer record based on the entry selected in the combo box. To ensure that the form's record is in sync with the entry selected in the combo box, the following line of code moves the form's bookmark to the same location as the recordset clone's bookmark if we are not at the end of file (EOF):

```
If Not rs.EOF Then Me.Bookmark = rs.Bookmark
```

Next, the procedure ensures that the controls used to display the contents of the previous record are visible whenever the selected record is not the first record. The control's Tag property is used to allow easy selection of controls that need to be hidden or made visible.

12. Press **Ctrl+S** to save the current changes.
13. Test your form by opening it in Form view. Selecting a company name from the combo box should fill the text boxes under the Current Record label with the selected company's data. The boxes under the Previous Record label should pull company data from the previous record.

Before we start working with this custom project, let's write a `Form_Load` event procedure to ensure that only the combo box and its label are visible when the form is opened.

14. In the Code window where you have written the `cboCompany_AfterUpdate` event procedure, select **Form** from the object drop-down box in the top-left corner. Select **Load** from the procedure drop-down box on the right. Complete the code for the `Form_Load` event procedure as shown here:

```

Private Sub Form_Load()
    Dim c As Control

```

```
For Each c In Me.Controls
    If c.Tag <> "cbo" Then
        c.Visible = False
    End If
Next
End Sub
```

15. Make sure there are no errors in your code by choosing **Debug | Compile Chap12**.
16. Save and close your form.
17. Open the **frmCompare** form in the Form view and test it by choosing various company names from the combo box.
18. Close the form.  
Think of ways to improve this form. For example, add a set of controls and write additional code to display the next record.

## INTRODUCTION TO DATA SHAPING

---

Designing database applications often requires that you pull information from multiple tables. For instance, to obtain a listing of customers and their orders, you must link the required tables with SQL **JOIN** statements as shown here:

```
SELECT Customers.CustomerID AS [Cust Id],
       Customers.CompanyName,
       Orders.OrderDate,
       [Order Details].OrderID,
       Products.ProductName,
       [Order Details].UnitPrice,
       [Order Details].Discount,
       CCur([Order Details].[UnitPrice]*[Quantity]*(1-
[Discount])/100)*100
          AS [Extended Price]
FROM Products
INNER JOIN ((Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN [Order Details]
ON Orders.OrderID = [Order Details].OrderID)
ON Products.ProductID = [Order Details].ProductID
ORDER BY Customers.CustomerID, Orders.OrderDate DESC;
```

When you execute this SQL statement in the Northwind\_Chap12.mdb database, your output will match Figure 12.16.

The screenshot shows the Microsoft Access interface with the 'External Data' tab selected. In the center, a grid displays a flat recordset resulting from a SQL JOIN statement. The columns are Customer ID, Company Name, Order Date, Order ID, Product Name, Unit Price, Discount, and Extended Price. The data includes multiple entries for the same customer (e.g., ALFKI) with different products and dates. The left pane shows the 'All Access Objects' browser with 'qrySQLJOIN' selected.

Customer ID	Company Name	Order Date	Order ID	Product Name	Unit Price	Discount	Extended Price
ALFKI	Alfreds Futterkiste	09-Apr-1998	11011	Fletenmyosot	\$21.50	0%	\$430.00
ALFKI	Alfreds Futterkiste	09-Apr-1998	11011	Escargots de Bourgogne	\$13.25	5%	\$503.50
ALFKI	Alfreds Futterkiste	16-Mar-1998	10952	Rössle Sauerkraut	\$45.60	0%	\$91.20
ALFKI	Alfreds Futterkiste	16-Mar-1998	10952	Grandma's Boysenberry Spread	\$25.00	5%	\$380.00
ALFKI	Alfreds Futterkiste	15-Jan-1998	10835	Original Frankfurter grüne Soße	\$13.00	20%	\$20.80
ALFKI	Alfreds Futterkiste	15-Oct-1997	10703	Lakkalkööri	\$38.00	0%	\$770.00
ALFKI	Alfreds Futterkiste	08-Oct-1997	10703	Angostura Syrup	\$20.00	0%	\$400.00
ALFKI	Alfreds Futterkiste	08-Oct-1997	10692	Apple-spread	\$43.90	0%	\$878.00
ALFKI	Alfreds Futterkiste	25-Aug-1997	10643	Chartreuse verte	\$18.00	25%	\$283.50
ALFKI	Alfreds Futterkiste	25-Aug-1997	10643	Spirituosa	\$32.00	25%	\$800.00
ALFKI	Alfreds Futterkiste	25-Aug-1997	10643	Rödeläder Sauerkraut	\$45.60	25%	\$513.00
ANATR	Ana Trujillo Emparedados y helados	04-Mar-1998	10926	Konke	\$6.00	0%	\$60.00
ANATR	Ana Trujillo Emparedados y helados	04-Mar-1998	10926	Mozzarella di Giovanni	\$34.80	0%	\$348.00
ANATR	Ana Trujillo Emparedados y helados	04-Mar-1998	10926	Teatime Chocolate Biscuits	\$9.20	0%	\$64.40
ANATR	Ana Trujillo Emparedados y helados	04-Mar-1998	10926	Queso Cabrales	\$21.00	0%	\$42.00
ANATR	Ana Trujillo Emparedados y helados	28-Nov-1997	10759	Mascarpone Fabioli	\$32.00	0%	\$320.00
ANATR	Ana Trujillo Emparedados y helados	08-Aug-1997	10625	Singaporean Hokkien Fried Mee	\$14.00	0%	\$70.00
ANATR	Ana Trujillo Emparedados y helados	08-Aug-1997	10625	Camembert Pierrot	\$34.00	0%	\$340.00
ANATR	Ana Trujillo Emparedados y helados	08-Aug-1997	10625	Tofu	\$23.25	0%	\$69.75

FIGURE 12.16 When you use SQL JOIN statements you get a flat recordset with a lot of duplicate information.

When you output your data in a standard way by using the SQL `JOIN` syntax, you get a lot of duplicate information. You can eliminate this redundant information by using an advanced feature of ADO known as a shaped (or hierarchical) recordset.

*Data shaping* allows you to create recordsets within recordsets with a single ADO object. This sort of hierarchical data arrangement is often seen as a *parent-child relationship*. The parent recordset contains the child recordset. A child recordset can contain another child recordset, which is a grandchild of the original recordset. A parent-child relationship can be placed in an easy-to-read tree structure. You will produce such a structure in Custom Project 12.4 later in this chapter. For now, let's focus on learning some new concepts that will enable you to present your data in a format that's easy to view and navigate.

### Writing a Simple SHAPE Statement

You can easily create a hierarchy of data by using a data shaping language. All you need to know is how to use the following three commands: `SHAPE`, `APPEND`, and `RELATE`. The basic syntax looks like this:

```
SHAPE {parent-command}
APPEND ({child-command} [[AS] table-alias]
RELATE (parent-column TO child-column)
```

`parent-command` and `child-command` are often SQL `SELECT` statements that pull the data from the required tables. Let's look at the following example that uses the preceding syntax:

```
SHAPE {SELECT CustomerID AS [Cust Id],  
CompanyName AS (Company) Customers}  
APPEND ({SELECT CustomerId, OrderDate, OrderId,  
Freight FROM Orders} AS custOrders  
RELATE (CustomerID TO CustomerID)
```

The preceding statement is a shaped recordset. This statement selects two fields from the Customers table and four fields from the Orders table. By using this `SHAPE` statement, you can list all orders for each of the customers in the Customers table without returning any redundant information.

Notice that there are two `SELECT` statements in this recordset:

- The first `SELECT` statement is the parent recordset. This recordset retrieves the data from the Customers table. Notice this `SELECT` statement is surrounded by curly braces and preceded by the `SHAPE` command, which defines a recordset.
- The second `SELECT` statement is the child recordset. It gets the data from the Orders table. Notice that this `SELECT` statement is also surrounded by curly braces; however, it is preceded by the `APPEND` clause and an opening parenthesis. The `APPEND` clause will add the child recordset to the parent.

<b>NOTE</b>	<i>When you append a child recordset to the parent recordset, a new field (column) is created in a parent recordset. This field is called a chapter column and has a data type called adChapter. You can use the AS clause to assign a name to the chapter column. If the appended column has no chapter alias, a name will be generated for it automatically. In our example, the chapter column is called custOrders. Always specify an alias for your child recordset if you are planning to refer to it later in your code.</i>
-------------	---

After specifying the `SELECT` statement for the child recordset, you must indicate how you want the two recordsets to be linked. You do this with the `RELATE` clause. The column (`CustomerID`) from the parent recordset is related to the column (`CustomerID`) of the child recordset. Notice that you don't have to specify table names in the `RELATE` clause. Always specify the name of the parent column first.

<b>NOTE</b>	<i>The fields you use to relate parent and child recordsets must be in both recordsets. For example, you could not relate both recordsets if you did not select CustomerID from the Orders table.</i>
-------------	---

Finally, remember to place a closing parenthesis at the end of the statement.

## Working with Data Shaping

---

To work with data shaping in your VBA procedure, you need two providers: one for the data shaping functionality and the other for the data itself. Therefore, before you can create shaped (hierarchical) recordsets in your programs, you will need to specify:

- The name of a service provider

The data shaping functionality is provided by the data shaping service for OLE DB. The name of this service provider is `MSDataShape` and it is specified as the value of the `Connection` object's `Provider` property like this:

```
conn.Provider = «MSDataShape»
```

or it can be a connection string like this:

```
"Provider=MSDataShape"
```

- The name of a data provider

Because a shaped recordset needs to be populated with rows of data, you must specify the name of a data provider as the value of the `DataProvider` property of the `Connection` object:

```
conn.DataProvider = «Microsoft.ACE.OLEDB.12.0;»
```

or in the connection string like this:

```
"Data Provider=Microsoft.ACE.OLEDB.12.0;"
```

The following is a code fragment from the procedure in Hands-On 12.66 that demonstrates how to specify the names of the data and service providers:

```
' define database connection string
' using the OLE DB provider
' and Northwind_Chap12.mdb database as Data Source

strConn = "Data Provider=Microsoft.ACE.OLEDB.12.0;"
strConn = strConn & "Data Source = " &
          "C:\VBAAccess2021_ByExample\Northwind_Chap12.mdb"

' specify Data Shaping provider
' and open connection to the database
Set conn = New ADODB.Connection
With conn
    .ConnectionString = strConn
    .Provider = "MSDataShape"
```

```
.Open  
End With
```

### SIDE BAR *Data Shaping with Other Databases*

The data shaping service creates a shaped (hierarchical) recordset from any data supplied by a data provider. In order to provide shaped data from a database other than Access, let's say, an SQL Server database, a connection string might look like this:

```
Dim conn As ADODB.Connection  
Set conn = New ADODB.Connection  
conn.Open = "Provider=MSDataShape;" & _  
    "Data Provider=SQLOLEDB;" & _  
    "Server=myServerName;" & _  
    "Initial Catalog=Northwind;" & _  
    "User ID=myId;Password="
```

or like this:

```
Dim conn As ADODB.Connection  
Set conn = New ADODB.Connection  
conn.Provider = "MSDataShape"  
conn.Open "Data Provider=SQLOLEDB;" & _  
    "Integrated Security=SSPI;" & _  
    "Database=Northwind"
```

In Hands-On 12.66, you learn how to create a shaped recordset in a VBA procedure and display hierarchical data in the Immediate window (see Figure 12.17).



### Hands-On 12.66 Creating a Shaped Recordset

1. In the Visual Basic Editor window of the Chap12.accdb database, choose **Insert | Module**.
2. In the module's Code window, enter the **ShapeDemo** procedure shown here:

```
Sub ShapeDemo()  
    Dim conn As ADODB.Connection  
    Dim rst As ADODB.Recordset  
    Dim rstChapter As Variant  
    Dim strConn As String  
    Dim shpCmd As String  
  
    ' define database connection string  
    ' using the OLE DB provider  
    ' and Northwind_Chap12.mdb database as Data Source
```

```
strConn = "Data Provider=Microsoft.ACE.OLEDB.12.0;"  
strConn = strConn & "Data Source = " &  
    "C:\VBAAccess2021_ByExample\Northwind_Chap12.mdb"  
  
' specify Data Shaping provider  
' and open connection to the database  
Set conn = New ADODB.Connection  
With conn  
    .ConnectionString = strConn  
    .Provider = "MSDataShape"  
    .Open  
End With  
  
' define the SHAPE command for  
' the shaped recordset  
shpCmd = "SHAPE " &  
    "{SELECT CustomerID AS [Cust Id], " &  
    " CompanyName AS Company FROM Customers}" &  
    " APPEND ({SELECT CustomerID, OrderDate," &  
    " OrderID, Freight FROM Orders}" &  
    " AS custOrders" &  
    " RELATE [Cust Id] TO CustomerID)"  
  
' create and open the parent recordset  
' using the open connection  
Set rst = New ADODB.Recordset  
rst.Open shpCmd, conn  
  
' output data from the parent recordset  
Do While Not rst.EOF  
    Debug.Print rst("Cust Id"); _  
    Tab; rst("Company")  
    rstChapter = rst("custOrders")  
    ' write out column headings  
    ' for the child recordset  
    Debug.Print Tab; _  
        "OrderDate", "Order #", "Freight"  
    ' output data from the child recordset  
    Do While Not rstChapter.EOF  
        Debug.Print Tab; _  
            rstChapter("OrderDate"), _  
            rstChapter("OrderID"), _  
            Format(rstChapter("Freight"), "$ #.##")  
        rstChapter.MoveNext  
    Loop
```

```

        rst.MoveNext
Loop

' Cleanup
rst.Close
Set rst = Nothing
Set conn = Nothing
End Sub

```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

This procedure begins by specifying the data provider and data source name in the `strConn` variable. Next, we define a new ADO Connection object and set the `ConnectionString` property of this object to the `strConn` variable. Now that we have the data provider name and also know which database we need to pull the data from, we specify the data shaping service provider. This is done by using the `Provider` property of the `Connection` object. We set this property to `MSDataShape`, which is the name of the service provider for the hierarchical recordsets. Now we are ready to actually open a connection to the database. Before we can pull the required data from the database, we define the shaped recordset statement and store it in the `shpCmd` String variable. Next, we create a new Recordset object and open it using the open database connection. Then, we populate it with the content of the `shpCmd` variable like this:

```

Set rst = New ADODB.Recordset
rst.Open shpCmd, conn

```

Now that we have filled the hierarchical recordset, we begin to loop through the parent recordset. The first statement in the loop:

```
Debug.Print rst("Cust Id"); Tab; rst("Company")
```

will write out the customer ID (Cust Id) and the company name (Company) to the Immediate window.

In the second statement in the loop:

```
rstChapter = rst("custOrders")
```

we create a Recordset object variable based on the value of the `custOrders` field. As you recall from an earlier discussion, `custOrders` is an alias for the child recordset. The object variable (`rstChapter`) can be any name you like as long as it's not a VBA keyword.

**NOTE**

*Because a child recordset is simply a field in a parent recordset, when you retrieve the value of that field you will get the entire recordset filtered to include only the related records.*

Before iterating through the child recordset, the column headings are output to the Immediate window for the fields we want to display. This way it is much easier to understand the meaning of the data in the child recordset. The next block of code loops through the child recordset and dumps the data to the Immediate window under the appropriate column heading. Once the data is retrieved for each parent record, we can close the recordset and release the memory.

The screenshot shows the Microsoft Visual Studio IDE. On the left is the Immediate window displaying a hierarchical recordset with data for customers ALFKI, ANATR, ANTON, and AROUT. On the right is the code editor for the 'ShapeDemo' procedure. The code uses the SHAPE command to append a child recordset ('custOrders') to the parent recordset ('rst'). It then iterates through the child recordset, printing its columns ('Order Date', 'Order #', 'Freight') under the parent's 'Company' column.

```

Immediate
ALFKI    Alfreds Futterkiste
OrderDate Order #   Freight
8/25/1997 10643   $ 29.46
10/3/1997 10692   $ 61.02
10/13/1997 10702   $ 23.94
1/15/1998 10835   $ 69.53
3/16/1998 10952   $ 40.42
4/9/1998 11011   $ 1.21
ANATR    Ana Trujillo Emparedados y helados
OrderDate Order #   Freight
9/18/1996 10308   $ 1.61
8/8/1997 10625   $ 43.9
11/28/1997 10759   $ 11.99
3/4/1998 10926   $ 39.92
ANTON   Antonio Moreno Taqueria
OrderDate Order #   Freight
11/27/1996 10365   $ 22.
4/15/1997 10507   $ 47.45
5/13/1997 10535   $ 15.64
6/19/1997 10573   $ 84.84
9/22/1997 10677   $ 4.03
9/25/1997 10682   $ 36.13
1/28/1998 10856   $ 58.43
AROUT   Around the Horn
OrderDate Order #   Freight
11/15/1996 10355   $ 41.95

(General) ShapeDemo
End With
' define the SHAPE command for
' the shaped recordset
shpCmd = "SHAPE " &
"(SELECT CustomerID AS [Cust Id], " &
" CompanyName AS Company FROM Customers)" & -
" APPEND ({SELECT CustomerID, OrderDate," & -
" OrderId, Freight FROM Orders})" & -
" AS custOrders" &
" RELATE [Cust Id] TO CustomerID"
' create and open the parent recordset
' using the open connection
Set rst = New ADODB.Recordset
rst.Open shpCmd, conn
' output data from the parent recordset
Do While Not rst.EOF
    Debug.Print rst("Cust Id"); -
    Tab: rst("Company")
    rstChapter = rst("custOrders")
    ' write out column headings
    ' for the child recordset
    Debug.Print Tab;
    "Order Date", "Order #", "Freight"
    ' output data from the child recordset
    Do While Not rstChapter.EOF
        Debug.Print Tab;
        rstChapter("Order Date"), -
        rstChapter("Order ID"),
        Format(rstChapter("Freight"), "$ ##.##")
        rstChapter.MoveNext
    Loop
    rst.MoveNext
Loop

```

**FIGURE 12.17** After running the ShapeDemo procedure in Hands-On 12.66, you can see the contents of the hierarchical recordset in the Immediate window.

### SIDE BAR How to Determine If a Recordset Contains a Field Pointing to Another Recordset

To find out if a certain recordset contains another recordset, you can use the following conditional statement:

```

Dim rst as New ADODB.Recordset
If rst.Fields("custOrders").Type = adChapter then
    Debug.Print "This is a child recordset"
End If

```

#### NOTE

*custOrders is the chapter column alias you created with the AS clause while appending a child recordset to the parent.*

## Writing a Complex **SHAPE** Statement

---

In the previous section, you worked with a simple **SHAPE** statement that displayed order information for each customer in the Northwind\_Chap12.mdb database in the Immediate window. You learned how to nest a child recordset within a parent recordset and access the fields in both. In the following sections, you will learn how to write more complex **SHAPE** statements that include multiple child and grandchild recordsets.

### *Shaped Recordsets with Multiple Children*

---

Data shaping does not limit you to having just one child recordset within a parent recordset. You can specify as many children as you want. For example, to display a parent with two children, use the following syntax:

```
SHAPE {SELECT * FROM Parent}
APPEND ({SELECT * FROM Child1}
RELATE parent-column TO child1-column) AS child1Alias,
({SELECT * FROM Child2}
RELATE parent-column TO child2-column) AS child2Alias
```

Notice that additional children (*siblings*) are added to the end of the **APPEND** clause.

Suppose you want to display both the orders and products for a customer in the Northwind database. Using the syntax provided earlier, you can shape your hierarchical recordset as demonstrated in the **ShapeMultiChildren** procedure shown in Hands-On 12.67.



### **Hands-On 12.67 Creating a Shaped Recordset with Multiple Children**

1. In the Visual Basic Editor window of the Chap12.accdb database, choose **Insert | Module**.
2. In the module's Code window, enter the **ShapeMultiChildren** procedure shown here:

```
Sub ShapeMultiChildren()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim rstChapter1 As Variant
    Dim rstChapter2 As Variant
    Dim strConn As String
    Dim shpCmd As String
    Dim strParent As String
    Dim strChild1 As String
```

```
Dim strChild2 As String
Dim strLink As String
Dim str1stChildName As String
Dim str2ndChildName As String

' define database connection string
' using the OLE DB provider
' and Northwind database as Data Source
strConn = _
    "Data Provider=Microsoft.ACE.OLEDB.12.0;" &_
strConn = strConn & "Data Source = " & _
    "C:\VBAAccess2021_ByExample\Northwind_Chap12.mdb"

' specify Data Shaping provider
' and open connection to the database
Set conn = New ADODB.Connection
With conn
    .ConnectionString = strConn
    .Provider = "MSDataShape"
    .Open
End With

' define the SHAPE command for
' the shaped recordset

strParent = "SELECT CustomerID AS [Cust Id], " & _
    "CompanyName AS Company FROM Customers"

strChild1 = "SELECT CustomerID, OrderDate," & _
    "OrderID, Freight FROM Orders"

strChild2 = "SELECT Customers.CustomerID," & _
    "Products.ProductName FROM Products " & _
    "INNER JOIN ((Customers INNER JOIN Orders ON " & _
    "Customers.CustomerID = Orders.CustomerID) " & _
    "INNER JOIN [Order Details] ON " & _
    "Orders.OrderID = [Order Details].OrderID) ON " & _
    "Products.ProductID = [Order Details].ProductID " & _
    "Order By Products.ProductName"

str1stChildName = "custOrders"
str2ndChildName = "custProducts"

strLink = "RELATE [Cust Id] TO CustomerID"

shpCmd = "SHAPE {"
```

```
shpCmd = shpCmd & strParent
shpCmd = shpCmd & "}"
shpCmd = shpCmd & " APPEND ({"
shpCmd = shpCmd & strChild1
shpCmd = shpCmd & "}"
shpCmd = shpCmd & strLink
shpCmd = shpCmd & ")"
shpCmd = shpCmd & " AS " & str1stChildName
shpCmd = shpCmd & ", ({"
shpCmd = shpCmd & strChild2
shpCmd = shpCmd & "}" "
shpCmd = shpCmd & strLink
shpCmd = shpCmd & ")"
shpCmd = shpCmd & " AS " & str2ndChildName

' create and open the parent recordset
' using the open connection
Set rst = New ADODB.Recordset
rst.Open shpCmd, conn

' output data from the parent recordset
Do While Not rst.EOF
    Debug.Print rst("Cust Id"); Tab; rst("Company")
    rstChapter1 = rst("custOrders")

        ' write out column headings
        ' for the 1st child recordset
    Debug.Print Tab(4); " (" & rst("Cust Id") & _
        " Orders)"
    Debug.Print Tab; "OrderDate", "Order #", "Freight"

        ' output data from the 1st child recordset
    Do While Not rstChapter1.EOF
        Debug.Print Tab; _
            rstChapter1("OrderDate"), _
            rstChapter1("OrderID"), _
            Format(rstChapter1("Freight"), "$ #,##0.00")
        rstChapter1.MoveNext
    Loop

    rstChapter2 = rst("custProducts")
    ' write out column headings
    ' for the 2nd child recordset
    Debug.Print Tab(4); " (" & rst("Cust Id") & _
        " Products)"
```

```

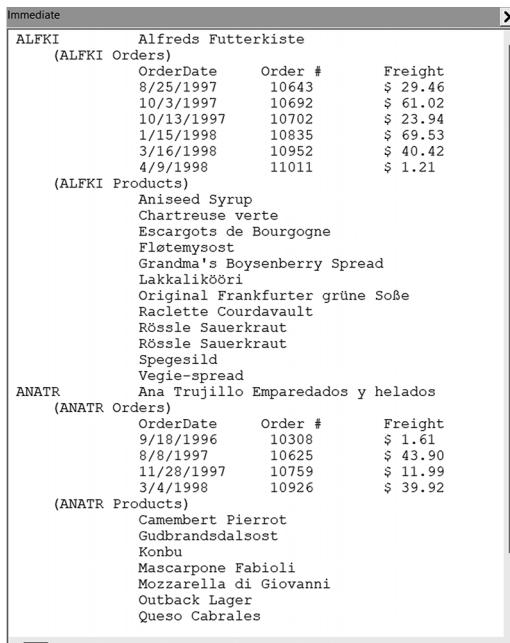
' output data from the 2nd child recordset
Do While Not rstChapter2.EOF
    Debug.Print Tab; _
    rstChapter2("ProductName")
    rstChapter2.MoveNext
Loop
rst.MoveNext
Loop

' Cleanup
rst.Close
Set rst = Nothing
Set conn = Nothing
End Sub

```

**3. Choose Run | Run Sub/UserForm** to execute the procedure.

The `SHAPE` statement in this procedure has been specially formatted so that you can easily create any shaped recordset containing multiple children by replacing `SELECT` statements with your own. This procedure produces the output in the Immediate window as shown in Figure 12.18. Notice that each customer has two child records: Orders and Products.



The Immediate window displays the output of the `ShapeMultiChildren` procedure. It shows two main sections: one for customer ALFKI and one for customer ANATR. Each section contains a header, a list of orders, and a list of products.

ALFKI      Alfers Futterkiste			
(ALFKI Orders)			
OrderDate	Order #	Freight	
8/25/1997	10643	\$ 29.46	
10/3/1997	10692	\$ 61.02	
10/13/1997	10702	\$ 23.94	
1/15/1998	10835	\$ 69.53	
3/16/1998	10952	\$ 40.42	
4/9/1998	11011	\$ 1.21	
(ALFKI Products)			
Aniseed Syrup			
Chartreuse verte			
Escargots de Bourgogne			
Fl��temysost			
Grandma's Boysenberry Spread			
Lakkalik��ri			
Original Frankfurter gr��ne So��e			
Raclette Courdavault			
R��ssle Sauerkraut			
R��ssle Sauerkraut			
Spicesild			
Vegie-spread			
ANATR      Ana Trujillo Emparedados y helados			
(ANATR Orders)			
OrderDate	Order #	Freight	
9/18/1996	10308	\$ 1.61	
8/8/1997	10625	\$ 43.90	
11/28/1997	10759	\$ 11.99	
3/4/1998	10926	\$ 39.92	
(ANATR Products)			
Camembert Pierrot			
Gudbrandsdalsost			
Konbu			
Mascarpone Fabioli			
Mozzarella di Giovanni			
Outback Lager			
Queso Cabrales			

**FIGURE 12.18** The `ShapeMultiChildren` procedure in Hands-On 12.67, generates the following output of the hierarchical recordset with multiple children in the Immediate window.

### ***Shaped Recordsets with Grandchildren***

In addition to the parent recordset having multiple children, the child recordset can contain a child of its own. Simply put, your hierarchical recordset can contain grandchildren. Creating such a hierarchy is a bit harder, but it can be tackled in no time if you take a step-by-step approach. The **SHAPE** syntax that includes grandchildren looks like this:

```
SHAPE {SELECT * FROM Parent}
APPEND ((SHAPE {SELECT * FROM Child}
APPEND ((SELECT * FROM Grandchild}
RELATE child-column TO grandchild-column) AS grandchild-alias)
RELATE parent-column TO child-column) as child-alias
```

Notice that when grandchildren are present, the child recordset is appended with another **SHAPE** command.

Although you can have as many children or grandchildren as you want, it will be more difficult to write a **SHAPE** statement that uses more than three or four levels.

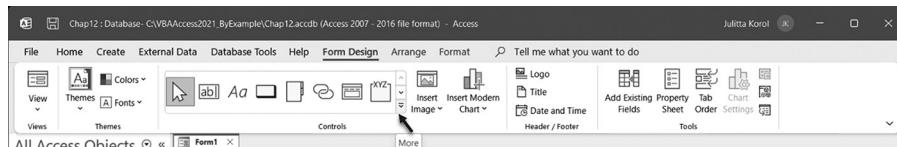
In Custom Project 12.4, you create a shaped recordset that contains both children and grandchildren. Next, you display this recordset on the Access form in the ActiveX TreeView control (see Figure 12.19 for the final output). This project will also introduce you to using aggregate functions within your shaped recordsets.



### **Custom Project 12.4 Using Hierarchical Recordsets**

#### ***Part 1: Creating a Form with a TreeView Control***

1. In the Access window of the Chap12.accdb database, choose **Create | Form Design**. The Form design window opens.
2. In the Controls area of the Form Design tab, click the More button in the Scroll area, and choose **ActiveX Controls** (see Figures 12.19 and 12.20).



**FIGURE 12.19** Adding an ActiveX control to an Access form (Step 1).

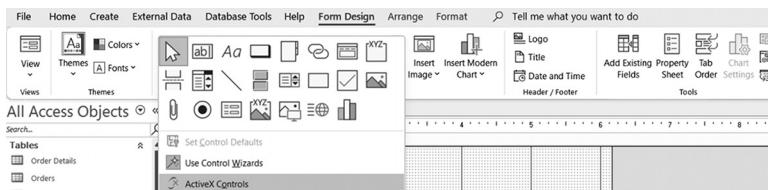


FIGURE 12.20 Adding an ActiveX control to an Access form (Step 2).

3. In the Insert ActiveX Control window, choose **Microsoft TreeView Control, version 6.0** as shown in Figure 12.21, and click **OK** to place a TreeView control on the form.

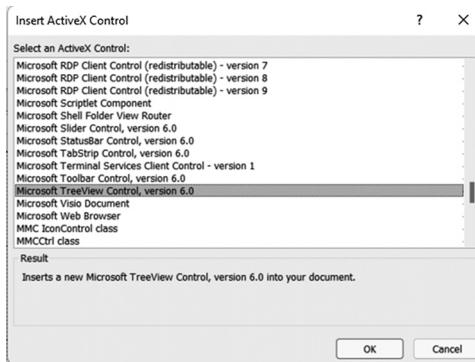


FIGURE 12.21 The Microsoft TreeView control provides an excellent way to display shaped recordsets in an Access form.

4. Resize the TreeView control and the form to match Figure 12.22.
5. Click the TreeView control to select it. In the property sheet, change the Name property of the TreeView control from TreeView0 to **myTreeCtrl**.

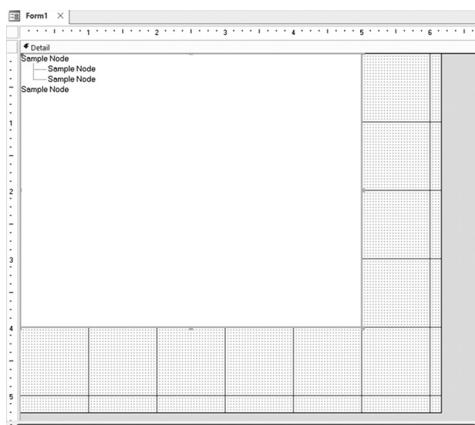


FIGURE 12.22 A TreeView control after being placed and resized on the Access form.

6. You can use the Property Sheet to adjust various properties of the Tree View Control (Figure 12.23).

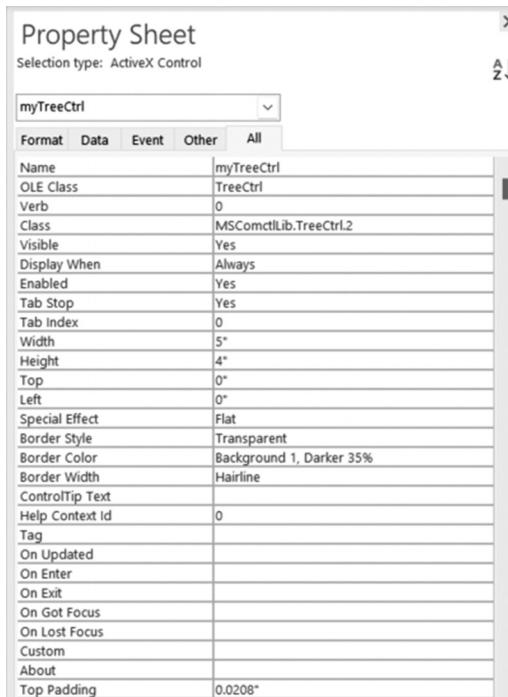


FIGURE 12.23 You can set custom properties of the TreeView control in the Property Sheet of TreeView control.

7. Save the form as **frmOrders**.

### ***Part 2: Writing an Event Procedure for the Form Load Event***

---

1. In the property sheet, select **Form** from the drop-down box and click the **Event** tab for the selected form.
2. Click the **Build** button (...) next to the **On Load** event name to display the Choose builder dialog box.
3. In the Choose Builder dialog box, select **Code Builder** and click **OK**. The form module window appears with the following Form\_Load event procedure stub:

```
Private Sub Form_Load()
End Sub
```

4. Type the code for the **Form\_Load** event procedure shown here, or copy the procedure code from **Chap12.txt** in the companion files:

```
Private Sub Form_Load()
    Dim conn As ADODB.Connection
    Dim rstCustomers As ADODB.Recordset
    Dim rstOrders As ADODB.Recordset
    Dim rstOrderDetails As ADODB.Recordset
    Dim fld As Field
    Dim objNode1 As Node
    Dim objNode2 As Node
    Dim strConn As String
    Dim strSQL As String

    Dim strSQLCustomers As String
    Dim strSQLOrders As String
    Dim strSQLOrderDetails As String
    Dim strSQLRelParentToChild As String
    Dim strSQLRelGParentToParent As String

    ' Create the ADO Connection object
    Set conn = New ADODB.Connection

    ' Specify a valid connection string
    strConn = "Data Provider=Microsoft.ACE.OLEDB.12.0;" &
    strConn = strConn & "Data Source = " &
    "C:\VBAAccess2021_ByExample\Northwind_Chap12.mdb"
    conn.ConnectionString = strConn

    ' Specify the Data Shaping provider
    conn.Provider = "MSDataShape"

    ' Open the connection
    conn.Open

    ' Specify SELECT statement for the Grandparent
    strSQLCustomers = "SELECT CustomerID " & _
        "AS [Cust #]," & _
        "CompanyName AS [Customer] " & _
        "FROM Customers"
    ' Specify SELECT statement for the Parent
    strSQLOrders = "SELECT OrderID AS " & _
        "[Order #]," & _
        "OrderDate AS [Order Date]," & _
        "Orders.CustomerID AS [Cust #]" & _
        "FROM Orders ORDER BY OrderDate DESC"
```

```
' Specify SELECT statement for the Child
strSQLOrderDetails =
    "SELECT od.OrderID AS [Order #]," & _
    "p.CategoryId AS [Category]," & _
    "p.ProductName AS [Product]," & _
    "od.Quantity," & _
    "od.ProductId," & _
    "od.UnitPrice AS [Unit Price]," & _
    "(od.UnitPrice * od.Quantity) " & _
    "AS [Extended Price] " & _
    "FROM [Order Details] od " & _
    "INNER JOIN Products p " & _
    "ON od.ProductID = p.ProductID " & _
    "ORDER BY p.CategoryId, p.ProductName"

' Specify RELATE clause to link Parent to Child
strSQLRelParentToChild =
    "RELATE [Order #] TO [Order #]"

' Specify RELATE clause to link Grandparent
' to Parent
strSQLRelGParentToParent =
    "RELATE [Cust #] TO [Cust #]"

' Build complete SQL statement for the
' shaped recordset adding aggregate
' functions for the Grandparent and Parent
strSQL = "SHAPE(SHAPE{" & strSQLCustomers & "})"
strSQL = strSQL &
    "APPEND((SHAPE{" & strSQLOrders & "}) "
strSQL = strSQL &
    "APPEND({" & strSQLOrderDetails & "}) "
strSQL = strSQL &
    strSQLRelParentToChild & ") AS rstOrderDetails,"
strSQL = strSQL &
    "COUNT(rstOrderDetails.Product) "
strSQL = strSQL &
    "           AS [Items On Order],"
strSQL = strSQL &
    "SUM(rstOrderDetails.[Extended Price]) "
strSQL = strSQL &
    "           AS [Order Total])"
strSQL = strSQL &
    strSQLRelGParentToParent & ") AS [rstOrders],"
strSQL = strSQL &
```

```
"SUM(rstOrders.[Order Total]) "
strSQL = strSQL & _
"           AS [Cust Grand Total]"
strSQL = strSQL & ") AS rstCustomers"

' Create and open the Grandparent recordset
Set rstCustomers = New ADODB.Recordset
rstCustomers.Open strSQL, conn

' Fill the TreeView control
Do While Not rstCustomers.EOF
    Set objNode1 = myTreeCtrl.Nodes.Add _
        (Text:=rstCustomers.Fields(0) & _
        "   " & rstCustomers.Fields(1) & _
        "   ($ " & rstCustomers.Fields(3) & ")")
    Set rstOrders = _
        rstCustomers.Fields("rstOrders").Value
    Do While Not rstOrders.EOF
        Set objNode2 = myTreeCtrl.Nodes.Add _
            (relative:=objNode1.Index, _
            relationship:=tvwChild, _
            Text:=rstOrders.Fields(0) & _
            "   " & rstOrders.Fields(1) & _
            "   " & rstOrders.Fields(4) & " (items)" & _
            "   $" & rstOrders.Fields(5) & _
            " (Order Total)")
        Set rstOrderDetails = _
            rstOrders.Fields("rstOrderDetails").Value
        Do While Not rstOrderDetails.EOF
            myTreeCtrl.Nodes.Add _
                relative:=objNode2.Index, _
                relationship:=tvwChild, _
                Text:=rstOrderDetails.Fields(3) & _
                "   " & rstOrderDetails.Fields(2) & _
                "   $" & rstOrderDetails.Fields(6) & _
                "   (" & rstOrderDetails.Fields(3) & _
                " x $" & rstOrderDetails.Fields(5) & ")"
            rstOrderDetails.MoveNext
        Loop
        rstOrders.MoveNext
    Loop
    rstCustomers.MoveNext
Loop

' Cleanup
```

```

rstCustomers.Close
Set rstCustomers = Nothing
Set conn = Nothing
End Sub

```

5. Choose Tools | References and set the reference to the **Microsoft Windows Common Controls 6.0 (SP6)**. If this reference is not listed in the Available References list box, click the **Browse** button. Switch to the **Windows\System 32** or **Windows\SysWOW64** folder and in the Reference window, select **ActiveX Controls (\*.ocx)** in the files of type drop-down box, scroll down to locate and select **MSCOMCTL.OCX**. Click the **Open** button to confirm your selection, and then click **OK** to exit the References window.
6. Press Ctr+F11 to return to the Access application window and open **frmOrders** in Form view.

When you open the frmOrders form, the Form\_Load procedure populates the TreeView control with the data from the Northwind\_Chap12.mdb database. As you can see in Figure 12.24, the results are quite impressive. Double-clicking on the nodes in the TreeView control expands and collapses the details underneath those nodes.

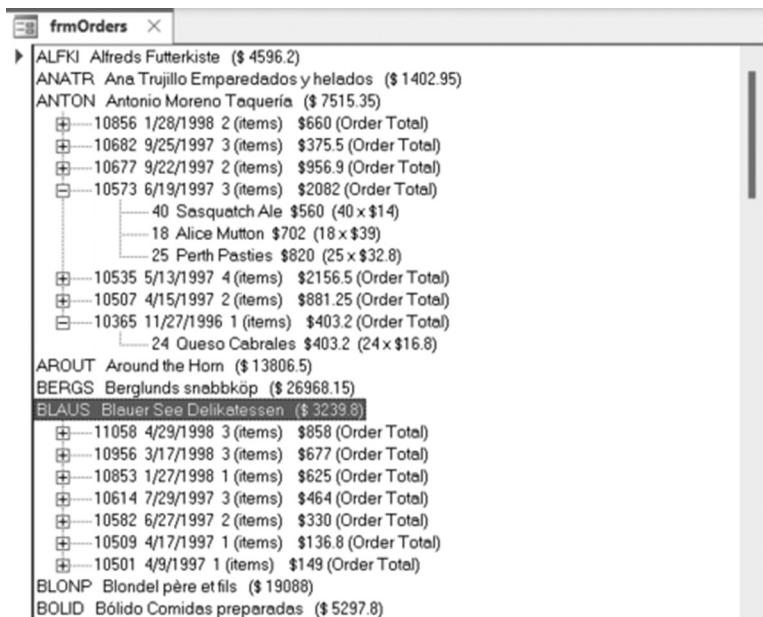


FIGURE 12.24 The TreeView control is filled with the data from the Northwind database when the user opens the form.

Prior to populating the TreeView control with the data, we connect to the database and enlist the services of the Data Shaping provider:

```
conn.Provider = «MSDataShape»
```

Because a TreeView control displays data as a hierarchy, we need to build a complex SQL statement using the `SHAPE` syntax we learned in preceding sections. To make things easier for ourselves, we start by defining SQL statements with fields we want to display for parent, child, and grandchild recordsets. Notice that we renamed some fields using the `AS` clause. We also defined separate statements to allow us to link grandparent to parent and parent to child. The structure we need to create can be illustrated like this:

```
Grandparent  
Parent  
Child
```

Now that we've defined the relationship and the fields for our data hierarchy, we use the `SHAPE` commands to build the complete `SHAPE` statement:

```
strSQL = "SHAPE(SHAPE{ " & strSQLCustomers & " }"  
strSQL = strSQL &  
    "APPEND((SHAPE{ " & strSQLOrders & " } "  
strSQL = strSQL &  
    "APPEND({ " & strSQLOrderDetails & " } "  
strSQL = strSQL &  
    strSQLRelParentToChild & ") AS rstOrderDetails,"  
strSQL = strSQL &  
    "COUNT(rstOrderDetails.Product) "  
strSQL = strSQL &  
    "           AS [Items On Order],"  
strSQL = strSQL &  
    "SUM(rstOrderDetails.[Extended Price]) "  
strSQL = strSQL &  
    "           AS [Order Total])"  
strSQL = strSQL &  
    strSQLRelGParentToParent & ") AS [rstOrders],"  
strSQL = strSQL &  
    "SUM(rstOrders.[Order Total]) "  
strSQL = strSQL &  
    "           AS [Cust Grand Total]"  
strSQL = strSQL & ") AS rstCustomers"
```

While creating the `SHAPE` statement, we added additional calculated fields using the aggregate functions. For instance, in the parent recordset (`rstOrders`) we calculated the number of items ordered using the `COUNT` function:

```
COUNT(rstOrderDetails.Product) AS [Items On Order]
```

We also used the `SUM` function to obtain the total amount of the order:

```
SUM(rstOrderDetails.[Extended Price]) AS [Order Total]
```

In the grandparent recordset (`rstCustomers`), we used the `SUM` function to calculate the total amount owed by a customer.

When expanded, the complete `SHAPE` statement will look as follows:

```
strSQL = "SHAPE(SHAPE{"
strSQL = strSQL & "SELECT CustomerID AS [Cust #],"
strSQL = strSQL & "CompanyName AS [Customer]"
strSQL = strSQL & "FROM Customers"
strSQL = strSQL & "})"
strSQL = strSQL & "APPEND((SHAPE{"
strSQL = strSQL & "SELECT OrderID AS [Order #],"
strSQL = strSQL & "OrderDate AS [Order Date],"
strSQL = strSQL & "Orders.CustomerID AS [Cust #]"
strSQL = strSQL & "FROM Orders"
strSQL = strSQL & "ORDER BY OrderDate DESC"
strSQL = strSQL & "})"
strSQL = strSQL & "APPEND({"
strSQL = strSQL & "SELECT od.OrderID AS [Order #],"
strSQL = strSQL & "p.CategoryId AS [Category],"
strSQL = strSQL & "p.ProductName AS [Product],"
strSQL = strSQL & "od.Quantity,"
strSQL = strSQL & "od.ProductID,"
strSQL = strSQL & "od.UnitPrice AS [Unit Price],"
strSQL = strSQL & "(od.UnitPrice * od.Quantity) "
strSQL = strSQL & "AS [Extended Price] "
strSQL = strSQL & "FROM [Order Details] od INNER JOIN Products p"
strSQL = strSQL & " ON od.ProductID = p.ProductID "
strSQL = strSQL & "ORDER BY p.CategoryId, p.ProductName"
strSQL = strSQL & "})"
strSQL = strSQL & "RELATE [Order #] TO [Order #]"
strSQL = strSQL & ")"
strSQL = strSQL & "AS rstOrderDetails,"
strSQL = strSQL & "COUNT(rstOrderDetails.Product) "
strSQL = strSQL & "AS [Items On Order],"
strSQL = strSQL & "SUM(rstOrderDetails.[Extended Price]) "
strSQL = strSQL & "AS [Order Total])"
strSQL = strSQL & "RELATE [Cust #] TO [Cust #]"
```

```

strSQL = strSQL & ")"
strSQL = strSQL & "AS [rstOrders],"
strSQL = strSQL & "SUM(rstOrders.[Order Total]) "
strSQL = strSQL & "AS [Cust Grand Total]) AS rstCustomers"

```

Notice that the `SHAPE` statement we built contains standard fields pulled from the database tables and child recordsets (`rstOrders`, `rstOrderDetails`), as well as calculated columns. The `rstOrders` recordset is a field in the `rstCustomers` recordset. This field contains order information for a customer. `rstOrderDetails` is a field within the `rstOrders` recordset. This field contains the order details information for a customer's order.

Now that we've completed the `SHAPE` statement, we can open the grandparent recordset and begin populating the TreeView control with our data.

A TreeView control consists of Node objects, which you can expand or collapse to display or hide child nodes. Nodes that have child nodes are referred to as *parent* nodes. The nodes located at the top of the tree control are referred to as *root* nodes. Root nodes can have *sibling* nodes that are located on the same level. For example, customer ALFKI (see Figure 12.24) is a root node, and so is the customer ANATR, ANTON, and so on. They are also siblings of one another.

To populate a TreeView control, we use the `Add` method of the `Nodes` collection like this:

```
Set objNode1 = myTreeCtrl.Nodes.Add
```

`objNode1` is an object variable representing the `Node` object. The first node added to a TreeView is a root node. The `Add` method of the `Nodes` collection uses the following syntax:

```
object.Add([relative,] [relationship,] [key], text[, image,]
[selectedimage])
```

The only required arguments in the syntax are `object` and `text`. The `object` is the object variable (`myTreeCtrl`) representing the TreeView control. The `text` is a string that appears in the node. The following complete statement:

```

Set objNode1 = myTreeCtrl.Nodes.Add _
(Text:=rstCustomers.Fields(0) & _
" " & rstCustomers.Fields(1) & _
" (" & rstCustomers.Fields(3) & ")")

```

creates a root node to display the following information:

```
Cust # (rstCustomers.Fields(0))
Customer (rstCustomers.Fields(1))
Cust Grand Total (rstCustomers.Fields(3))
```

Because the preceding statement appears inside a looping structure, the TreeView control will display all the customers at their root level.

Now that we've taken care of the root node, we go on to add children and grandchildren. A child node has a relationship to a parent node that has already been added. To define a child node, in addition to the required `text` argument, we will use two optional arguments of the `Add` method as follows:

- `relative`—This is the index number or key of a preexisting Node object. In our example, we used the index of the parent node that we just created (`relative:=objNode1.Index`).

**NOTE**

*When a Node object is created, it is automatically assigned an index number. This number is stored in the Node object's Index property.*

- `relationship`—Specifies the type of relationship you are creating. Use the `tvwChild` setting to create a child node of the node named in the `relative` argument. The statement that creates a child node looks like this:

```
Set objNode2 = myTreeCtrl.Nodes.Add _  
    (relative:=objNode1.Index, _  
     relationship:=tvwChild, _  
     Text:=rstOrders.Fields(0) & _  
     " " & rstOrders.Fields(1) & _  
     " " & rstOrders.Fields(4) & "(items)" & _  
     $" & rstOrders.Fields(5) & _  
     "(Order Total)")
```

The preceding statement displays order information for a customer. The child node `text` argument is set to display:

```
Order # (rstOrders.Fields(0))  
Order Date (rstOrders.Fields(1))  
Items On Order (rstOrders.Fields(4))  
Order Total (rstOrders.Fields(5))
```

Because this statement appears inside a looping structure, the TreeView control will display the order information for each customer. Finally, we add grandchildren using the following statement:

```
myTreeCtrl.Nodes.Add _  
    relative:=objNode2.Index, _  
    relationship:=tvwChild, _  
    Text:=rstOrderDetails.Fields(3) & _
```

```
" " & rstOrderDetails.Fields(2) & _  
" $" & rstOrderDetails.Fields(6) & _  
" (" & rstOrderDetails.Fields(3) & _  
" x $" & rstOrderDetails.Fields(5) & ") "
```

This statement displays order details for a customer's order. Notice that this Node object references the index number of the child object that has just been added (`relative:=objNode2.Index`).

The grandchild node `text` argument is set to display:

```
Quantity (rstOrderDetails.Fields(3))  
Product (rstOrderDetails.Fields(2))  
Extended Price (rstOrderDetails.Fields(6))  
Quantity x Unit Price (rstOrderDetails.Fields(3) & "x $" &  
rstOrderDetails.Fields(5))
```

The looping structure ensures that these order details are listed for all customers' orders.

Now that you are done with this custom project, you should be able to provide your own hierarchical data in a pretty neat user interface.

## TRANSACTION PROCESSING

---

To improve your application's performance and to ensure that database activities can be recovered in case an unexpected hardware or software error occurs, consider grouping sets of database activities into a transaction. A *transaction* is a set of operations that are performed together as a single unit. If you use an automatic teller machine (ATM), you are already familiar with transaction processing. When you go to the bank to get cash, your account must be debited. In other words, the cash withdrawal must be deducted from your savings or checking account. A transaction is a two-sided operation. If anything goes wrong during the transaction, the entire transaction is canceled. If both operations succeed, that is, you get the cash and the bank debits your account, the transaction's work is saved (or committed).

Database transactions often involve modifications and additions of one or more records in a single table or in several tables. When a transaction has to be undone or canceled, the transaction is rolled back. Often, when you perform batch updates to database tables and an error occurs, updates to all tables must be canceled or the database could be left in an inconsistent state, resulting not only in loss of important information but also in a number of other headaches.

Transactions are extremely important for maintaining data integrity and consistency. In ADO, the Connection object offers three methods (`BeginTrans`, `CommitTrans`, and `RollbackTrans`) for managing transaction processing. You should use these methods to save or cancel a series of changes made to the data as a single unit.

- `BeginTrans`—Begins a new transaction
- `CommitTrans`—Saves any changes and ends the current transaction
- `RollbackTrans`—Cancels any changes made during the current transaction and ends the transaction

Please note that in ADO a transaction is limited to one database because the Connection object can only point to one database.

To work with transaction processing in DAO, use the transaction methods of the Workspace or DBEngine object: `BeginTrans`, `CommitTrans`, and `Rollback`. Within a Workspace transaction you can perform operations on more than one connection or database.

### **Creating a Transaction**

---

Use the `BeginTrans` method to specify the beginning of a transaction and the `CommitTrans` method to save the changes. `BeginTrans` and `CommitTrans` are used in pairs. The data-modifying instructions you place between these keywords are stored in memory until VBA encounters the `CommitTrans` statement. After reaching `CommitTrans`, Access writes to the disk the changes that have occurred since the `BeginTrans` statement; therefore, any changes you've made in the tables become permanent.

If an error is generated during the transaction process, the `RollbackTrans` statement placed further down in your procedure will undo all changes made since the `BeginTrans` statement. The rollback ensures that the data is returned to the state it was in before you started the transaction.

Using transaction processing helps improve database performance because the operations carried out during a transaction are run in memory. If the transaction succeeds, the results are written to the disk in a single operation. If any operation included in a transaction fails, the transaction is simply aborted and no changes are written to the database. If you don't use transactions, the results of each operation must be written to the disk separately—a process that consumes more database resources.

The procedure in Hands-On 12.68 assumes that you want to enter an order for a new customer. Because this customer does not exist in the database, you

will use a transaction to ensure that the new order is entered only after the customer record has been created in the Customers table. The result is shown in Figure 12.25.



### Hands-On 12.68 Using a Database Transaction to Insert Records

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, enter the **Create\_Transaction\_ADO** procedure as shown here:

```
Sub Create_Transaction_ADO()
    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler

    Set conn = New ADODB.Connection

    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .ConnectionString = "Data Source = " & _
            "C:\VBAAccess2021_ByExample\Northwind_Chap12.mdb"
        .Open
        .BeginTrans

        ' insert a new customer record
        .Execute "INSERT INTO Customers " & _
        "Values ('GWIPO', 'Gwiazda Polarna', " & _
        "'Marcin Garnia', 'Sales Manager', " & _
        "'ul.Majewskiego 10', 'Warszawa', Null, " & _
        "'02-106', 'Poland', '0114822230445', Null)"

        ' insert the order for that customer
        .Execute "INSERT INTO Orders " & _
        " (CustomerId, EmployeeId, " & _
        " OrderDate, RequiredDate) " & _
        " Values ('GWIPO', 1, Date(), Date() +5)"
        .CommitTrans
        .Close
        MsgBox "Both inserts completed."
    End With

    ExitHere:
    Set conn = Nothing
    Exit Sub
ErrorHandler:
```

```

If Err.Number = -2147467259 Then
    MsgBox Err.Description
    Resume ExitHere
Else
    MsgBox Err.Description
    With conn
        .RollbackTrans
        .Close
    End With
    Resume ExitHere
End If
End Sub

```

### 3. Choose Run | Run Sub/UserForm to execute the procedure.

The first SQL `INSERT INTO` statement inserts the customer data into the Customers table in the Northwind\_Chap12.mdb database. Before the customer can actually order specific products, a record must be added to the Orders table. The second SQL `INSERT INTO` statement takes care of this task. Because both inserts must occur prior to filling in order details, they are treated as a single transaction. If an error occurs anywhere (for example, the Orders table is open in Design view), the entire transaction is rolled back. Notice how the `INSERT INTO` statement is used in this procedure. If you do not specify the field names, you will need to include values for each field in the table.

Customer ID	Company Name	Contact Name	Contact Title	Address
GOURL	Gourmet Lanchonetes	André Fonseca	Sales Associate	Av. Brasil, 442
GREAL	Great Lakes Food Market	Howard Snyder	Marketing Manager	2732 Baker Blvd.
GROSER	GROSELLA-Restaurante	Manuel Pereira	Owner	5ª Ave. Los Palos Grandes
GWipo	Gwiazda Polarna	Marcin Gartia	Sales Manager	ul.Majewskiego 10
*	Order ID	Employee	Order Date	Required Date
*	11078	Davolio, Nancy	26-Nov-2021	01-Dec-2021
*	(New)			
				Freight
				\$0.00
				\$0.00

Customer ID	Company Name	Contact Name	Contact Title	Address
HANAR	Hanari Carnes	Mario Pontes	Accounting Manager	Rua do Paço, 67
HILAA	HILARIÓN-Abastos	Carlos Hernández	Sales Representative	Carrera 22 con Ave. Carlos Soublette
HUNG	Hungry Coyote Import Store	Yoshi Latimer	Sales Representative	City Center Plaza
HUNGO	Hungry Owl All-Night Grocers	Patricia McKenna	Sales Associate	8 Johnstown Road
ISLAT	Island Trading	Helen Bennett	Marketing Manager	Garden House

**FIGURE 12.25** After running the procedure in Hands-On 12.68, a record for a new customer, “GWIPO,” is added to the Customers and Orders tables.

## EXAMINING THE REFERENCES COLLECTION

As you have seen in this chapter your programming code may require the presence of certain libraries that provide the required objects, properties, and methods. Access provides you with the `References` collection that you can

use to examine which references are set for the current application. The following ShowProjReferences procedure iterates through the references set in the Chap12.accdb database and prints the reference name and its path to the Immediate window.

```
Sub ShowProjReferences()

    Dim intRef As Integer

    For intRef = 1 To References.Count
        Debug.Print "Reference Name:" & _
            References(intRef).Name & _
            "(" & References(intRef).FullPath & ")"
    Next
End Sub
```

After running the above procedure, the following list of references is pulled from the current VBA project:

```
Reference Name:VBA(C:\Program Files\Common Files\Microsoft Shared\VBA\VBA7.1\VBE7.DLL)
Reference Name:Access(C:\Program Files\Microsoft Office\root\Office16\MSACC.OLB)
Reference Name:stdole(C:\Windows\System32\stdole2.tlb)
Reference Name:DAO(C:\Program Files\Common Files\Microsoft Shared\OFFICE16\ACEDAO.DLL)
Reference Name:ADOX(C:\Program Files\Common Files\System\ado\msadox.dll)
Reference Name:ADODB(C:\Program Files\Common Files\System\ado\msado15.dll)
Reference Name:Word(C:\Program Files\Microsoft Office\root\Office16\MSWORD.OLB)
Reference Name:MSComctlLib(C:\WINDOWS\system32\MSCOMCTL.OCX)
```

The References collection also provides the `BuiltIn` property that returns a Boolean value (True / False) indicating whether a Reference object points to a default reference that's necessary for Microsoft Access to function correctly. Let's look at another procedure that does that using the for each loop:

```
Sub IsBuiltInReference()
    Dim ref As Reference
```

```
For Each ref In References
    If ref.BuiltIn = True Then
        Debug.Print ref.Name
    End If
Next ref
End Sub
```

## SUMMARY

---

This marathon chapter covered quite a bit of simple and advanced ADO material you will find useful in developing professional applications in Access. You started by creating your own recordset from scratch and using it for storing non-relational data. Next, you learned how to disconnect a recordset from a database and work with it offline. You also learned that a recordset can be saved to a disk file and later reopened without an active connection to the database. Next, you discovered how you can use the `Clone` method of the Recordset object to create a recordset that is a copy of another recordset. Finally, you familiarized yourself with the concepts of data shaping and learned statements that make it possible to create impressive hierarchical views of your data. You also learned how transactions are used to ensure that certain database operations are always performed as a single unit.

In the next chapter, you will learn special Data Definition Language commands for creating a new Access database, as well as creating, modifying, and deleting tables. You also learn commands for adding, modifying, and deleting fields and indexes.

Part



# *ACCESS STRUCTURED QUERY LANGUAGE (SQL)*

**D**ata Definition Language (DDL) is a component of Structured Query Language (SQL), which is used for defining database objects (tables, views, stored procedures, primary keys, indexes, and constraints) and managing database security. In this part of the book, you will learn how to use DDL with Jet databases, ADO, and the Jet 4.0/ACE OLE DB Provider.

- Chapter 13** Creating, Modifying, and Deleting Tables and Fields
- Chapter 14** Enforcing Data Integrity and Relationships between Tables
- Chapter 15** Defining Indexes and Primary Keys
- Chapter 16** Views and Stored Procedures



# Chapter 13

## *CREATING, MODIFYING, AND DELETING TABLES AND FIELDS*

In Part II of this book, you tried out different methods that are available in Access for creating and manipulating databases via VBA programming code using the DAO and ADO object models. In particular, you learned how to create new databases from scratch, add tables and indexes, set up relationships between tables, secure a database with a password, define user and group security accounts, and handle object permissions. In addition to using DAO and ADO, you can perform many of the mentioned database tasks by using Data Definition Language (DDL), which is a component of Structured Query Language (SQL).

## INTRODUCTION TO ACCESS SQL

---

SQL is a widely used language for data retrieval and manipulation in databases. The SQL specification (known as ANSI SQL-89) was first published in 1989 by the American National Standards Institute (ANSI). The ANSI SQL standard was revised in 1992; this version is referred to as ANSI SQL-92 or SQL-2. This revised specification is supported by the major database vendors, many of whom have created their own extensions of the SQL language. Access 2021 supports both SQL specifications and refers to them as *ANSI SQL query modes*.

While the ANSI-89 SQL query mode (also called Microsoft Jet SQL and ANSI SQL) uses the traditional Jet SQL syntax, the ANSI-92 SQL mode uses syntax that is more compliant with SQL-92 and Microsoft SQL Server. For example, ANSI-92 uses the percent sign (%) and the underscore character (\_) for its wildcards instead of the asterisk (\*) and the question mark (?), which are commonly used in VBA. Microsoft Access Jet Engine does not implement the complete ANSI SQL-92 standard and provides its own Jet 4.0 ANSI SQL-92 extensions to support new features of Access. You can use the ANSI-92 syntax in your VBA procedures with the Microsoft OLE DB Provider for Jet or with the Data Definition Language, which we cover in this part of the book. ANSI-89 is the default setting for a new Access database in Access 2002–2003 and 2000 file formats. Because the two ANSI SQL query modes are not compatible, you must decide which query mode you are going to use for the current database. This can easily be done in the Access user interface as outlined in Hands-On 13.1.

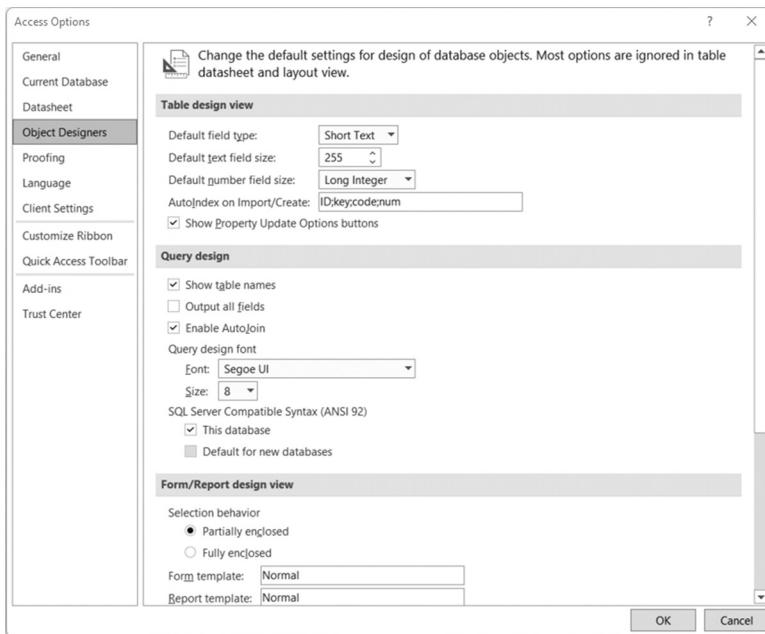
**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



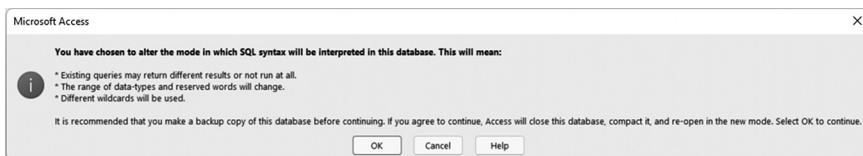
### Hands-On 13.1 Setting the ANSI SQL Query Mode

1. Start Access and create a new database named **Chap13.accdb** in your C:\VBAAccess2021\_ByExample folder.
2. Click the **File** tab and select **Options**.
3. In the left pane of the Access Options window, select **Object Designers**.
4. In the right pane, in the Query design section, look for the SQL Server Compatible Syntax (ANSI 92) area (see Figure 13.1). Set the query mode to ANSI-92 SQL by clicking the **This database** checkbox. When **This database** checkbox is not selected, the query mode is assumed to be ANSI-89 SQL.



**FIGURE 13.1** Use the Access Options window to set the ANSI SQL query mode for the current database or all new databases.

5. Click **OK** to exit the Access Options window.
6. Access displays a message as shown in Figure 13.2.
7. Click **OK** to accept the message.
8. The Access database will close and reopen with the new settings in effect.



**FIGURE 13.2** When you change the query mode to ANSI-92, Microsoft Access displays an informational message alerting you to possible problems.

There are two areas of Access SQL:

- Data Definition Language (DDL) offers several SQL statements to manage database security and to create and alter database components (such as tables, indexes, relationships, views, and stored procedures). These statements are: CREATE TABLE, DROP TABLE, ALTER TABLE, CREATE

INDEX, DROP INDEX, CHECK CONSTRAINT, CREATE VIEW, DROP VIEW, CREATE PROCEDURE, DROP PROCEDURE, EXECUTE, ALTER DATABASE, ADD USER, ALTER USER, CREATE USER, CREATE GROUP, DROP GROUP, DROP USER, GRANT, and REVOKE.

- Data Manipulation Language (DML) offers SQL statements that allow you to retrieve and manipulate data contained in the database tables as well as perform transactions. These statements are: SELECT, UNION, UPDATE, DELETE, INSERT INTO, SELECT INTO, INNER JOIN, LEFT JOIN, RIGHT JOIN, TRANSFORM, PARAMETERS, BEGIN TRANSACTION, COMMIT, and ROLLBACK.

This chapter and the remaining chapters of Part III focus on using the DDL language for creating and changing the underlying structure of a database. To get the most out of these chapters, you should be familiar with using DAO and ADO, discussed in Part II.

## CREATING TABLES

---

Using the Access SQL `CREATE TABLE` statement and the `Execute` method of either the DAO Database object or the ADO Connection object, you can define a new table, its fields, and field constraints. The `CREATE TABLE` statement can only be used with Microsoft Jet and Microsoft Access engine databases. The two examples that follow illustrate how to create a table named `tblSchools` in the currently open database and in a new database using ADO.



### Hands-On 13.2 Creating a Table in the Current Database (DDL with ADO)

1. In the `Chap13.accdb` database that you created in Hands-On 13.1, switch to the Visual Basic Editor window and choose **Tools | References**. In the References dialog box, scroll down to locate **Microsoft ActiveX Data Objects 6.1 Library**. Click the checkbox to the left of this library name to set a reference to it and click **OK** to exit the dialog box.
2. Choose **Insert | Module** to add a new module to the current VBA project.
3. In the module's Code window, type the following **CreateTable** procedure:

```
Sub CreateTable()  
    ' you must set up a reference to  
    ' the Microsoft ActiveX Data Objects Library  
    ' in the References dialog box
```

```
Dim conn As ADODB.Connection
Dim strTable As String
Dim strSQL As String

On Error GoTo ErrorHandler

Set conn = CurrentProject.Connection

strTable = "tblSchools"
strSQL = "CREATE TABLE " & strTable
strSQL = strSQL & "(SchoolID AUTOINCREMENT(100, 5), "
strSQL = strSQL & "SchoolName CHAR, "
strSQL = strSQL & "City CHAR (25), "
strSQL = strSQL & "District CHAR (35), "
strSQL = strSQL & "YearEstablished DATE);"

Debug.Print strSQL

conn.Execute strSQL

Application.RefreshDatabaseWindow
ExitHere:
conn.Close
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub
```

4. Position the insertion point anywhere within the code of the CreateTable procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.

This procedure uses ADO to establish a connection to the current database (Chap13.accdb). The ADO Connection object's `Execute` method is used to execute the Data Definition Language `CREATE TABLE` statement that defines a new table and its fields. The first field is named `SchoolID` and its data type is defined as `AutoNumber`.

The seed and increment values of `AutoNumber` columns are specified using the following syntax:

```
Column_name AUTOINCREMENT (seed, increment)
```

The table, `tblSchools`, has an `AutoNumber` column with a seed of 100 and an increment of 5:

```
SchoolID AUTOINCREMENT(100, 5)
```

When you switch to the database window, open this table in Datasheet view, and proceed to entering records, the `SchoolID` for the first record will be 100, the second will be 105, the third 110, and so on.

Three fields are defined as Text fields and one field as a Date/Time field. The Text fields are defined using the CHAR data type (see Table 13.1). To specify the size of the Text field, put the appropriate value between parentheses. If the size of the Text field is not specified, it is assumed to be 255 characters long.

When you examine the code of the `CreateTable` procedure and compare the resultant table in Figure 13.3, you will notice that Access SQL uses different data types than those available in the Table Design window. See Table 13.1 for the comparison of data types.

Field Name	Data Type	Description (Optional)
SchoolID	AutoNumber	
SchoolName	Short Text	
City	Short Text	
District	Short Text	
YearEstablished	Date/Time	

Field Properties

General    Lookup

Field Size	Long Integer
New Values	Increment
Format	General
Caption	School ID
Indexed	No
Text Align	General

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

FIGURE 13.3. The `tblSchools` table was generated by the `CreateTable` procedure in Hands-On 13.2 using the Microsoft Access SQL statement `CREATE TABLE`.

TABLE 13.1. Table design data types and their Access SQL equivalents.

Table Design Data Types	Access SQL Data Types used to create tables
Text	CHAR or VARCHAR
Memo	LONGTEXT
Number (Field Size = Byte)	BYTE

Table Design Data Types	Access SQL Data Types used to create tables
Number (Field Size = Integer)	SHORT
Number (Field Size = Long Integer)	LONG
Number (Field Size = Single)	SINGLE
Number (Field Size = Double)	DOUBLE
Date/Time	DATETIME
Date/Time Extended (new in Access 2021)	No equivalent Access SQL data type
Currency	CURRENCY or MONEY
AutoNumber (Field Size = Long Integer)	AUTOINCREMENT or COUNTER
AutoNumber (Field Size = Replication Id)	GUID
Yes/No	BIT
OLE Object	LONGBINARY

The `RefreshDatabaseWindow` method of the `Application` object ensures that the database window is updated after the creation of the new table object. The error-handling code will alert you if an error is encountered. Try to run this procedure again in step mode (F8) to see what happens. Notice that the procedure uses two labels to mark appropriate sections in the procedure. The `On Error GoTo ErrorHandler` statement will transfer the procedure execution to the line labeled `ErrorHandler` when an error is triggered. Statements following this label will be executed until the `Resume` statement is encountered. This statement will direct the code execution to the line labeled `ExitHere`. The `Exit Sub` statement in the `ExitHere` block of code will allow us to exit the procedure whether or not an error is encountered.

Sometimes you may be required to create a new database and a new table in one procedure. Hands-On 13.3 demonstrates how to create a table in a brand-new database.



### Hands-On 13.3 Creating a Table in a New Database (DDL with ADO/AOX)

1. In the Visual Basic Editor window, choose **Tools | References**. In the References dialog box, scroll down to locate **Microsoft ADO Ext. 6.0 for DDL and Security Object Library**. Click the checkbox to the left of the library name to set a reference to it. Also, make sure that the Microsoft ActiveX Data Objects 6.1 Library is selected. Click **OK** to exit the dialog box.

2. In the module's Code window, enter the following **CreateTableInNewDB** procedure:

```
Sub CreateTableInNewDB()
    ' use the References dialog box
    ' to set up a reference to
    ' Microsoft ADO Ext. 6.0 for
    ' DDL and Security Object Library
    ' and Microsoft ActiveX Data
    ' Objects 6.1 Library
    Dim cat As ADOX.Catalog
    Dim conn As ADODB.Connection
    Dim strDb As String
    Dim strTable As String
    Dim strConnect As String

    On Error GoTo ErrorHandler

    Set cat = New ADOX.Catalog
    strDb = CurrentProject.Path & "\Sites.accdb"

    strConnect = "Provider = Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & strDb & ";"

    ' create a new database file
    cat.Create strConnect
    MsgBox "The database was created (" & strDb & ")."

    ' set connection to currently open catalog
    Set conn = cat.ActiveConnection

    strTable = "tblSchools"
    conn.Execute "CREATE TABLE " & strTable & _
        "(SchoolID AUTOINCREMENT(100, 5), " & _
        "SchoolName CHAR," & _
        "City CHAR (25), District CHAR (35), " & _
        "YearEstablished DATE);"
    ExitHere:
    Set cat = Nothing
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = -2147217897 Then
        ' delete the database file if it exists
        Kill strDb
```

```
' start from statement that caused this error
Resume 0
Else
    MsgBox Err.Number & ":" & Err.Description
    GoTo ExitHere
End If
End Sub
```

3. Position the insertion point anywhere within the `CreateTableInNewDB` procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.

The `CreateTableInNewDB` procedure shown here creates a new database named `Sites.accdb` in the current folder. You create an Access database by using the `Create` method of the ADOX Catalog object. Before creating a table in the new database, set the `conn` object variable to the currently open Catalog, like this:

```
Set conn = cat.ActiveConnection
```

Use the `Connection` object's `Execute` method to create a new table named `tblSchools`. Like other procedure examples in this section, this table contains an `AutoNumber` field with a sequence starting at 100 that will be incremented by 5 as new columns are added. Notice that the error-handling code demonstrated in this procedure is slightly different from previous examples. If you know the type of error that is most likely to occur, you can check for the error number in the error handler and execute the appropriate statement when the condition is met. If the database already exists, it will be deleted using the VBA `Kill` statement (don't do this in the production environment unless you are absolutely certain this is what you want to do). The statement `Resume 0` in the error-handling code will return the code execution to the line that caused the error. If other errors are encountered, error information will appear in a message box and the code execution will continue from the line following the `ExitHere` label.

## **DELETING TABLES**

---

It's time to remove some of our test data by using the `DROP TABLE` statement to delete an existing table from a database. Note that a table must be closed before it can be deleted. The procedure in Hands-On13.4 will delete the `tblSchools` table that was created in Hands-On 13.2.



### Hands-On 13.4 Deleting a Table

This hands-on exercise requires the prior completion of Hands-On 13.2.

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, enter the following **DeleteTable** procedure:

```
Sub DeleteTable()
    Dim conn As ADODB.Connection
    Dim strTable As String

    On Error GoTo ErrorHandler
    Set conn = CurrentProject.Connection

    strTable = "tblSchools"
    conn.Execute "DROP TABLE " & strTable
    Application.RefreshDatabaseWindow
ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = -2147217900 Then
        DoCmd.Close acTable, strTable, acSavePrompt
        Resume 0
    Else
        MsgBox Err.Number & ":" & Err.Description
        Resume ExitHere
    End If
End Sub
```

3. Position the insertion point anywhere within the code of the **DeleteTable** procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.

You can also execute the `DROP TABLE` statement directly in the Access user interface's Data Definition Query window by following these steps:

1. Choose **Create | QueryDesign**.
2. Click the **Close** button in the Show Table dialog box.
3. Choose **Design | SQL | Data Definition**.
4. Enter the following statement in the Query window:

```
DROP TABLE tblSchools;
```

5. Choose **Design | Run**.

## MODIFYING TABLES WITH DDL

---

You can modify a table definition by altering, adding, or dropping columns and constraints. *Constraints* allow you to enforce integrity by creating rules for a table. The procedures in the following sections illustrate how to use Access SQL DDL statements to

- Add new columns to a table
- Change the column's data type
- Change the size of a Text column
- Delete a field from a table
- Add a primary key to an existing table
- Add a unique, multiple-field index to an existing table
- Delete an index
- Set a default value for a column in a table
- Change the seed and increment values of AutoNumber columns

### Adding New Fields to a Table

---

Use the `ALTER TABLE` statement followed by a table name to modify the design of a table after it has been created with the `CREATE TABLE` statement. Prior to modifying the structure of an existing table, it's recommended that you make a backup copy of the table.

The `ALTER TABLE` statement can be used with the `ADD COLUMN` clause to add a new field to the table. For example, the procedure in Hands-On 13.5 adds a Currency field called `Budget2021` to the `tblSchools` table using the following statement:

```
ALTER TABLE tblSchools ADD COLUMN Budget2021 MONEY
```

When you add a new field to a table, you should specify the name of the field, its data type and, for Text and Binary fields, the size of the field.



### Hands-On 13.5 Adding a New Field to an Existing Table

Run the procedure in Hands-On 13.2 to create the `tblSchools` table in the current database if you deleted the table in Hands-On 13.4.

1. In the Visual Basic Editor window, choose **Insert | Module**.

2. In the module's Code window, enter the following **AddNewField** procedure:

```
Sub AddNewField()
    Dim conn As ADODB.Connection
    Dim strTable As String
    Dim strCol As String

    On Error GoTo ErrorHandler
    Set conn = CurrentProject.Connection

    strTable = "tblSchools"
    strCol = "Budget2021"

    conn.Execute "ALTER TABLE " & strTable & _
        " ADD COLUMN " & strCol & " MONEY;"

    ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub

ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

3. Position the insertion point anywhere within the code of the **AddNewField** procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.
4. Open the modified table to verify that the new column was created as specified.

### Changing the Data Type of a Table Column

You can use the **ALTER COLUMN** clause in the **ALTER TABLE** statement to change the data type of a table column. You must specify the name of the field, the desired data type, and the size of the field, if required.

The procedure in Hands-On 13.6 changes the data type of the **SchoolID** field in the **tblSchools** table from **AutoNumber** to a 15-character **Text** field.



#### Hands-On 13.6 Changing the Field Data Type

This hands-on exercise uses the **tblSchools** table created in Hands-On 13.2 and recreated in Hands-On 13.5.

1. In the same module where you entered the procedure in Hands-On 13.5, enter the following **ChangeFieldType** procedure:

```
Sub ChangeFieldType()
    Dim conn As ADODB.Connection
```

```
Dim strTable As String
Dim strCol As String

On Error GoTo ErrorHandler
Set conn = CurrentProject.Connection

strTable = "tblSchools"
strCol = "SchoolID"
conn.Execute "ALTER TABLE " & strTable & _
    " ALTER COLUMN " & strCol & " CHAR(15);"
ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Position the insertion point anywhere within the code of the **ChangeFieldType** procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.
3. Verify that the procedure you ran made the intended change to the SchoolID data type.

### **Changing the Size of a Text Column**

---

It's easy to increase or decrease the size of a Text column. Simply use the **ALTER TABLE** statement followed by the name of the table, and the **ALTER COLUMN** clause followed by the name of the column whose size you want to modify. Then specify the data type of the column and the new column size.

Hands-On 13.7 modifies the size of the SchoolName field from the default 255 characters to 40.



### **Hands-On 13.7 Changing the Size of a Field**

This hands-on exercise uses the **tblSchools** table created in Hands-On 13.2.

1. In the same module where you entered the procedure from the previous hands-on exercise, enter the following **ChangeFieldSize** procedure:

```
Sub ChangeFieldSize()
    Dim conn As ADODB.Connection
    Dim strTable As String
    Dim strCol As String
```

```
On Error GoTo ErrorHandler
Set conn = CurrentProject.Connection

strTable = "tblSchools"
strCol = "SchoolName"

conn.Execute "ALTER TABLE " & strTable & _
    " ALTER COLUMN " & strCol & " CHAR(40);"
ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Position the insertion point anywhere within the code of the ChangeFieldSize procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.
3. Verify that the procedure you ran made changed the SchoolName file size to 40 characters.

### **Deleting a Column from a Table**

---

Use the `DROP COLUMN` clause in the `ALTER TABLE` statement to delete a column from a table. You only need to specify the name of the field you want to remove.

The example procedure in Hands-On 13.8 deletes the Budget2021 column from the `tblSchools` table.



#### **Hands-On 13.8 Deleting a Field from a Table**

This hands-on exercise uses the `tblSchools` table created in Hands-On 13.2. Make sure this table contains the `Budget2021` column, which was added in Hands-On 13.5.

1. In the same module where you entered previous hands-on exercises, enter the following **DeleteField** procedure:

```
Sub DeleteField()
    Dim conn As ADODB.Connection
    Dim strTable As String
    Dim strCol As String

    On Error GoTo ErrorHandler
```

```
Set conn = CurrentProject.Connection

strTable = "tblSchools"
strCol = "Budget2021"

conn.Execute "ALTER TABLE " & strTable & _
    " DROP COLUMN " & strCol & ";"
ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Position the insertion point anywhere within the code of the DeleteField procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.
3. Verify that this procedure removed the Budget2021 field from the tblSchools table.

### **Adding a Primary Key to a Table**

---

You can use the **ADD CONSTRAINT** clause in the **ALTER TABLE** statement to define one or more columns as a primary key. The primary key is defined using the **PRIMARY KEY** keywords.

Hands-On 13.9 defines a primary key for the **tblSchools** table created in Hands-On 13.2. The result is shown in Figure 13.4.



### **Hands-On 13.9 Adding a Primary Key to a Table**

This hands-on exercise uses the **tblSchools** table created in Hands-On 13.2.

1. In the same module where you entered previous hands-on exercises, enter the following **AddPrimaryKey** procedure:

```
Sub AddPrimaryKey()
    Dim conn As ADODB.Connection
    Dim strTable As String
    Dim strCol As String

    On Error GoTo ErrorHandler
    Set conn = CurrentProject.Connection
```

```
strTable = "tblSchools"
strCol = "SchoolID"

conn.Execute "ALTER TABLE " & strTable & _
    " ADD CONSTRAINT pKey PRIMARY KEY " & _
    "(" & strCol & ");"
ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Position the insertion point anywhere within the code of the AddPrimaryKey procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.
3. Open the table to verify the existence of the Primary Key.

### **Adding a Multiple-Field Index to a Table**

---

Use the `ADD CONSTRAINT` clause and the `UNIQUE` keyword in the `ALTER TABLE` statement to add a multiple-field index. The `UNIQUE` keyword prevents duplicate values in the index.

#### **Hands-On 13.10 Adding a Unique Index Based on Two Fields to an Existing Table**

This hands-on exercise uses the `tblSchools` table created in Hands-On 13.2.

1. In the same module where you entered previous hands-on exercises, enter the following `AddMulti_UniqueIndex` procedure:

```
Sub AddMulti_UniqueIndex()
    Dim conn As ADODB.Connection
    Dim strTable As String
    Dim strCol As String

    On Error GoTo ErrorHandler
    Set conn = CurrentProject.Connection

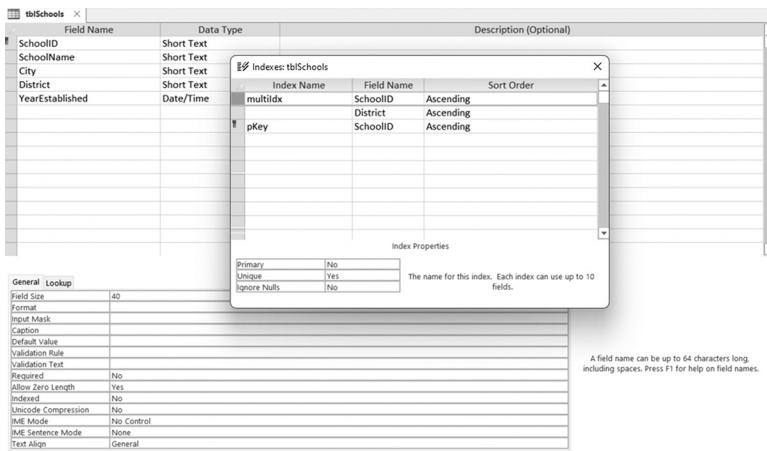
    strTable = "tblSchools"
    strCol = "SchoolID, District"
```

```

conn.Execute "ALTER TABLE " & strTable & _
    " ADD CONSTRAINT multiIdx UNIQUE " & _
    "(" & strCol & ");"
ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub

```

2. Position the insertion point anywhere within the code of the AddMulti\_UniqueIndex procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure. Figure 13.4 shows the result.



**FIGURE 13.4** After running the procedures in Hands-On 13.9 and 13.10, the **tblSchools** table contains a primary key and a unique index based on two fields.

## Deleting an Indexed Column

Deleting an index field is a two-step process:

- Use the `DROP CONSTRAINT` clause to delete an index. You must specify the index name.
- Use the `DROP COLUMN` clause to delete the desired column. You must specify the column name.

Both clauses must be used in the `ALTER TABLE` statement.

The procedure in Hands-On 13.11 deletes the District column from the tblSchools table. Recall that the procedure in Hands-On 13.10 added a multiple-field index based on the SchoolID and District columns.



### Hands-On 13.11 Deleting a Field that Is Part of an Index

This hands-on exercise uses the tblSchools table created in Hands-On 13.2. You must perform Hands-On 13.10 prior to running this procedure.

1. In the same module where you entered previous hands-on exercises, enter the following **DeleteIdxField** procedure:

```
Sub DeleteIdxField()
    Dim conn As ADODB.Connection
    Dim strTable As String
    Dim strCol As String
    Dim strIdx As String

    On Error GoTo ErrorHandler
    Set conn = CurrentProject.Connection

    strTable = "tblSchools"
    strCol = "District"
    strIdx = "multiIdx"

    conn.Execute "ALTER TABLE " & strTable & _
        " DROP CONSTRAINT " & strIdx & ","

    conn.Execute "ALTER TABLE " & strTable & _
        " DROP COLUMN " & strCol & ";"

    ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Position the insertion point anywhere within the code of the **DeleteIdxField** procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.
3. Verify that only the Primary Key exists in the table.

## Deleting an Index

Use the `DROP CONSTRAINT` clause to delete an index. You must specify the index name.

The procedure in Hands-On 13.12 deletes a primary key index from the `tblSchools` table.

### Hands-On 13.12 Deleting an Index

This hands-on exercise uses the `tblSchools` table created in Hands-On 13.2. You must perform Hands-On 13.10 prior to running this procedure.

1. In the same module where you entered previous hands-on exercises, enter the following `RemovePrimaryKeyIndex` procedure:

```
Sub RemovePrimaryKeyIndex()
    Dim conn As ADODB.Connection
    Dim strTable As String
    Dim strIdx As String

    On Error GoTo ErrorHandler
    Set conn = CurrentProject.Connection

    strTable = "tblSchools"
    strIdx = "pKey"

    conn.Execute "ALTER TABLE " & strTable & _
        " DROP CONSTRAINT " & strIdx & ";"

    ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Position the insertion point anywhere within the code of the `RemovePrimaryKeyIndex` procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.
3. Verify that `tblSchools` does not have any indexes.

## Setting a Default Value for a Table Column

Specifying a default value for a field automatically enters that value in the field each time a new record is added to a table unless the user provides a value for the field. Using DDL, you can add a default value for an existing column with the `SET DEFAULT` clause. The required syntax is as follows:

```
ALTER TABLE table_name ALTER [COLUMN] column_name SET DEFAULT  
default-value;
```

The `[COLUMN]` in the syntax is optional.



### Hands-On 13.13 Setting a Default Value for a Field

This hands-on exercise uses the `tblSchools` table created in Hands-On 13.2.

1. In the same module where you entered previous hands-on exercises, enter the following `SetDefaultFieldValue` procedure:

```
Sub SetDefaultFieldValue()  
    Dim conn As ADODB.Connection  
    Dim strTable As String  
    Dim strCol As String  
    Dim strDefVal As String  
    Dim strSQL As String  
  
    On Error GoTo ErrorHandler  
    Set conn = CurrentProject.Connection  
  
    strTable = "tblSchools"  
    strCol = "City"  
    strDefVal = "Boston"  
    strSQL = "ALTER TABLE " & strTable &  
        " ALTER " & strCol & " SET DEFAULT " & strDefVal  
  
    conn.Execute strSQL  
  
    ExitHere:  
        conn.Close  
        Set conn = Nothing  
        Exit Sub  
ErrorHandler:  
    MsgBox Err.Number & ":" & Err.Description  
    Resume ExitHere  
End Sub
```

2. Position the insertion point anywhere within the code of the SetDefaultFieldValue procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure. Figure 13.5 shows that the Default Value property of the City field has been set to Boston.

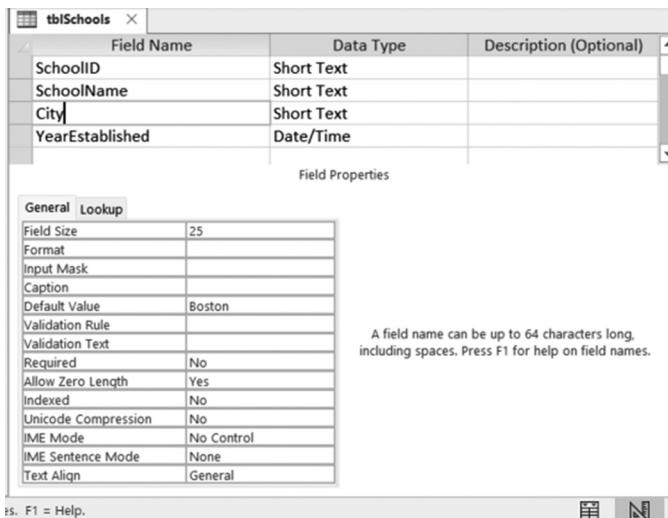


FIGURE 13.5. After running the procedure in Hands-On 13.13, the Default Value property of the City field is set to Boston.

## Changing the Seed and Increment Values of AutoNumber Columns

When a table contains a field with an AutoNumber data type, you can set a seed value and an increment value. The seed value is the initial value for the column, and the increment value is the number added to the seed value to obtain a new counter value for the next record. If not specified, both seed and increment values default to 1. You can use DDL to change the seed and increment values of AutoNumber columns by using one of the following three statements:

```
ALTER TABLE Table_name
ALTER COLUMN Column_name AUTOINCREMENT (seed, increment)
```

```
ALTER TABLE Table_name
ALTER COLUMN Column_name COUNTER (seed, increment)
```

```
ALTER TABLE Table_name
ALTER COLUMN Column_name IDENTITY (seed, increment)
```

The example procedure in Hands-On 13.14 modifies the seed value of the existing AutoNumber column in the SchoolID column to start at 1000. Because we

changed the SchoolID column's data type to the Text data type in one of the earlier hands-on exercises, you will modify the SchoolID column in the Sites.accdb file you created in Hands-On 13.3 earlier in this chapter.



### **Hands-On 13.14 Changing the Start (Seed) Value of the AutoNumber Field**

This hands-on exercise uses the Sites.accdb database file and tblSchools table created in Hands-On 13.3.

1. In the same module where you entered previous hands-on exercises, enter the following **ChangeAutoNumber** procedure:

```
Sub ChangeAutoNumber()
    Dim conn As ADODB.Connection
    Dim strDb As String
    Dim strConnect As String
    Dim strTable As String
    Dim strCol As String
    Dim intSeed As Integer

    On Error GoTo ErrorHandler

    strDb = CurrentProject.Path & "\\" & "Sites.accdb"

    strConnect = "Provider = Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & strDb & ";"

    strTable = "tblSchools"
    strCol = "SchoolID"
    intSeed = 1000

    Set conn = New ADODB.Connection
    conn.Open strConnect
    conn.Execute "ALTER TABLE " & strTable & _
        " ALTER COLUMN " & strCol & _
        " COUNTER (" & intSeed & ");"
    ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = -2147467259 Then
        MsgBox "The database file cannot be located.", _
            vbCritical, strDb
        Exit Sub
```

```
Else
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End If
End Sub
```

2. Position the insertion point anywhere within the code of the ChangeAutoNumber procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.
3. Launch Access with the **Sites.accdb** database and open the **tblSchools** table.
4. Enter a couple of new records in this table. In the **YearEstablished** field, enter the date in the format **mm/dd/yyyy**. Note that the first new record is numbered 1000, the second 1001, the third 1002, and so on.
5. Close the **Sites.accdb** database file.

## SUMMARY

---

In this chapter, you learned various Data Definition Language (DDL) commands for creating a new Access database, as well as creating, modifying, and deleting tables. You also learned how to add, modify, and delete fields and indexes, how to change the seed and increment values for AutoNumber fields, and how to change a field's data type. You also practiced assigning default values to table fields.

In the next chapter, you will learn about several DDL commands used for establishing relationships between tables and controlling referential integrity.



# Chapter 14 *ENFORCING DATA INTEGRITY AND RELATIONSHIPS BETWEEN TABLES*

When creating tables in a database, you often need to define rules regarding the values allowed in columns (fields). As mentioned in Chapter 13, constraints allow you to enforce integrity by creating rules for a table. The five types of constraints are listed in Table 14.1.

TABLE 14.1 Table constraints

Constraint Name	Usage
PRIMARY KEY	Identifies the column or set of columns whose values uniquely identify a row in a table.
FOREIGN KEY	Defines the relationship between tables and maintains data integrity when records are being added, changed, or deleted in a table.
UNIQUE	Ensures that no duplicate values are entered in a specific column or combination of columns that is not a table's primary key.
NOT NULL	Specifies that a column cannot contain a Null value. Primary key columns are automatically defined as NOT NULL. <b>Note:</b> A Null value is not the same as zero (0), blank, or a zero-length character string (""). A Null value indicates that no entry has been made. You can determine if a field contains a Null value by using the <code>IsNull</code> function.
CHECK	Enforces integrity by limiting the values that can be placed in a column.

When constraints are added, all existing data is verified for constraint violations.

## USING CHECK CONSTRAINTS

Tables and columns can contain multiple CHECK constraints. A *CHECK constraint* can validate a column value against a logical expression or another column in the same or another table. What you can't do with the CHECK constraint is to specify the custom validation message, as is possible to do in the Access user interface.

The procedure in Hands-On 14.1 uses a PRIMARY KEY constraint explicitly named PrimaryKey to identify the ID column as a primary key. The CHECK constraint used in this procedure ensures that only numbers within the specified range are entered in the YearsWorked column. You can apply CHECK constraints to a single column or to multiple columns. When a table is deleted, CHECK constraints are also dropped.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 14.1 Using a CHECK Constraint to Specify a Condition for All Values Entered for the Column

1. Start Access and create a new database named **Chap14.accdb** in your C:\VBAAccess2021\_ByExample folder.
2. Switch to the Visual Basic Editor window and choose **Tools | References**. In the References dialog box, scroll down to locate **Microsoft ActiveX Data Objects 6.1 Library**. Click the checkbox to the left of this library name to set a reference to it and click OK to exit the dialog box.
3. Choose **Insert | Module** to add a new module to the current VBA project.
4. In the module's Code window, type the **CheckColumnValue** procedure shown below.

```
Sub CheckColumnValue()
    ' you must set up a reference to the
    ' Microsoft ActiveX Data Objects Library
    ' in the References dialog box
    Dim conn As ADODB.Connection
    Dim strTable As String
    Dim strSQL As String

    strTable = "tblAwards"
    strSQL = "CREATE TABLE " & strTable
    strSQL = strSQL & "(ID AUTOINCREMENT CONSTRAINT «
```

```

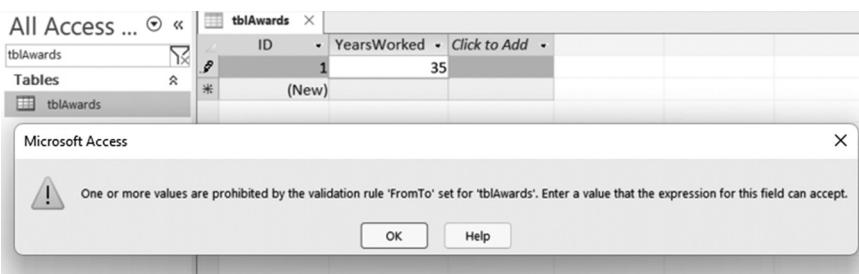
strSQL = strSQL & "PrimaryKey PRIMARY KEY,"
strSQL = strSQL & "YearsWorked INT, CONSTRAINT FromTo "
strSQL = strSQL & "CHECK (YearsWorked BETWEEN 1 AND 30));"

On Error GoTo ErrorHandler
Set conn = CurrentProject.Connection

conn.Execute strSQL
Application.RefreshDatabaseWindow
ExitHere:
conn.Close
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub

```

- Position the insertion point anywhere within the code of the **CheckColumnValue** procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.
- The **CheckColumnValue** procedure creates the **tblAwards** table with the **CHECK** constraint.
- Open the **tblAwards** table and enter a value in the **YearsWorked** column that does not fall between **1** and **30**. You should receive the message shown in Figure 14.1.



**FIGURE 14.1** This message appears when you attempt to enter a value in the YearsWorked column that is not within the range of values specified by the FromTo constraint.

The next Hands-On demonstrates how to create a CHECK constraint to ensure that the value of the **Items** column in the **tblBookOrders** table is less than the value of the **MaxUnits** column in the **tblSupplies** table for the specified ISBN number. This hands-on exercise also illustrates how to use the SQL Data Manip-

ulation Language (DML) statements INSERT INTO, BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION.

### Hands-On 14.2 Creating a Table with a Validation Rule Referencing a Column in Another Table

1. In the same module where you entered the procedure in Hands-On 14.1, enter the **ValidateAgainstCol\_InAnotherTbl** procedure shown here:

```
Sub ValidateAgainstCol_InAnotherTbl()
    Dim conn As ADODB.Connection
    Dim strTable1 As String
    Dim strTable2 As String
    Dim InTrans As Boolean

    strTable1 = "tblSupplies"
    strTable2 = "tblBookOrders"

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "BEGIN TRANSACTION"
    InTrans = True
    conn.Execute "CREATE TABLE " & strTable1 & _
        "(ISBN CHAR CONSTRAINT " & _
        "PrimaryKey PRIMARY KEY, " & _
        "MaxUnits LONG);", adExecuteNoRecords

    conn.Execute "INSERT INTO " & strTable1 & _
        " (ISBN, MaxUnits) " & _
        " Values ('158-76609-09', 5);", _
        adExecuteNoRecords

    conn.Execute "INSERT INTO " & strTable1 & _
        " (ISBN, MaxUnits) " & _
        " Values ('167-23455-69', 7);", _
        adExecuteNoRecords

    conn.Execute "CREATE TABLE " & strTable2 & _
        "(OrderNo AUTOINCREMENT CONSTRAINT " & _
        "PrimaryKey PRIMARY KEY, " & _
        "ISBN CHAR, Items LONG, " & _
        "CONSTRAINT OnHandConstr CHECK " & _
        "(Items <(SELECT MaxUnits FROM " & strTable1 & _
```

```
" WHERE ISBN = " & strTable2 & ".ISBN)));" , _  
adExecuteNoRecords  
  
conn.Execute "COMMIT TRANSACTION"  
InTrans = False  
Application.RefreshDatabaseWindow  
ExitHere:  
    conn.Close  
    Set conn = Nothing  
    Exit Sub  
ErrorHandler:  
    If InTrans Then  
        conn.Execute "ROLLBACK TRANSACTION"  
        Resume ExitHere  
    Else  
        MsgBox Err.Number & ":" & Err.Description  
        Exit Sub  
    End If  
End Sub
```

2. Position the insertion point anywhere within the code of the ValidateAgainstCol\_InAnotherTbl procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.

This procedure creates two tables. Because the Items column in the tblBookOrders table needs to be validated against the contents of the MaxUnits column in the tblSupplies table, we wrapped the process of creating these tables and entering data in the tblSupplies table into a transaction. To trap errors that could occur during the procedure execution, we declared a Boolean variable named InTrans to help us determine whether an error occurred during the transaction; if the value of InTrans is True, we will cancel the transaction.

Notice that in Access SQL syntax we use the BEGIN TRANSACTION statement to start the transaction, the COMMIT TRANSACTION statement to save the results of the transaction, and the ROLLBACK TRANSACTION statement to cancel any changes. These transaction statements can only be used through the Jet OLE DB Provider and ADO. They will cause an error when used with the Access user interface or DAO.

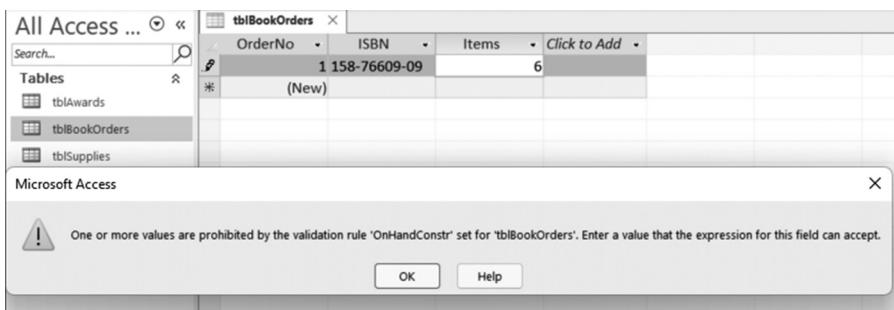
In this example procedure, we used the adExecuteNoRecords option to specify that no rows should be returned. You can use this setting with the Connection or Command object's Execute method to improve performance when no rows are returned or when you don't plan to access the returned rows in your procedure code. If you omit this setting, your ADO code will still execute successfully, but the ADO will unnecessarily create a Recordset object

as the return value for the `Execute` method. Using the `adExecuteNoRecords` setting is one of several techniques for optimizing data access using ADO.

3. Open the **tblBookOrders** table and enter the record shown at the top of Figure 14.2.

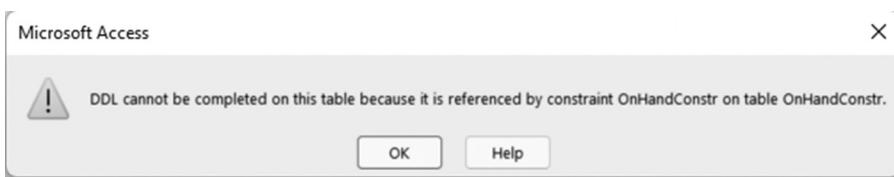
When you try to save this record or move to the next data row, Access will display a message informing you that the value you are trying to enter is prohibited.

4. Click **OK** to dismiss the message box, then press **Esc** to cancel the data entry.
5. Enter the value of **4** in the **Items** column. This time Access approves of the entry and no error message is displayed.
6. Close the **tblBookOrders** table.



**FIGURE 14.2** When you attempt to enter a value that does not meet the validation rule, Access displays an error message.

7. In the object Navigation pane on the left side of the database window, right-click the **tblBookOrders** table and choose **Delete**. Click **Yes** to confirm the deletion. Access will respond with the error message shown in Figure 14.3.



**FIGURE 14.3** If you try to manually delete a table referenced by the CHECK constraint, Access will display an error message.

Now, let's see how you can use the Access user interface to issue commands that delete tables and CHECK constraints.



### Hands-On 14.3 Deleting Tables and Constraints Using the Access User Interface

This hands-on exercise requires that you have created the *tblBookOrders* and *tblAwards* tables in Hands-On 14.1 and 14.2.

1. In the database window, choose **Create | Query Design**.
2. In the Query Type group, double-click **Data Definition**.
3. In the Data Definition Query window, enter **Drop Table *tblBookOrders*** statement as shown in Figure 14.4.

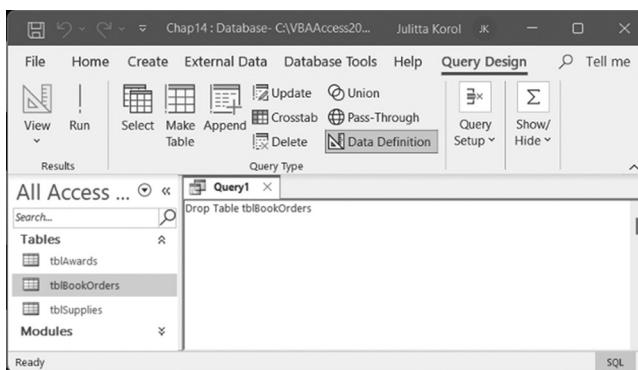


FIGURE 14.4 To delete a table that contains a CHECK constraint, type the DROP TABLE statement in the Data Definition Query window, then click Run.

4. To run the Data Definition query, click the **Run** button in the Ribbon.  
Note that a table must be closed before it can be deleted. If you don't want to delete a table but need to remove a constraint from a table, use the following syntax:

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name
```

To remove a constraint, you must know its name.

5. To delete the constraint from the *tblAwards* table, make sure that the *tblAwards* table is closed. Type the statement shown in Figure 14.5 in the Data Definition Query window and click the Run button on the Ribbon.



FIGURE 14.5. To remove a table constraint, use the DROP CONSTRAINT statement with ALTER TABLE.

**NOTE**

*Before using ALTER TABLE, it is a good idea to make a backup copy of the table.*

6. On your own, delete the tblSupplies table using the Data Definition Query window.
7. Close the Data Definition Query window without saving it.

## **ESTABLISHING RELATIONSHIPS BETWEEN TABLES**

---

To establish a link between the data in two tables, add one or more columns that hold one table's primary key values to the other table. This column becomes a *foreign key* in the second table. Using SQL, you can use a FOREIGN KEY constraint to reference another table. Foreign keys can be single- or multi column.

A *FOREIGN KEY constraint* enforces referential integrity by ensuring that changes made to data in the primary key table do not break the link to data in the foreign key table. For example, you cannot delete a record in a primary key table or change a primary key value if the deleted or changed primary key value corresponds to a value in the FOREIGN KEY constraint of another table. The REFERENCES clause identifies the parent table of the relation.

To create a brand-new table and relate it to an existing table, the following steps are required:

1. Use the CREATE TABLE statement followed by a table name.

```
CREATE TABLE tblOrder_Details
```

2. Follow the table name with one or more column definitions. A *column definition* consists of ColumnName followed by the data type and column size (if required).

```
InvoiceID CHAR, ProductId CHAR, Units LONG, Price MONEY
```

3. To designate a primary key, use the CONSTRAINT clause followed by the constraint name, the PRIMARY KEY clause, and the name of the column or columns to be designated as the primary key.

```
CONSTRAINT PrimaryKey PRIMARY KEY (InvoiceId, ProductId)
```

4. To designate a foreign key, use the CONSTRAINT clause followed by the constraint name, the FOREIGN KEY clause, and the name of the column to be designated as the foreign key.

```
CONSTRAINT fkInvoiceId FOREIGN KEY (InvoiceId)
```

5. Use the REFERENCES clause to specify the parent table to which a relationship is established.

```
REFERENCES tblProduct_Orders
```

6. If required, specify ON UPDATE CASCADE and/or ON DELETE CASCADE to enable referential integrity rules with cascading updates or deletes.

```
ON UPDATE CASCADE ON DELETE CASCADE
```

<b>NOTE</b>	<p><i>You may choose not to enforce referential integrity rules by specifying ON UPDATE NO ACTION or ON DELETE NO ACTION, or skipping the ON UPDATE or ON DELETE keywords. If you choose this path, you will not be able to change the value of a primary key if matching records exist in the foreign table.</i></p>
-------------	---

Refer to the procedure in Hands-On 14.4 to find out how to correctly combine the preceding example statements into a single SQL statement.



#### Hands-On 14.4 Relating Two Tables and Setting up Cascading Referential Integrity Rules

1. In the Visual Basic Editor window, choose **Insert | Module**.
2. In the module's Code window, enter the **RelateTables** procedure shown here:

```
Sub RelateTables()
    Dim conn As ADODB.Connection
    Dim strPrimaryTbl As String
    Dim strForeignTbl As String

    strPrimaryTbl = "tblProduct_Orders"
    strForeignTbl = "tblOrder_Details"

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "CREATE TABLE " & _
        strPrimaryTbl & _
        "(InvoiceID CHAR(15), " & _
        "PaymentType CHAR(20), " & _
        " PaymentTerms CHAR(25), " & _
        "Discount LONG, " & _
        " CONSTRAINT PrimaryKey " & _
```

```
"PRIMARY KEY (InvoiceID));", _
adExecuteNoRecords

conn.Execute "CREATE TABLE " & _
strForeignTbl & _
"(InvoiceID CHAR(15), " & _
"ProductID CHAR(15), " & _
" Units LONG, Price MONEY, " & _
"CONSTRAINT PrimaryKey PRIMARY KEY " & _
"(InvoiceID, ProductID), " & _
"CONSTRAINT fkInvoiceID " & _
"FOREIGN KEY (InvoiceID) " & _
"REFERENCES " & strPrimaryTbl & _
" ON UPDATE CASCADE ON DELETE CASCADE);", _
adExecuteNoRecords
Application.RefreshDatabaseWindow
ExitHere:
conn.Close
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub
```

3. Position the insertion point anywhere within the code of the RelateTables procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.

The RelateTables procedure creates and joins two tables. A primary key table named *tblProduct\_Orders* is created with a primary key on the *InvoiceID* field. The foreign key table named *tblOrder\_Details* is created with a multifield primary key index based on the *ProductID* and *InvoiceID* fields. The REFERENCES clause specifies the *tblProduct\_Orders* table as the parent table. The created relationship has the referential integrity rules enforced via the *ON UPDATE CASCADE* and *ON DELETE CASCADE* statements.

The outcome of the RelateTables procedure is illustrated in the following figures. Figure 14.6 displays the one-to-many relationship between *tblProduct\_Orders* and *tblOrder\_Details*.

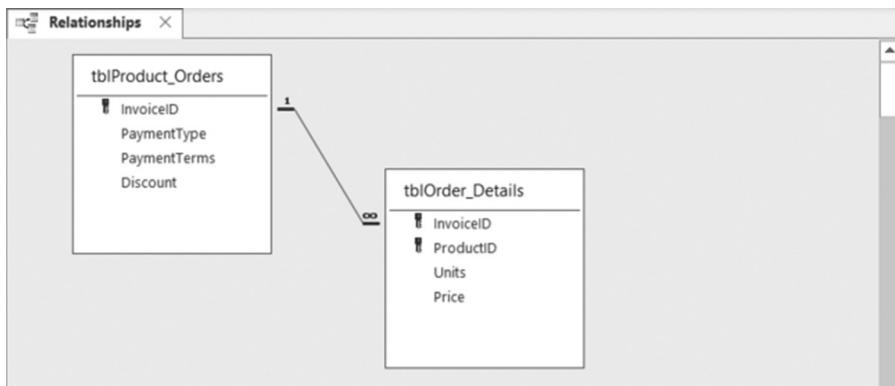


FIGURE 14.6 To access the Relationships window, choose Database Tools | Relationships.

Figure 14.7 presents the Edit Relationships window in which both cascading updates and deletes are selected.

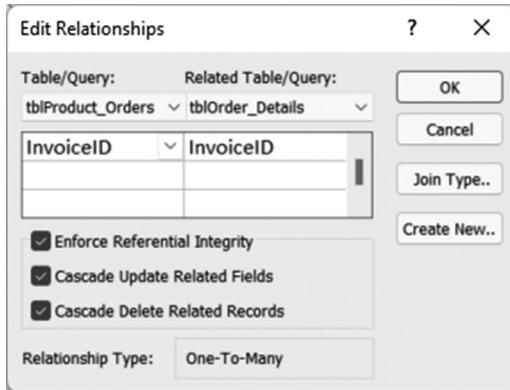


FIGURE 14.7 To access the Edit Relationships window, choose Relationships Design | Edit Relationships.

## USING THE DATA DEFINITION QUERY WINDOW

To enhance your understanding of creating tables and relationships with Data Definition Language, perform Hands-On 14.5 using the Data Definition Query window.



### Hands-On 14.5 Running DDL Statements in the Access User Interface

Each of the statements in this hands-on exercise can be executed by choosing **Design | Run**.

1. In the database window, choose **Create | Query Design**.
2. In the Query Type group of the Query Design tab, double-click **Data Definition**.
3. To create a table on the primary (one) side of the relationship, type the following statement on one line in the query window and run the query:

```
CREATE TABLE myPrimaryTbl (ID COUNTER CONSTRAINT pKey  
    PRIMARY KEY, COUNTRY TEXT(15));
```

4. To create a table on the foreign (many) side of the relationship, delete the preceding statement, then type the following statement, and run the query:

```
CREATE TABLE myForeignTbl (ID LONG, Region TEXT (15));
```

5. To create a one-to-many relationship between myPrimaryTbl and myForeignTbl, delete the preceding statement, then type the following statement on one line in the query window, and run the query:

```
ALTER TABLE myForeignTbl ADD CONSTRAINT Rel FOREIGN KEY (ID)  
    REFERENCES myPrimaryTbl (ID);
```

6. In the database window, choose **Database Tools | Relationships**.
7. In the Relationships window click the **All Relationships** button on the Ribbon.
8. This will add both tables (myPrimaryTbl and myForeignTbl) to the Relationships window (see Figure 14.8).

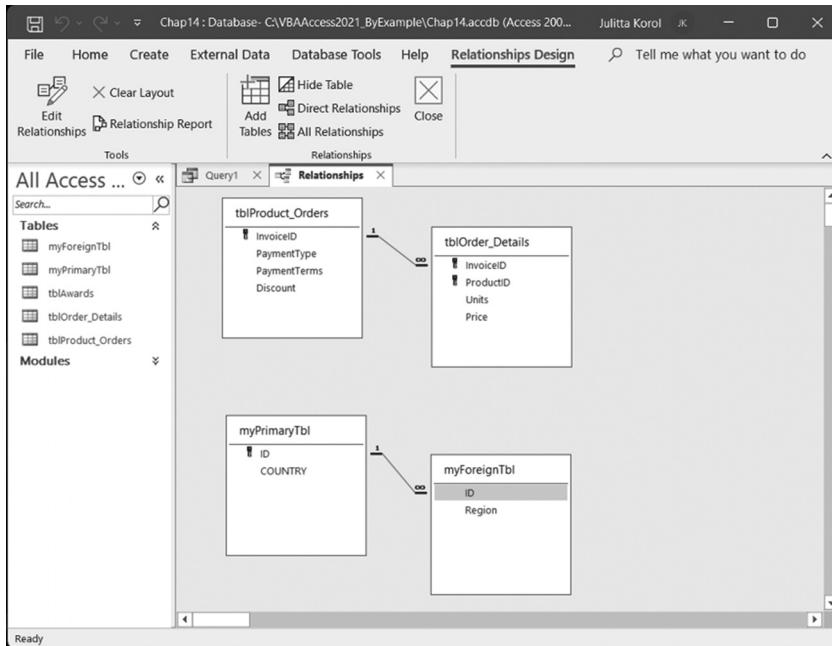


FIGURE 14.8 Notice that the tables you created by running the DDL statements in steps 5 and 6 are joined on the ID column (see step 7).

9. Right-click the line that joins the myPrimaryTbl and myForeignTbl tables and choose **Edit Relationship** to open the Edit Relationships dialog box, as shown in Figure 14.9. You can also double-click the line to open this dialog box.

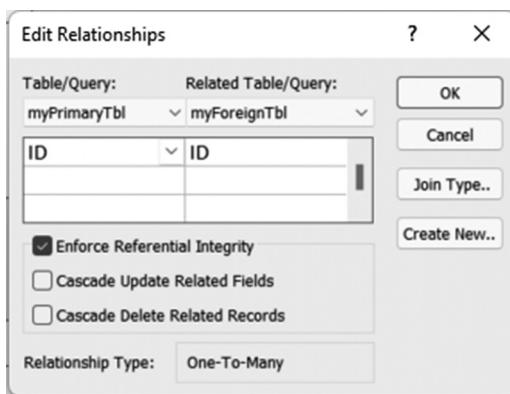


FIGURE 14.9 You can edit relationships between tables via the Edit Relationships window.

10. Click **Close** to exit the Edit Relationships window.
11. To delete the relationship between the tables, type the following statement in the Data Definition Query window (overwriting the previously entered statement), and run the query:

```
ALTER TABLE myForeignTbl DROP CONSTRAINT Rel;
```

12. To delete the table on the one side (myPrimaryTbl), type the following statement in the Data Definition Query window (overwriting the preceding statement), and run the query:

```
DROP TABLE myPrimaryTbl;
```

13. To delete the table on the many side (myForeignTbl), type the following statement in the Data Definition Query window (overwriting the preceding statement), and run the query:

```
DROP TABLE myForeignTbl;
```

14. Close the Data Definition query window without saving.

## SUMMARY

---

In this short chapter, you learned how to enforce data integrity by creating rules for tables with constraints. You learned how to validate data against another column in the same table or a column located in another table. You also learned how to use the Access Data Definition Query window to delete tables that have constraints and remove constraints from a table. Finally, you saw how you can establish relationships between tables using DDL commands inside a VBA procedure.

The next chapter focuses on ways to use DDL for defining and removing indexes and primary keys.

# Chapter 15 *DEFINING INDEXES AND PRIMARY KEYS*

Indexes speed the processes of finding and sorting records. You should create indexes for fields that are frequently used in searches and in sorting. You can create an index on a new or existing table. An index can be made of one or more fields. This chapter presents a number of procedures that use Data Definition Language statements to define indexes and primary keys.

## CREATING TABLES WITH INDEXES

You can create an index while creating a table by using the `CONSTRAINT` clause with the `CREATE TABLE` statement. The procedure in Hands-On 15.1 creates a new table called `Supplier1` with a unique index called `idxSupplierName` based on the `SupplierName` field.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 15.1 Creating a Table with a Single-Field Index

1. Start Access and create a new database named `Chap15.accdb` in your `C:\VBAAccess2021_ByExample` folder.
2. Switch to the Visual Basic Editor window and choose **Tools | References**. In the References dialog box, scroll down to locate **Microsoft ActiveX Data Objects 6.1 Library**. Click the checkbox to the left of this library name to set a reference to it and click **OK** to exit the dialog box.
3. Choose **Insert | Module** to add a new module to the current VBA project.
4. In the module's Code window, type the following **SingleField\_Index** procedure:

```
Sub SingleField_Index()
    Dim conn As ADODB.Connection
    Dim strTable As String

    strTable = "Supplier1"

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "CREATE TABLE " & strTable _
        & "(SupplierID INTEGER, " _
        & "SupplierName CHAR (30), " _
        & "SupplierPhone CHAR (12), " _
        & "SupplierCity CHAR (19), " _
        & "CONSTRAINT idxSupplierName UNIQUE " _
        & "(SupplierName));"
    Application.RefreshDatabaseWindow
ExitHere:
    conn.Close
    Set conn = Nothing
Exit Sub
```

```
ErrorHandler:  
    MsgBox Err.Number & ":" & Err.Description  
    Resume ExitHere  
End Sub
```

5. Position the insertion point anywhere within the code of the SingleField\_Index procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.

When you run this procedure then switch to the Access database window, you will notice a table named Supplier1. There is a unique index on the SupplierName field, as shown in Figure 15.1.

Indexes: Supplier1		
Index Name	Field Name	Sort Order
idxSupplierName	SupplierName	Ascending
Index Properties		
Primary	No	
Unique	Yes	The name for this index. Each index can use up to 10 fields.
Ignore Nulls	No	

FIGURE 15.1. The idxSupplierName index was created by running the procedure in Hands-On 15.1.

## ADDING AN INDEX TO AN EXISTING TABLE

To add an index to an existing table, use the `CREATE INDEX` statement. You can add an index based on one or more fields. The procedure in Hands-On 15.2 demonstrates how to add an index to the Supplier1 table you created in Hands-On 15.1.

### Hands-On 15.2 Adding a Single-Field Index to an Existing Table

1. In the same module where you entered the procedure in Hands-On 15.1, enter the **SingleField\_Index2** procedure shown here:

```
Sub SingleField_Index2()  
    Dim conn As ADODB.Connection
```

```

Dim strTable As String

strTable = "Supplier1"
On Error GoTo ErrorHandler

Set conn = CurrentProject.Connection

conn.Execute "CREATE INDEX idxCity ON " & strTable & _
    "(SupplierCity);"
ExitHere:
conn.Close
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub

```

2. Position the insertion point anywhere within the code of the SingleField\_Index2 procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.

The preceding procedure adds a single-field index named idxCity to the Supplier1 table. The following procedure will add a multiple-field index named idxSupplierNameCity to the Supplier2 table.

```

Sub MultiField_Index()
    Dim conn As ADODB.Connection
    Dim strTable As String

    strTable = "Supplier2"
    On Error GoTo ErrorHandler
    Set conn = CurrentProject.Connection

    conn.Execute "CREATE TABLE " & strTable & _
        "& "(SupplierID INTEGER, " &_
        "& "SupplierName CHAR(30), " &_
        "& "SupplierPhone CHAR(12), " &_
        "& "SupplierCity CHAR(19), " &_
        "& "CONSTRAINT idxSupplierNameCity UNIQUE " &_
        "& "(SupplierName, SupplierCity));"

    Application.RefreshDatabaseWindow
ExitHere:
conn.Close

```

```
Set conn = Nothing
Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

## CREATING A TABLE WITH A PRIMARY KEY

---

When you create a database table, you should define a primary key to uniquely identify rows within the table. A *primary key* allows you to relate a particular table with other tables in the database (for procedure examples, refer to the previous chapter). A table can have only one primary key; however, a primary key can consist of more than one column.

To create a table with a primary key, use the `CONSTRAINT` clause with the `CREATE TABLE` statement. The procedure in Hands-On 15.3 uses the following `CONSTRAINT` clause to create a single-field primary key based on the `SupplierID` field:

```
CONSTRAINT idxPrimary PRIMARY KEY(SupplierID)
```

To create a table with a primary key based on two or more columns, specify column names in parentheses following the `PRIMARY KEY` keywords. For example, the following `CONSTRAINT` clause will create a primary key based on the `SupplierID` and `SupplierName` columns:

```
CONSTRAINT idxPrimary PRIMARY KEY (SupplierID, SupplierName)
```



### Hands-On 15.3 Creating a Single-Field Primary Key

1. Switch to the Visual Basic Editor window and insert a new module.
2. In the module's Code window, enter the `SingleField_PKey` procedure shown here:

```
Sub SingleField_PKey()
    Dim conn As ADODB.Connection
    Dim strTable As String

    strTable = "Supplier3"
    On Error GoTo ErrorHandler
```

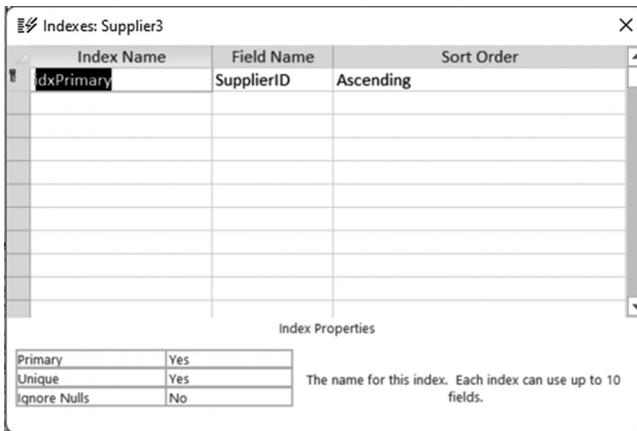
```

Set conn = CurrentProject.Connection

conn.Execute "CREATE TABLE " & strTable _
& "(SupplierID INTEGER, " _
& "SupplierName CHAR(30), " _
& "SupplierPhone CHAR(12), " _
& "SupplierCity CHAR(19), " _
& "CONSTRAINT idxPrimary PRIMARY KEY " _
& "(SupplierID));"
Application.RefreshDatabaseWindow
ExitHere:
conn.Close
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub

```

3. Position the insertion point anywhere within the code of the SingleField\_PKey procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure. After running this procedure, you will have a primary key index named idxPrimary based on the SupplierID column, as shown in Figure 15.2.



**FIGURE 15.2.** The result of running the SingleField\_PKey procedure in Hands-On 15.3 is a primary key index named idxPrimary based on the SupplierID column.

## CREATING INDEXES WITH RESTRICTIONS

You can use the `CREATE INDEX` statement to add an index to an existing table. The `CREATE INDEX` statement can be used with the following options:

- `PRIMARY` option—Creates a primary key index that does not allow duplicate values in the key.
- `DISALLOW NULL` option—Creates an index that does not allow adding records with Null values in the indexed field.
- `IGNORE NULL` option—Creates an index that does not include records with Null values in the key.

Use the `WITH` keyword to declare the preceding index options.

The procedure in Hands-On 15.4 designates the `SupplierID` field as the primary key by using the `PRIMARY` option (see Figure 15.3).



### Hands-On 15.4 Creating a Primary Key Index with Restrictions

1. Switch to the Visual Basic Editor window and insert a new module.
2. In the module's Code window, enter the following `Index_WithPrimaryOption` procedure:

```
Sub Index_WithPrimaryOption()
    Dim conn As ADODB.Connection
    Dim strTable As String

    strTable = "Supplier1"
    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "CREATE INDEX idxPrimary1 ON " & strTable _
        & "(SupplierID) WITH PRIMARY;"
    ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

- Position the insertion point anywhere within the code of the Index\_WithPrimaryOption procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.



FIGURE 15.3. The index created by the procedure in Hands-On 15.4 has the Primary and Unique properties set to Yes, which means that this index is a primary key and every value in this index must be unique.

#### NOTE

*Primary key indexes are automatically created as unique indexes.*

You can prohibit the entry of Null values in the indexed fields by using the DISALLOW NULL option as shown in the example procedure in Hands-On 15.5. The result of running this procedure is an index called idxSupplierCity that does not allow Null values, as shown in Figure 15.4.



#### Hands-On 15.5 Creating an Index that Disallows Null Values in the Key

- Switch to the Visual Basic Editor window and insert a new module.
- In the module's Code window, enter the following **Index\_WithDisallowNullOption** procedure:

```
Sub Index_WithDisallowNullOption()
    Dim conn As ADODB.Connection
    Dim strTable As String

    strTable = "Supplier3"
    On Error GoTo ErrorHandler
```

```

Set conn = CurrentProject.Connection

conn.Execute
"CREATE INDEX idxSupplierCity ON " & strTable _
& "(SupplierCity) WITH DISALLOW NULL;"

ExitHere:
conn.Close
Set conn = Nothing
Exit Sub

ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub

```

- Position the insertion point anywhere within the code of the `Index_WithDisallowNullOption` procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.

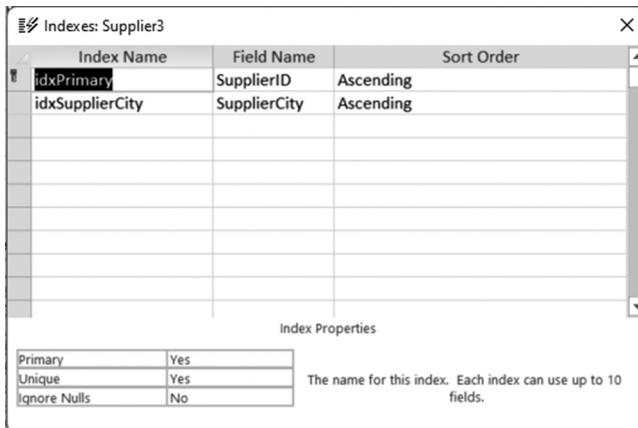


FIGURE 15.4. The result of running the procedure in Hands-On 15.5 is an index called `idxSupplierCity` that does not allow Null values.

You can prevent records with Null values in the indexed fields from being included in the index by using the `IGNORE NULL` option, as illustrated in Hands-On 15.6. Figure 15.5 shows the result of this procedure.

### ④ Hands-On 15.6 Creating an Index with the Ignore Null Option

- Switch to the Visual Basic Editor window and insert a new module.
- In the module's Code window, enter the following `Index_WithIgnoreNullOption` procedure:

```

Sub Index_WithIgnoreNullOption()
    Dim conn As ADODB.Connection
    Dim strTable As String

    strTable = "Supplier3"

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection
    conn.Execute "CREATE INDEX idxSupplierPhone ON " _
    & strTable _
        & "(SupplierPhone) WITH IGNORE NULL;"
    ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub

```

3. Position the insertion point anywhere within the code of the `Index_WithIgnoreNullOption` procedure and press F5 or choose **Run | Run Sub/UserForm** to execute the procedure.



**FIGURE 15.5.** The result of running the procedure in Hands-On 15.6 is an index called `idxSupplierPhone` that allows Null values in the key. However, records containing Null values will be excluded from any searches that use that index.

## DELETING INDEXES

---

Use the `DROP INDEX` statement to remove an index. Anytime you want to delete a column that is part of an index, you must first remove the index using the `DROP CONSTRAINT` or `DROP INDEX` statement. Before removing the index, make sure that the table containing the index is closed. The procedure in Hands-On 15.7 deletes the index named `idxSupplierName` from the `Supplier1` table.



### Hands-On 15.7 Deleting an Index

1. Switch to the Visual Basic Editor window and insert a new module.
2. In the module's Code window, enter the following **DeleteIndex** procedure:

```
Sub DeleteIndex()
    Dim conn As ADODB.Connection
    Dim strTable As String

    strTable = "Supplier1"
    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "DROP INDEX idxSupplierName ON " _
        & strTable & ";"
    ExitHere:
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

3. Position the insertion point anywhere within the code of the `DeleteIndex` procedure and press **F5** or choose **Run | Run Sub/UserForm** to execute the procedure.
4. On your own, write a procedure to remove other indexes created in this chapter's procedures.

## SUMMARY

This chapter introduced you to using DDL statements for creating indexes. Columns that are frequently used in database queries should be indexed to allow for faster access to the information. However, if you frequently add, delete, and update rows, you might want to limit the number of indexes, as they take up disk space and slow data operations. You also learned that a primary key is a special type of index that allows you to uniquely identify rows in a table as well as create a relationship between two tables, as demonstrated in the previous chapter.

In the next chapter, you will learn how to organize your data using structures known as views and how to use stored procedures in lieu of Access Action and Parameter queries.

# Chapter 16 VIEWS AND STORED PROCEDURES

In this chapter, we will work with advanced Data Definition Language statements that are used for creating, altering, and deleting two special database objects known as views and stored procedures. These objects are used to perform various query operations. *Views* are like Access Select queries; however, you can't use the `ORDER BY` clause to sort your data or use parameters to filter records. *Stored procedures* perform the same operations as Access Action and Parameter queries. They can also be used for creating sorted Select queries. Stored procedures are saved precompiled so that at runtime the procedure executes much faster than a standard SQL statement. Learning how to create and use views and stored procedures will give you more control over your database.

## **CREATING A VIEW**

If you want users to view and update data in a table or set of tables, but you do not want them to open the underlying tables directly, you can create a view. An SQL *view* is like a virtual table. Similar to an Access Select query, a view can display data from one or more tables. Instead of providing all the available data in your tables, you decide exactly what fields you'd like to include for viewing.

To create a view, use a `SELECT` statement to select the columns you want to include in the view and the `FROM` keyword to specify the table. Next, associate the `SELECT` statement with a `CREATE VIEW` statement. The syntax looks like this:

```
CREATE VIEW viewName [ (columnNames) ]
AS
SELECT (columnNames)
FROM tableName;
```

Views must have unique names in the database. The name of the view cannot be the same as the name of an existing table. Specifying the names of columns following the name of the view is optional (note the square brackets in the preceding syntax). Column names must be specified in the `SELECT` statement. Use the asterisk (\*) to select all columns.

Let's put more meaning into the preceding syntax. The following example statement creates a view that lists only orders with a Freight amount less than \$20.

```
CREATE VIEW cheapFreight
AS
SELECT Orders.OrderID,
Orders.[Shipping Fee],
Orders.[ShipCountry/Region]
FROM Orders
WHERE Orders.[Shipping Fee] < 20;
```

The `SELECT` statement that defines the view cannot contain any parameters and cannot be typed directly in the SQL pane of the Query window. It must be used through the ADO Connection object's `Execute` method after establishing the connection to a database, as illustrated here:

```
Sub Create_View_CheapFreight()
    Dim conn As ADODB.Connection
    Set conn = CurrentProject.Connection
    conn.Execute "CREATE VIEW CheapFreight AS " & _
        "SELECT Orders.[Order ID], Orders.[Shipping Fee], " & _
        "Orders.[Ship Country/Region] " & _
        "FROM Orders WHERE Orders.[Shipping Fee] < 20;"
    Application.RefreshDatabaseWindow
    conn.Close
    Set conn = Nothing
End Sub
```

The `Application.RefreshDatabaseWindow` statement ensures that after the view is created it is immediately listed in the Navigation pane in the Access

application window. If you omit this statement, you will need to refresh the Navigation pane manually by selecting any object in it and pressing Shift+F5.

To return data from the CheapFreight view, simply double-click its name in the Navigation pane.

A view can be used as if it were a table. The following statement can be used to return all records from the CheapFreight view:

```
SELECT * FROM CheapFreight;
```

Remember that a view never stores any data; it simply returns the data as stated in the `SELECT` statement used in the view definition. Because a view is like a Select query, you can use the `OpenQuery` method of the Access `DoCmd` object to open it from your VBA code:

```
Sub OpenView()
    DoCmd.OpenQuery "CheapFreight", acViewNormal
End Sub
```

The `OpenQuery` method is used to carry out the `OpenQuery` action in Visual Basic.

To get working experience with the views, let's proceed to the hands-on section. We will start by creating a view called `vw_Employees`. This view is based on the `Employees` and `Orders` tables, and contains four columns (Employee ID, Full Name, Job Title, and Order ID).

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 16.1 Creating a View Based on a Table

1. Start Access and create a new database named `Chap16.accdb` in your C:\VBAAccess2021\_ByExample folder.
2. Choose **External Data | New Data Source | From Database | Access**.
3. In the File name box of the Get External Data dialog box, enter C:\VBAAccess2021\_ByExample\Northwind 2007.accdb and click **OK**.
4. In the Import Objects dialog box, select the **Employees**, **Orders**, and **Shippers** tables and click **OK**.
5. Click **Close** to exit the Get External Data dialog box.  
The Employees, Orders and Shippers tables are now listed in the Navigation pane.
6. Switch to the Visual Basic Editor window and choose **Tools | References**.  
In the References dialog box, scroll down to locate **Microsoft ActiveX Data**

**Objects 6.1 Library.** Click the checkbox to the left of this library name to set a reference to it and click **OK** to exit the dialog box.

7. Choose **Insert | Module** to add a new module to the current VBA project.
8. In the module's Code window, type the following **Create\_View** procedure:

```
' Don't forget to set up a reference to the
' Microsoft ActiveX Data Objects 6.1 Library
' in the References dialog box

Sub Create_View()
    Dim conn As ADODB.Connection

    Set conn = CurrentProject.Connection

    On Error GoTo ErrorHandler

    conn.Execute _
        "CREATE VIEW vw_Employees AS " & _
        "SELECT Employees.ID AS [Employee ID], " & _
        "[First Name] & chr(32) & [Last Name] " & _
        "AS [Full Name], " & _
        "[Job Title], Orders.[Order ID] " & _
        "AS [Order ID] " & _
        "FROM Employees " & _
        "INNER JOIN Orders ON " & _
        "Orders.[Employee ID] = Employees.ID;" & _
        Application.RefreshDatabaseWindow
    ExitHere:
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = -2147217900 Then
        conn.Execute "DROP VIEW vw_Employees"
        Resume
    Else
        MsgBox Err.Number & ":" & Err.Description
        Resume ExitHere
    End If
End Sub
```

9. Run the **Create\_View** procedure.

This procedure creates a view named **vw\_Employees**. If the view already exists, it will be deleted using the **DROP VIEW** statement. The **chr(32)** statement will

insert a space between the first and last name. Another way to add a space is by using the `Space(1)` function, like this:

Notice that views don't differ much from saved queries. When you open the view created by the `Create_View` procedure in Design view, you will notice that this view is simply a Select query. Because the query defined by the `SELECT` statement is updatable, the `vw_Employees` view is also updatable. If the query were not updatable, the view would be read-only.

Views cannot contain the `ORDER BY` clause. To return the records in a specific order, you might want to use the view in a stored procedure, as discussed later in this chapter.

## ENUMERATING VIEWS

---

You can find out the names of the views that have been created by iterating through the `Views` collection of the ADOX Catalog object, as illustrated in Hands-On 16.2.



### Hands-On 16.2 Generating a List of Saved Views

1. In the Visual Basic Editor window, choose **Tools | References**. In the References dialog box, scroll down to locate **Microsoft ADO Ext. 6.0 for DDL and Security Object Library**. Click the checkbox to the left of this library name to set a reference to it and click **OK** to exit the dialog box.
2. Choose **Insert | Module** to add a new module to the current VBA project.
3. In the module's Code window, enter the `List_Views` procedure shown here:

```
' Don't forget to set up a reference to the
' Microsoft ADO Ext. 2.8 for DDL and Security
```

```
Sub List_Views()
    Dim cat As New ADOX.Catalog
    Dim myView As ADOX.View

    cat.ActiveConnection = CurrentProject.Connection

    For Each myView In cat.Views
        Debug.Print myView.Name
    Next myView
End Sub
```

4. Run the List\_VIEWS procedure.

The List\_VIEWS procedure writes the names of the existing views to the Immediate window. You should see the name of the vw\_Employees view created in Hands-On 16.1.

## DELETING A VIEW

---

Use the `DROP VIEW` statement to delete a particular view from the database. You must specify the name of the view you want to delete. The following example procedure deletes a view named vw\_Employees that was created by the procedure in Hands-On 16.1.

Note that both the `CREATE VIEW` and `DROP VIEW` statements can only be executed using the `Execute` method of the ADO Connection object.



### Hands-On 16.3 Deleting a View

In the same module where you entered the procedure in Hands-On 16.2, enter the `Delete_View` procedure shown here:

```
Sub Delete_View()
    Dim conn As ADODB.Connection

    Set conn = CurrentProject.Connection

    On Error GoTo ErrorHandler
    conn.Execute "DROP VIEW vw_Employees"
    ExitHere:
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = -2147217865 Then
        MsgBox "The view was already deleted."
        Exit Sub
    Else
        MsgBox Err.Number & ":" & Err.Description
        Resume ExitHere
    End If
End Sub
```

5. Run the `Delete_View` procedure.

Run the List\_VIEWS procedure from Hands-On 16.2 to find out if the vw\_Employees view was deleted.

## CREATING A STORED PROCEDURE

---

Stored procedures allow you to perform bulk operations that delete, update, or append records. Unlike views, stored procedures allow the ORDER BY clause and parameters. Use the CREATE PROCEDURE (or CREATE PROC) statement to create a stored procedure. You must specify the name of the stored procedure and the AS keyword followed by the desired SQL statement that performs the required database operation. The syntax is as follows:

```
CREATE PROC [EDURE] procName
[(param1 datatype1 [, param2 datatype2 [, ... ] ])]
AS
    sqlStatement;
```

The name of the stored procedure cannot be the same as the name of an existing table. To pass values to a stored procedure, include parameters after the procedure name. Parameter names are followed by a data type and are separated by commas. The parameters are listed in parentheses (see Hands-On 16.4 in the next section). Up to 255 parameters can be specified in the parameter list. If the stored procedure does not require parameters, the AS keyword immediately follows the name of the stored procedure.

The SQL statement for the stored procedure can be prepared in the Access Query Design window and then copied to the VBA procedure from the SQL view and appropriately formatted.

To return the employee records from the vw\_Employees view (see Hands-On 16.1) ordered by Full Name, the following stored procedure can be written:

```
CREATE PROCEDURE usp_EmpByFullName
AS
    SELECT * FROM vw_Employees
    ORDER BY [Full Name];
```

This stored procedure selects all columns that exist in the vw\_Employees view and orders the returned data by the Full Name field. Notice that this procedure does not require any parameters. You might want to precede the name of the stored procedure with a prefix indicating the type of stored procedure. The “usp” prefix is often used to indicate a user-defined stored procedure.

Like views, stored procedures are created via the ADO Connection object's `Execute` method after establishing a connection to the database. Therefore, you can use the following VBA code to create the `usp_EmpByFullName` stored procedure. If you want to try this on your own, make sure to re-run the procedure that creates `vw_Employees`, as this view was deleted in the previous Hands-On:

```
Sub Create_StoredProc()
    Dim conn As ADODB.Connection

    Set conn = CurrentProject.Connection
    conn.Execute "CREATE PROCEDURE usp_EmpByFullName AS " & _
        "SELECT * FROM vw_Employees " & _
        "ORDER BY [Full Name];"
    Application.RefreshDatabaseWindow
    conn.Close
    Set conn = Nothing
End Sub
```

Once created, stored procedures can be executed in the Access user interface by double-clicking the stored procedure name in the Navigation pane, or from VBA code by calling the `EXECUTE` statement with the ADO Connection object's `Execute` method (see Hands-On 16.5).

## CREATING A PARAMETERIZED STORED PROCEDURE

---

Most advanced stored procedures require one or more parameters. The parameters are then used as part of the SQL statement, usually the `WHERE` clause. When creating a parameterized stored procedure, Access allows you to specify up to 255 parameters in the parameters list. The stored procedure parameters must be separated by commas and enclosed in parentheses.

The procedure in Hands-On 16.4 creates a stored procedure that allows you to insert a new record into the `Shippers` table on the fly by supplying the required parameter values. Note that the SQL Data Manipulation Language (DML) `INSERT INTO` statement is used for adding new records to a table.

### **Hands-On 16.4 Creating a Stored Procedure that Accepts Parameters**

1. Switch to the Visual Basic Editor window and insert a new module.
2. In the module's Code window, enter the following `Create_SpWithParam` procedure:

```
Sub Create_SpWithParam()
```

```
Dim conn As ADODB.Connection

On Error GoTo ErrorHandler

Set conn = CurrentProject.Connection

conn.Execute _
"CREATE PROCEDURE usp_procEnterData " & _
"(@Company TEXT (50), " & _
"@Tel TEXT (25)) AS " & _
"INSERT INTO Shippers " & _
"(Company, [Business Phone]) " & _
"VALUES (@Company, @Tel);"
Application.RefreshDatabaseWindow
ExitHere:
If Not conn Is Nothing Then
    If conn.State = adStateOpen Then conn.Close
End If
Set conn = Nothing
Exit Sub
ErrorHandler:
If InStr(1, Err.Description, _
"procEnterData") Then
    conn.Execute "DROP PROC procEnterData"
    Resume
Else
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End If
End Sub
```

**3.** Run the Create\_SpWithParam procedure.

The preceding stored procedure requires two values to be entered at runtime. The first value is passed by the @Company parameter and the second value by the @Tel parameter. In this example, the names of the parameters have been preceded with the @ sign for easy migration of the stored procedure into the SQL Server environment. If you omit the @ sign, the procedure will still execute correctly in Access. If the procedure already exists, it will be dropped using the `DROP PROC` statement.

Like views, stored procedures appear in the Navigation pane in the Access application window. Because we used the `SQL INSERT INTO` statement, Access treats this stored procedure as a parameterized Append query.

**4.** Run the stored procedure named `usp_procEnterData` by double-clicking its name in the Navigation pane of the Access application window. Figures 16.1

through 16.4 outline the process of running this stored procedure, and Figure 16.5 shows the result.

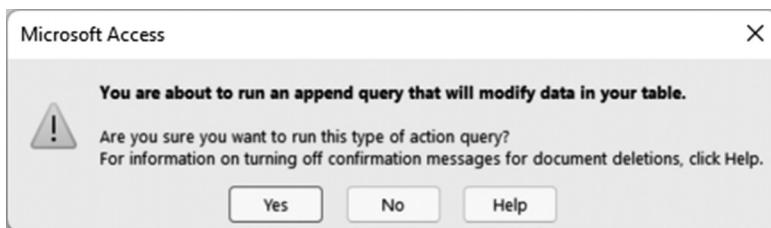


FIGURE 16.1 When you double-click a stored procedure name in the Navigation pane of the Access database window, Access displays this message when the stored procedure expects parameters and its SQL statement attempts to insert data into a table.

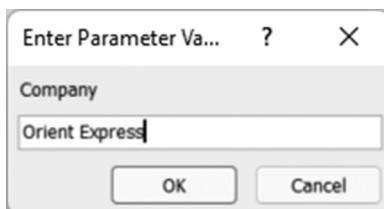


FIGURE 16.2 Because the stored procedure expects some input, you are prompted for the first parameter value.

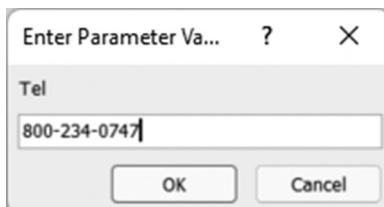


FIGURE 16.3. Here you are prompted to enter the phone number for the second stored procedure parameter.

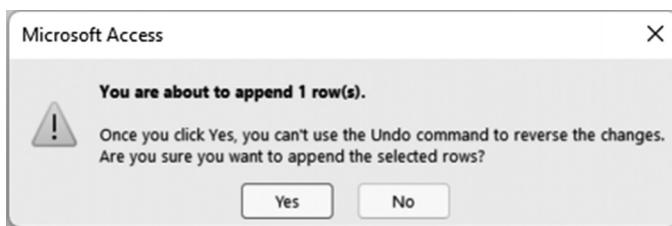


FIGURE 16.4. Once all input has been gathered via the parameters, Access informs you about the action that is to be performed. Click Yes to execute the stored procedure or No to cancel.

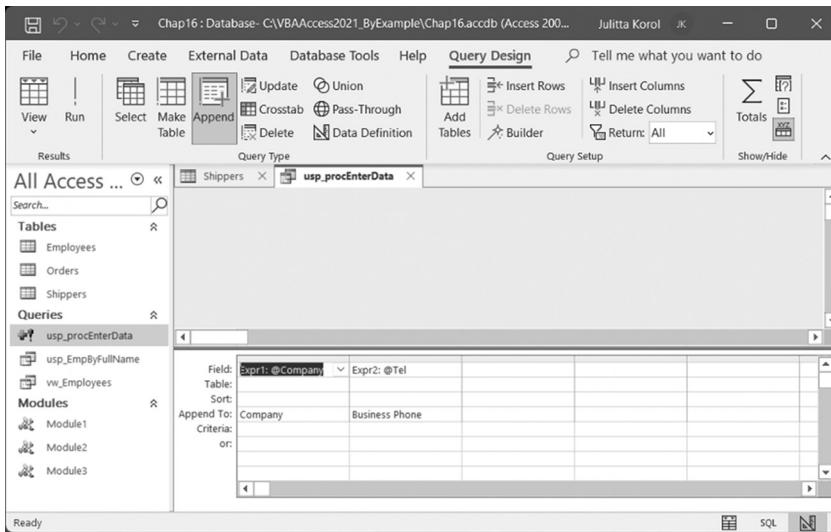
The screenshot shows the 'Shippers' table in Microsoft Access. The table has columns: ID, Company, Last Name, First Name, E-mail Address, Job Title, and Business Phone. There are seven records listed: 'Shipping Company A', '2 Shipping Company B', '3 Shipping Company C', '4 Orient Express', and a new record '(New)' which has been added. The 'Business Phone' field for the new record contains the value '800-234-0747'.

ID	Company	Last Name	First Name	E-mail Address	Job Title	Business Phone
1	Shipping Company A					
2	2 Shipping Company B					
3	3 Shipping Company C					
4	Orient Express					800-234-0747
*	(New)					

**FIGURE 16.5** After you click Yes, Access runs the Append query. To view the result of this operation, double-click the Shippers table in the Navigation pane. Notice that a new record (Orient Express) was added to the Shippers table.

## EXAMINING THE CONTENTS OF A STORED PROCEDURE

You can examine the contents of the stored procedure created in Hands-On 16.4 by right-clicking on the usp\_procEnterData procedure in the Navigation pane and choosing Design View. Figure 16.6 displays the Design view of the Append query. Other stored procedures that you create may be presented as different Action queries.



**FIGURE 16.6.** To view or modify the contents of a stored procedure, open it in Design view.

You can examine the SQL statements used by Access to execute your stored procedure by switching to the SQL view (click Query Design | View and select SQL View), as shown in Figure 16.7.



```
PARAMETERS Company Text ( 255 ), Tel Text ( 255 );
INSERT INTO Shippers ( Company, [Business Phone] )
SELECT @Company AS Expr1, @Tel AS Expr2;
```

FIGURE 16.7. The SQL view of the Query window displays the SQL statement that Access will execute when you run the stored procedure created in Hands-On 16.4.

## EXECUTING A PARAMETERIZED STORED PROCEDURE

---

In the preceding section, you learned how to run a parameterized stored procedure from the Access user interface. To execute an existing stored procedure from VBA code, use the `Execute` method of the ADO Connection or Command object. Here's how:

- With the `Execute` method of the Connection object:

```
conn.Execute "usp_procEnterData"
```

- With the `Execute` method of the Command object:

```
cmd.CommandText = "usp_procEnterData"
cmd.CommandType = adCmdStoredProc
```

```
cmd.Execute
rst.Open cmd
```

If the stored procedure requires parameters, parameter values follow the procedure name as a comma-separated list. Here's an example procedure that executes the `usp_procEnterData` stored procedure and contains the values for its two parameters:

```
Sub RunProc_WithParam()
    Dim conn As ADODB.Connection

    Set conn = CurrentProject.Connection
    conn.Execute _
"usp_procEnterData ""My Company2""", """(234) 334-3344"""
    conn.Close
    Set conn = Nothing
End Sub
```

Instead of surrounding parameters with sets of double quotes, you can use single quotes like this:

```
conn.Execute "usp_procEnterData 'My Company2', '(234) 334-3344'"
```

The procedure in Hands-On 16.5 runs the stored procedure named `usp_procEnterData` created in Hands-On 16.4. Notice how this procedure uses the `InputBox` function to obtain the parameter values from the user instead of hard-coding them in the `Execute` method of the `Connection` object (as shown in the preceding example). Still another way of providing parameter values to a stored procedure would be via an Access form. This is left for you to try on your own.



### Hands-On 16.5 Executing a Parameterized Stored Procedure

1. Switch to the Visual Basic Editor window and insert a new module.
2. In the module's Code window, enter the following `Execute_StoredProcWithParam` procedure:

```
Sub Execute_StoredProcWithParam()
    Dim conn As ADODB.Connection
    Dim param1 As String
    Dim param2 As String
    Dim strSQL As String

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    param1 = InputBox("Please enter " & _
                      "company name:", "Company")
    param2 = InputBox("Please enter " & _
                      "the phone number:", "Phone")

    strSQL = "usp_procEnterData " & param1 & ", " & param2

    If param1 <> "" And param2 <> "" Then

        conn.Execute (strSQL)

    End If

    ExitHere:
    If Not conn Is Nothing Then
```

```
If conn.State = adStateOpen Then conn.Close
End If
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub
```

**3.** Run the Execute\_StoredProcWithParam procedure.

When you run the parameterized stored procedure in Hands-On 16.5, Access displays an input box for each parameter. After you have supplied values for both required parameters, a new record is entered into the Shippers table.

## **DELETING A STORED PROCEDURE**

---

Use the `DROP PROCEDURE` (or `DROP PROC`) statement to delete a stored procedure. The syntax looks like this:

```
DROP PROC[EDURE] procedureName
```

The following example procedure deletes the stored procedure named `usp_procEnterData` from the current database.



### **Hands-On 16.6 Deleting a Stored Procedure**

1. Switch to the Visual Basic Editor window and insert a new module.
2. In the module's Code window, enter the `Delete_StoredProc` procedure shown here:

```
Sub Delete_StoredProc()
    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "DROP PROCEDURE usp_procEnterData; "
ExitHere:
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
Exit Sub
```

```
ErrorHandler:  
    If InStr(1, Err.Description, "cannot find") Then  
        MsgBox "The procedure you want to delete " & _  
            "does not exist.", _  
            vbDefaultButton1 + vbInformation, _  
            "Request failed"  
    Else  
        MsgBox Err.Number & ":" & Err.Description  
    End If  
    Resume ExitHere  
End Sub
```

3. Run the Delete\_StoredProc procedure to remove the usp\_procEnterData procedure from the database.

## CHANGING DATABASE RECORDS WITH STORED PROCEDURES

---

Stored procedures can perform various actions similar to what Access Action queries and Select queries with parameters can do. For example, the VBA procedure in Hands-On 16.7 creates a stored procedure that, when executed, deletes a record from a specific table. It is a recommended practice to always make a backup copy of your table before trying out statements that perform CRUD (Create, Update, Delete) operations, as incorrectly written statement may cause all records, instead of just one, to be deleted or updated.



### Hands-On 16.7 Create a Stored Procedure that Deletes a Record

1. Switch to the Visual Basic Editor window and insert a new module.
2. In the module's Code window, enter the **CreateProc\_DeleteRecordFromTable** procedure shown here:

```
Sub CreateProc_DeleteRecordFromTable(strTable As String)  
    Dim conn As ADODB.Connection  
  
    On Error GoTo ErrorHandler  
  
    Set conn = CurrentProject.Connection  
    conn.Execute "CREATE PROCEDURE usp_DeleteRec " & _  
        "(ID Integer) " & _  
        "AS " & _
```

```
"DELETE FROM " & strTable & " WHERE " & _
strTable & ".ID = ID;"  
  
Application.RefreshDatabaseWindow  
ExitHere:  
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
    Exit Sub
ErrorHandler:  
    If InStr(1, Err.Description, "already exists") Then
        conn.Execute "DROP PROCEDURE usp_DeleteRec; "
        Resume 0
    Else
        MsgBox Err.Number & ":" & Err.Description
    End If
    Resume ExitHere
End Sub
```

3. Enter a new record in the Employees and Shippers tables and note the IDs of the created records.
4. In the Visual Basic Editor window, activate Immediate Window by pressing **Ctrl+G** or choose **View | Immediate Window**. Type the following statement and press Enter:

```
CreateProc_DeleteRecordFromTable "Shippers"
```

The above statement calls the procedure and passes to it the name of the table from which you want to delete a record.

5. In the Navigation pane, double-click usp\_DeleteRec query and press Yes to confirm that you want to run this action query.
6. Enter the ID for the Shipper record you created in Step 3 and click **OK**.
7. Click **Yes**, to confirm the deletion of 1 record.
8. Open the Shippers table to verify that the specified record was deleted.
9. In the Immediate Window, execute the procedure again this time, passing it the Employees table.
10. Run the query usp\_DeleteRec to delete the Employee record you created in Step 3.

To update a phone number in a specified record in the Shippers table, you may want to create your own stored procedure that performs the specified record update with the following statement:

```
conn.Execute "CREATE PROCEDURE usp_UpdatePhone " & _
    "(ID Integer, tel text (25)) " & _
    "AS " & _
    "UPDATE Shippers SET Shippers.[Business Phone] = tel " & _
    "WHERE ID = Shippers.ID;"
```

## SUMMARY

---

This chapter introduced you to two powerful database objects you can use in Access: views and stored procedures. You learned how views are used as virtual tables to make specific rows and columns from one or more tables available to your Access users. Remember that views are similar to `SELECT` statements, except they cannot contain the `ORDER BY` clause to sort the data and they do not allow parameters. Views can be used in queries to hide from users the complexity of joins between the tables. Converting your Access queries into views and stored procedures will help with migration of your Access applications to the SQL Server environment in the future.

This chapter concludes Part III of this book, which presented numerous examples of using SQL DDL statements inside VBA procedures. In particular, you learned how DDL statements are used to create tables, views, stored procedures, primary keys, indexes, and constraints that define the database. You also learned some advanced Data Manipulation Language (DML) statements. Although there is more to Access SQL than this part of the book has covered, the information presented here should be quite sufficient to get you started using SQL in your own Access database applications.



Part

# IV

## *IMPLEMENTING DATABASE SECURITY*

This part of the book focuses on various methods of securing Access databases in the .ACCDB and .MDB file formats. Whether you are creating an Access database for yourself or others, security should never be taken lightly. Before deciding on any particular security model for your Access database familiarize yourself with various security features offered in both file formats. You can secure Access databases by setting a database password or implementing a Workgroup-based security.

Chapter 17 – Implementing Database Security with DDL

Chapter 18 – Implementing User-Level and Share-Level Security



The procedures in this chapter demonstrate how to use simple Data Definition Language (DDL) statements to easily manage database and user passwords, create or delete user and group accounts, and grant or delete permissions for database objects.

## TWO TYPES OF DATABASE SECURITY

---

Database security can be handled at *share-level* or *user-level*.

- *Share-level* security is the easiest to implement, as it only requires that you set a password on the database. With this security, users are required to enter the password before they can open and work with the database. The data in the database is encrypted so that it cannot be read with a text editor. The password security works well for small groups of individuals where it is not required to know who opened the database and which objects were accessed while working with the database. In share-level security, authorized users share the same password and have the same privileges. If the password gets into the hands of unauthorized people, the database can be compromised.

- *User-level* security grants permissions to the database objects based on a specific user and / or group. This type of security can only be used with Access databases created in the legacy (.MDB) file format. This security model is discussed in detail in Chapter 18.

## SETTING THE DATABASE PASSWORD

---

To set a new database password or change an existing password, use the `ALTER DATABASE PASSWORD` statement in the following format:

```
ALTER DATABASE PASSWORD newPassword oldPassword
```

When setting the password for the first time, use Null for the old password (see the example procedure in Hands-On 17.1). The Access database must be opened in exclusive mode to perform password operations. Therefore, when using ADO, set the ADO Connection object's Mode property to `adModeShareExclusive` before opening a database.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 17.1 Setting a Database Password

This hands-on exercise sets a database password on the Chap16.accdb database file you created in the previous chapter

1. Start Access and create a new database named **Chap17.accdb** in your **C:\VBAAccess2021\_ByExample** folder.
2. Switch to the Visual Basic Editor window and choose **Tools | References**. In the References dialog box, scroll down to locate **Microsoft ActiveX Data Objects 6.1 Library**. Click the checkbox to the left of this library name to set a reference to it and click **OK** to exit the dialog box.
3. Choose **Insert | Module** to add a new module to the current VBA project.
4. In the module's Code window, type the following **SetDBPassword** function:

```
Function SetDBPassword(strFullPath)
    Dim conn As ADODB.Connection
    On Error GoTo ErrorHandler
    Set conn = New ADODB.Connection
    With conn
        .Mode = adModeShareExclusive
        .Open "Provider = Microsoft.ACE.OLEDB.12.0;" & _
```

```
"Data Source=" & strFullFilePath & ";"  
.Execute "ALTER DATABASE PASSWORD secret null "  
End With  
ExitHere:  
If Not conn Is Nothing Then  
    If conn.State = adStateOpen Then conn.Close  
End If  
Set conn = Nothing  
Exit Function  
ErrorHandler:  
    MsgBox Err.Number & ":" & Err.Description  
    Resume ExitHere  
End Function
```

After opening a database in exclusive mode, this function procedure changes the database password from Null to “secret.” Notice that the new password is listed first, followed by the old password. Notice also how the function uses the State property of the ADO Connection object to determine whether the connection to the database is open. State returns `adStateOpen` if the Connection object is open and `adStateClosed` if it is not.

5. Execute the SetDBPassword function from the Immediate window by typing the following statement and pressing **Enter**:

```
SetDBPassword "C:\VBAAccess2021_ByExample\Chap16.accdb"
```

Activate the `C:\VBAAccess2021_ByExample\Chap16.accdb` database and see if you can open it by providing the password set in this Hands-On.

## REMOVING THE DATABASE PASSWORD

---

To remove a database password, replace the existing password with Null. The password can be removed by using the `ALTER DATABASE PASSWORD` statement, as illustrated in the preceding section. When the database is secured with a password, you will need to use the Jet/ACE OLEDB:Database Password property to specify the password to open the database. This is a Microsoft Jet 4.0/ACE OLE DB Provider-specific property of the Connection object. The following procedure shows how to remove the password “secret” from the `Chap16.accdb` database that was set by the `SetDBPassword` function in Hands-On 17.1.



### Hands-On 17.2 Deleting a Database Password

This procedure requires prior completion of Hands-On 17.1.

1. In the same module where you entered the SetDBPassword function (Hands-On 17.1), enter the **ResetDBPassword** function shown here:

```
Function ResetDBPassword(strFullPath, _
    strNewPwd, strOldPwd)

    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler
    Set conn = New ADODB.Connection

    With conn
        .Mode = adModeShareExclusive
        .Open "Provider = Microsoft.ACE.OLEDB.12.0;" & _
            "Data Source=" & strFullPath & _
            "; Jet OLEDB:Database Password=" & _
            strOldPwd & ";"
        .Execute "ALTER DATABASE PASSWORD " & _
            strNewPwd & " " & _
            strOldPwd
    End With
    ExitHere:
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
    Exit Function
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Function
```

2. Execute the ResetDBPassword function from the Immediate window by typing the following statement on one line and pressing **Enter**:

```
ResetDBPassword "C:\VBAAccess2021_ByExample\Chap16.accdb",
    "null", "secret"
```

3. Close the **Chap17.accdb** database file, saving the changes when prompted.

## **CREATING A USER ACCOUNT**

---

Establishing database security at a user level is more involved than setting a database password. As mentioned earlier, this type of security requires an

Access database in .mdb file format. To set up user level security you must create group and user accounts and assign permissions to groups and users to perform operations on various database objects. Use the `CREATE USER` statement to create a new user account. Specify the username to log in to the account followed by the required password and a personal identifier (PID) to make the account unique. The syntax of creating a user account looks like this:

```
CREATE USER userLoginName password PID
```

You can create more than one user account at a time by separating the usernames with a comma.

The procedure in Hands-On 17.3 sets up a new user account for GeorgeM with “fisherman” as the login password and “0302” as the PID. While this example procedure uses a simple PID number, the PID number you choose for a production environment should be from 4 to 20 characters long (preferably a combination of numbers and uppercase and lowercase letters that will be difficult for someone to guess).



### Hands-On 17.3 Creating a User Account

1. Create a new Access database named **Chap17.mdb** in your **C:\VBAAccess2021\_ByExample** folder. Be sure to select Microsoft Access databases (2002–2003) (\*.mdb) file format.
2. Switch to the Visual Basic Editor window and choose **Tools | References**. In the References dialog box, scroll down to locate **Microsoft ActiveX Data Objects 6.1 Library**. Click the checkbox to the left of this library name to set a reference to it and click **OK** to exit the dialog box.
3. Choose **Insert | Module** to add a new module to the current VBA project.
4. In the module’s Code window, enter the **CreateUserAccount** procedure shown here:

```
Sub CreateUserAccount()
    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "CREATE USER GeorgeM fisherman 0302"
ExitHere:
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
```

```
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

5. Run the CreateUserAccount procedure.
6. Press **Alt+F11** to switch to the main Access application window. Choose **File | Info | Users and Permissions | User and Group Accounts**. After running the CreateUserAccount procedure in Hands-On 17.3, you will see a listing for the GeorgeM user account in the User and Group Accounts window, as shown in Figure 17.1.



FIGURE 17.1. The User and Group Accounts window.

7. Click **Cancel** to close the User and Group Accounts window.

## CHANGING A USER PASSWORD

---

A user account password can be changed by using the `ALTER USER` statement in the following form:

```
ALTER USER userAccountName PASSWORD newPassword oldPassword
```

The procedure in Hands-On 17.4 changes the GeorgeM account's user password from "fisherman" to "primate."



### Hands-On 17.4 Changing a User Password

This hands-on exercise requires prior completion of Hands-On 17.3.

1. In the same module where you entered the CreateUserAccount procedure in Hands-On 17.3, enter the following **ChangeUserPassword** procedure as:

```
Sub ChangeUserPassword()
    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "ALTER USER GeorgeM PASSWORD " & _
                 "primate fisherman"

    ExitHere:
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Run the ChangeUserPassword procedure.

## CREATING A GROUP ACCOUNT

---

Use the **CREATE GROUP** statement to create a new group account. You must specify the group name followed by a unique PID (personal identifier):

```
CREATE GROUP groupName PID
```

You can create more than one group at a time by separating the group names with a comma. The procedure in Hands-On 17.5 creates a new group account called Mozart with “2021Best” as the PID.



### Hands-On 17.5 Creating a Group Account

1. In the same module where you entered the ChangeUserPassword procedure in Hands-On 17.4, enter the **CreateGroupAccount** procedure shown here:

```
Sub CreateGroupAccount()
    Dim conn As ADODB.Connection
```

```
On Error GoTo ErrorHandler

Set conn = CurrentProject.Connection

conn.Execute "CREATE GROUP Mozart 2021Best"
ExitHere:
If Not conn Is Nothing Then
    If conn.State = adStateOpen Then conn.Close
End If
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub
```

## 2. Run the CreateGroupAccount procedure.

The Groups tab in the User and Group Accounts window (see Figure 17.2) will now list the name of the newly created Mozart user group.

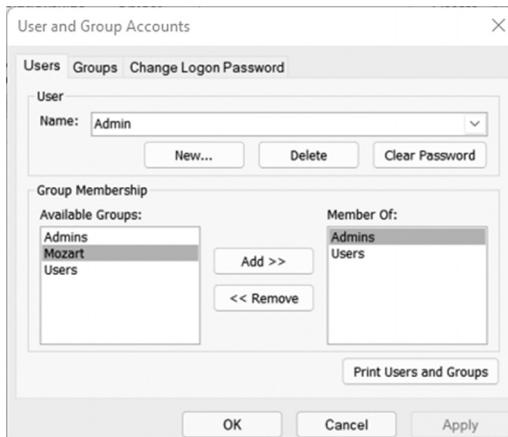


FIGURE 17.2. The User and Group Accounts window shows the Mozart group after running the CreateGroupAccount procedure in Hands-On 17.5.

## ADDING USERS TO GROUPS

Use the `ADD USER` statement to make a user account a member of a group. Specify the user account name followed by the `TO` keyword and a group name:

```
ADD USER userAccountName TO groupName
```



### Hands-On 17.6 Making a User Account a Member of a Group

This hands-on exercise requires prior completion of Hands-On 17.3 and 17.5.

1. In the same module where you entered the procedure in Hands-On 17.5, enter the following **AddUserToGroup** procedure:

```
Sub AddUserToGroup()
    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "ADD USER GeorgeM TO Mozart"

    ExitHere:
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Run the AddUserToGroup procedure.

The user account GeorgeM is now a member of the Mozart group account. This can be easily verified by opening the User and Group Accounts window in the Access application window (see Step 6 in Hands-On 17.3) and selecting GeorgeM from the Name drop-down.

## REMOVING A USER FROM A GROUP

---

To delete a user from a group, use the `DROP USER` statement followed by the username, the `FROM` keyword, and the group name. For example, to delete the GeorgeM account from the Mozart group, use the following statement:

```
DROP USER GeorgeM FROM Mozart
```



### Hands-On 17.7 Removing a User Account from a Group

This hands-on exercise requires prior completion of Hands-On 17.5 and 17.6.

1. In the same module where you entered the procedure in Hands-On 17.6, enter the **RemoveUserFromGroup** procedure shown here:

```
Sub RemoveUserFromGroup()
    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "DROP USER GeorgeM FROM Mozart"

    ExitHere:
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Run the RemoveUserFromGroup procedure to remove the GeorgeM user account from the Mozart group.

## DELETING A USER ACCOUNT

---

To delete a user account, use the `DROP USER` statement followed by the user account name, as demonstrated by the **DeleteUserAccount** procedure in Hands-On 17.8.



### Hands-On 17.8 Deleting a User Account

This procedure requires prior completion of Hands-On 17.3.

1. In the same module where you entered the procedures in the previous hands-on exercises, enter the following **DeleteUserAccount** procedure:

```
Sub DeleteUserAccount()
    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler
```

```

Set conn = CurrentProject.Connection

conn.Execute "DROP USER GeorgeM"
ExitHere:
If Not conn Is Nothing Then
    If conn.State = adStateOpen Then conn.Close
End If
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub

```

2. Run the DeleteUserAccount procedure to delete the user account named GeorgeM.

## GRANTING PERMISSIONS FOR AN OBJECT

---

Use the `GRANT` statement to assign security permissions for an object in a database to an existing user or group account. The procedure in Hands-On 17.9 grants the SELECT, DELETE, INSERT, and UPDATE permissions on all tables to the Mozart group.

The `GRANT` statement requires the following:

- A list of privileges to be granted
- The keyword `ON` followed by the name of a table, a non-table object, or an object container (e.g., Tables, Forms, Reports, Modules, Scripts)
- The keyword `TO` followed by the user or group name

```
GRANT listOfPermissions ON tableName | objectName |
    containerName TO accountName
```

Please note that in addition to tables, the Tables container contains queries, views, and procedures, and the Scripts container includes macros.

### Hands-On 17.9   Granting Permissions for Tables to an Existing Group

This hands-on exercise requires prior completion of Hands-On 17.5.

1. In the same module where you entered the procedure in Hands-On 17.8, enter the following `SetTblPermissions` procedure:

```

Sub SetTblPermissions()
    Dim conn As ADODB.Connection

```

```
On Error GoTo ErrorHandler

Set conn = CurrentProject.Connection

conn.Execute "GRANT SELECT, DELETE, INSERT, " _ 
& "UPDATE ON CONTAINER TABLES TO Mozart"
ExitHere:
If Not conn Is Nothing Then
    If conn.State = adStateOpen Then conn.Close
End If
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Number & ":" & Err.Description
Resume ExitHere
End Sub
```

## 2. Run the SetTblPermissions procedure.

After running the SetTblPermissions procedure, you can open the User and Group Permissions window (choose **File | Info | Users and Permissions | User and Group Permissions**) to check out the privileges granted to the members of the Mozart group, as shown in Figure 17.3.

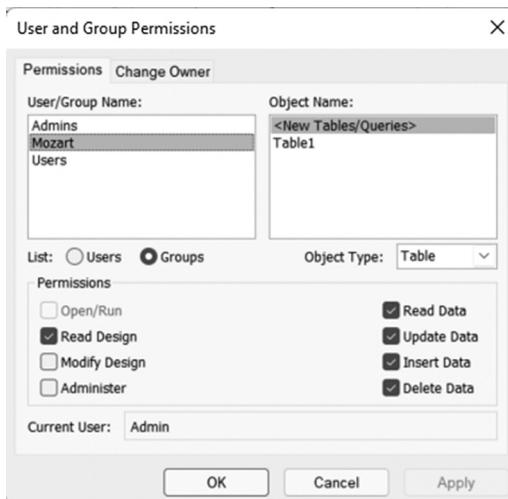


FIGURE 17.3 Verifying the group permissions to database objects.

## REVOKING SECURITY PERMISSIONS

---

Use the REVOKE statement to revoke security permissions for an object from an existing user or group account. This statement has the following form:

```
REVOKE listOfPermissions ON tableName | objectName | container-
Name FROM accountName
```

The procedure in Hands-On 17.10 removes the privilege of deleting tables from the members of the Mozart group (see Figure 17.4).



### Hands-On 17.10 Revoking Security Permissions

This hands-on exercise requires prior completion of Hands-On 17.5 and 17.9.

1. In the same module where you entered the procedure in Hands-On 17.9, enter the **RevokePermission** procedure shown here:

```
Sub RevokePermission()
    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "REVOKE DELETE ON CONTAINER TABLES FROM Mozart"
ExitHere:
    If Not conn Is Nothing Then
        If conn.State = adStateOpen Then conn.Close
    End If
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
    Resume ExitHere
End Sub
```

2. Run the RevokePermission procedure.

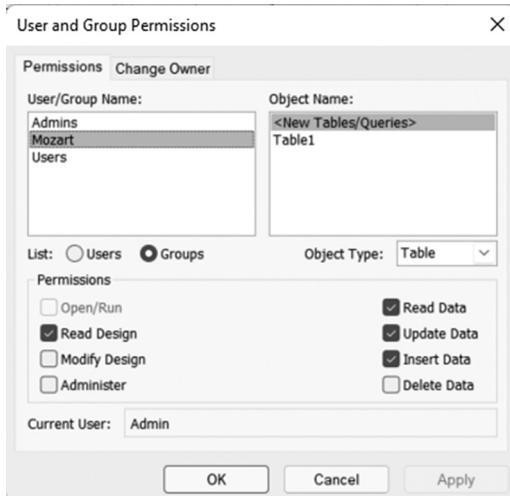


FIGURE 17.4 After running the procedure in Hands-On 17.10, the Delete Data permission on new tables and queries for the members of the Mozart group is turned off.

## DELETING A GROUP ACCOUNT

Use the `DROP GROUP` statement to delete a group account. You only need to specify the name of the group account you want to delete. To delete more than one account, separate each group name with a comma.



### Hands-On 17.11 Deleting a Group Account

This hands-on exercise requires prior completion of Hands-On 17.5.

1. In the same module where you entered the procedure in Hands-On 17.9, enter the following `DeleteGroupAccount` procedure:

```
Sub DeleteGroupAccount()
    Dim conn As ADODB.Connection

    On Error GoTo ErrorHandler

    Set conn = CurrentProject.Connection

    conn.Execute "DROP GROUP Mozart"
```

```
ExitHere:  
    If Not conn Is Nothing Then  
        If conn.State = adStateOpen Then conn.Close  
    End If  
    Set conn = Nothing  
    Exit Sub  
ErrorHandler:  
    MsgBox Err.Number & ":" & Err.Description  
    Resume ExitHere  
End Sub
```

2. Run the DeleteGroupAccount procedure to delete the Mozart group account.
3. Close the Chap17.mdb database, saving the changes when prompted.

## SUMMARY

---

This chapter demonstrated the use of Data Definition Language (DDL) statements in VBA procedures for managing security in an Access database. You used the `ALTER DATABASE PASSWORD` statement to create, modify, and remove the database password. You managed user-level accounts with the `CREATE`, `ADD`, `ALTER`, and `DROP` statements. You also learned how to use the `GRANT` and `REVOKE` statements to establish and remove permissions on database objects for user and group accounts in an Access MDB database created in the 2002–2003 file format.

In the next chapter, you will learn how to implement user-level and share-level security in Access databases.



# Chapter 18 *IMPLEMENTING USER-LEVEL AND SHARE-LEVEL SECURITY*

**A**s you learned in the previous chapter, the Access .accdb file format does not support user-level security. This means that you cannot create user and group accounts or assign object permissions in these types of databases. While you already know how to create Users and Groups in .mdb-type databases using the DDL statements, this chapter brings more Access features to your skillset.

You will learn how to:

- Use the Users and Groups collections of the ADOX Catalog object to create and manage security user accounts.
- Use the `GetPermissions` and `SetPermissions` methods of the ADOX User and Group objects to retrieve and set permissions on database objects.
- Use the `ChangePassword` method of the ADOX User object to change the user's password.
- Use the `CompactDatabase` method of the DAO DbEngine object to compact database and set database password.

<b><u>NOTE</u></b>	<p><i>To use ADOX in your VBA procedures, you must set a reference to the Microsoft ADO Ext. 6.0 for DDL and Security Object Library Library (choose Tools   References in the Visual Basic Editor window to open the References dialog box).</i></p>
--------------------	---

## SHARE-LEVEL SECURITY

---

Using passwords to secure the database or objects in the database is known as *share-level security*. When you set a password on the database, users are required to enter a password in order to gain access to the data and database objects. Anyone with the password has unrestricted access to all Access data and database objects.

To manually change the database password:

- For an Access database in the .accdb file format, choose File | Info | Encrypt with Password.
- For an Access database in the .mdb file format, choose File | Info | Set Database Password.

<b><u>NOTE</u></b>	<p><i>Refer to the sections titled “Setting a Database Password Using the CompactDatabase Method” and “Setting a Database Password Using the NewPassword Method” later in this chapter to set a database password from within a VBA procedure.</i></p>
--------------------	--

## USER-LEVEL SECURITY

---

*User-level security* is a relatively complex process that secures the code and objects in your database so that users can't accidentally modify or change them. With this type of security you can provide the most restrictive access to the database and the objects it contains. When you use user-level security, a *workgroup information file* is used to determine who can open a database and what objects are available to them.

The workgroup information file holds group and user information, including passwords. The information contained in this file determines not only who can open the database, but also the permissions users and groups have on the objects in the database. The workgroup information file contains built-in groups (Admins and Users) and a generic user account (Admin) with unlimited privileges on the database and the objects it contains.

When an Access .mdb database file is open in Access 2021, you can manually implement user-level security by choosing File | Info | Users and Permissions. You can also define user and group accounts and their passwords from your VBA procedures as demonstrated later in this chapter.

## UNDERSTANDING WORKGROUP INFORMATION FILES

---

To successfully run the procedures in this chapter, you need to know the location of the workgroup information file on your computer. This file, also known as *system database* (System.mdw), is created automatically on your computer when you create a .mdb format database (see Table 18.1). This file, which can also be named System1.mdw, System2.mdw and so on, is stored in the AppData folder on your main system drive (C:\). This is a hidden folder, so to browse it you must first enable hidden files:

- In Windows 7: activate Windows Explorer and choose Tools | Folder Options. In the Folder Options window, click the View tab, click the option button next to Show hidden files and folders, and click OK.
- In Windows 8 and 10: activate File Explorer and click the View tab. Check the Hidden Items in the View/Hide section of the ribbon.
- In Windows 11: activate File Explorer and choose View | Show | Hidden Items.

Now you should be able to access the path, where <username> is the name of your user profile. Take a few minutes right now to locate the System.mdw file on your machine using the Table 18.1.

TABLE 18.1 The workgroup information file in different versions of Access

Access Version	Default Workgroup Information Filename	Workgroup Information File Location
2000	System.mdw	C:\Program Files\Common Files\System
2002–2003	System.mdw	C:\Documents and Settings\<username>\Application Data\Microsoft\Access
2007–2010	System.mdw	C:\Users\<username>\AppData\Roaming\Microsoft\Access\System.mdw
2013–2021	System.mdw	C:\Users\<username>\AppData\Roaming\Microsoft\Access\System.mdw

You can also find the location and name of the workgroup information file by starting Microsoft Access and opening any MDB database. Try it with the Chap17.mdb database you created in Chapter 17. Switch to the Visual Basic Editor window and activate the Immediate window. Type the following statement on one line (beginning with a question mark) and press Enter to execute:

```
? CurrentProject.Connection.Properties(  
    "Jet OLEDB:System Database").Value
```

When you press Enter, Access displays the full path of the workgroup information file that the currently open database uses for its security information. Jet OLEDB:System Database is a provider-specific property of the Microsoft OLE DB Provider for Jet in the ADO Properties collection of the Connection object.

Access uses the workgroup information file to store the following information:

- The name of each user and group
- The list of users who belong to each group
- The encrypted logon password for each workgroup user
- The Security Identifier (SID) of each user and group in a binary format

Once you add user and group accounts to your database, the workgroup information file will contain vital security information. *YOU DON'T WANT TO LOSE THIS INFORMATION.* Always take time to make a backup copy of the system file and store it in a safe location. This way, if the original file gets corrupted, you'll be able to quickly restore your backup file and avoid having to recreate user and group accounts.

The workgroup information file is like any other Access database file except that it contains hidden system tables with information regarding user and group accounts and their actual permissions. However, you cannot change the security information by opening this file directly. All the security data stored in hidden system tables is encrypted and protected. Changes to the workgroup information file are done automatically by the JetEngine when you use the built-in Access commands to manage security or execute VBA code.

You can use the same workgroup information file for more than one database or you can create a separate workgroup information file for each database you are securing. You can also give this file a name other than the default System.mdw. Most people find it best to use the same name as the database file. For example, if your secured database file is named Assets.mdb, you could create a workgroup information file called Assets.mdw and put it in the same folder as the database file. This way, you'd know right away that these two files are associated with one another even after many weeks or months have passed since

you created them. Keeping track of which workgroup information file goes with which database can be quite challenging, especially if you are managing more than a couple of secured Access databases.

<b>NOTE</b>	<p><i>If you try to open a secured database while another workgroup information file is active, Access displays the following message:</i></p> <p><i>You do not have the necessary permissions to use the &lt;name&gt; object. Have your system administrator or the person who created this object establish the appropriate permissions for you.</i></p> <p><i>If you receive the preceding message while opening an Access database in the .mdb file format, you should look for the accompanying workgroup information file and perform one of the following:</i></p> <ul style="list-style-type: none"><li>● <b>Set Up a Shortcut</b></li></ul> <p><i>Set up a shortcut to the database file that uses the /WRKGRP command-line switch to load the specified workgroup information file when the database is opened (see Custom Project 18.1).</i></p> <ul style="list-style-type: none"><li>● <b>Use the Workgroup Administrator Tool in Access 2021</b></li></ul> <ol style="list-style-type: none"><li>1. Start Access and open any Access database.</li><li>2. Activate the Visual Basic Editor window.</li><li>3. Choose <b>View   Immediate Window</b>.</li><li>4. In the Immediate window, type the following statement and press <b>Enter</b> to execute: <code>DoCmd.RunCommand acCmdWorkgroupAdministrator</code></li><li>5. In the Workgroup Administrator dialog box, click <b>Join</b>, then click <b>Browse</b>.</li><li>6. Locate the workgroup information file and then click <b>Open</b>. See Table 18.1 for the .mdw filenames used with various versions of Access.</li><li>7. In the Workgroup Administrator dialog box, click <b>OK</b>, then click <b>Exit</b>.</li></ol>
-------------	---

### **Creating and Joining Workgroup Information Files**

When you open a database, Access reads the workgroup information file to find out who is allowed to access the database. If security was put into place, you will be prompted for the user ID and password. Custom Project 18.1 walks you

through the steps required to create and join a new workgroup information file. Once you join the workgroup, you create a new Access database and set up a password for the Admin user. This information is saved in the workgroup information file that you just joined. The workgroup information file is created using the User-Level Security Wizard. This option is available by choosing File | Info | Users and Permissions.

Securing a database boils down to creating a new workgroup information file, adding a new member to the Admins group, and removing the default Admin user from that group. You also need to remove permissions from the Admin user and from the Users group, and assign permissions to your own groups that you create. Don't be discouraged if you need to go over the security steps more than once. Access security is complex and can be approached from many different angles. Books of several hundred pages have been written to explain its inner workings. The approach presented here simply provides us with a secured Access database file we use to perform the programming exercises in this chapter. Although you could learn how to use the ADOX commands for managing security using the currently open unsecured Access database, this particular approach gives you a better set of skills to build from. So let's begin.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*

**Custom Project 18.1 Securing an Access MDB Database**

You must complete this project in order to work with the hands-on exercises in this chapter.

1. Start Access and create a new blank database called **SpecialDb.mdb** in your **C:\VBAAccess2021\_ByExample** folder. Be sure to select **Microsoft Access Databases (2002–2003) (\*.mdb)** file format.

In the next step, you will use the built-in User-Level Security Wizard to secure the blank Access database (**SpecialDb.mdb**) you just created.

2. Choose **File | Info | Users and Permissions | User-Level Security Wizard**.
3. Click **Yes** in response to the message that the database should be opened in shared mode to run the Security Wizard.

Access closes the database and reopens it in shared mode.

4. Access automatically activates the Security Wizard (see Figure 18.1). Click **Next** to continue.

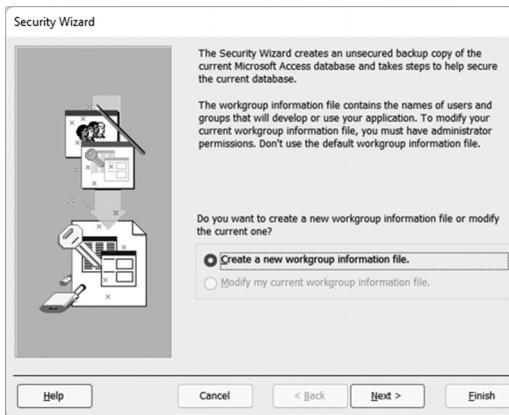


FIGURE 18.1. Security Wizard (screen 1).

5. Another Security Wizard window appears (see Figure 18.2). Do not make any changes in this screen. Click **Next** to continue.

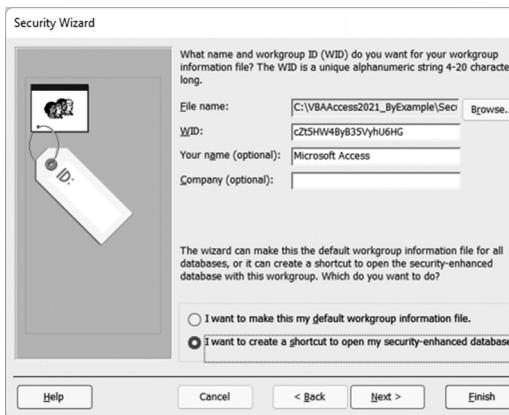


FIGURE 18.2. Security Wizard (screen 2). The workgroup information file named Security.mdb stores user and group account information for the SpecialDb database.

6. The Security Wizard window now shows an empty tabbed screen that normally displays database objects (Figure 18.3). Because our database does not contain any tables, queries, reports, etc., there's nothing you can do in this screen. Click **Next** to continue.

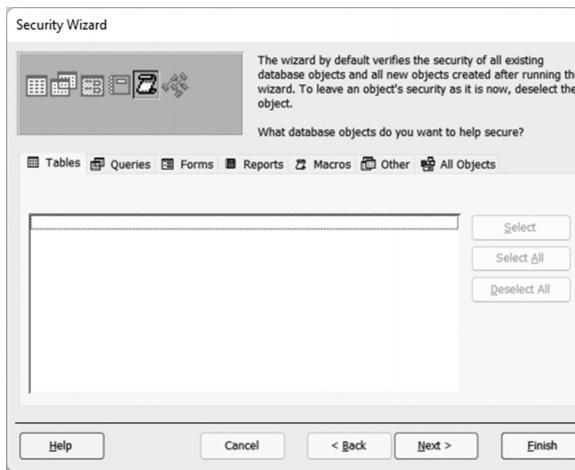


FIGURE 18.3. Security Wizard (screen 3).

7. The Security Wizard window now displays a list of optional security accounts that you could include in your new workgroup information file (Figure 18.4). Because we will define our accounts in programming code later in this chapter, do not make any selections in this screen. Click **Next** to continue to the next screen.

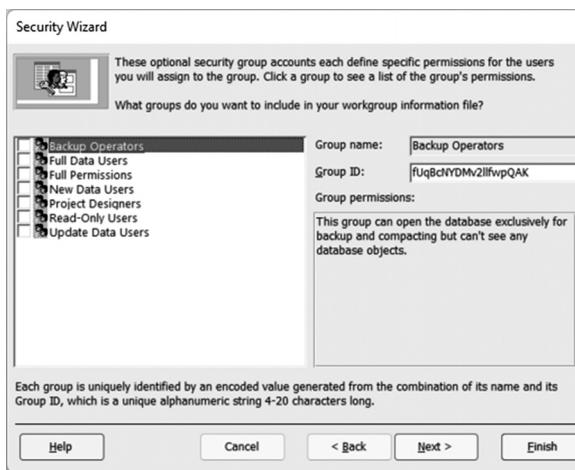


FIGURE 18.4 Security Wizard (screen 4).

8. Now the Security Wizard asks whether you want to grant permissions to the Users group (Figure 18.5). The Users group will have no permissions, so do

not make any changes in this screen. We will work with permissions in our VBA procedures later. Click **Next** to continue.

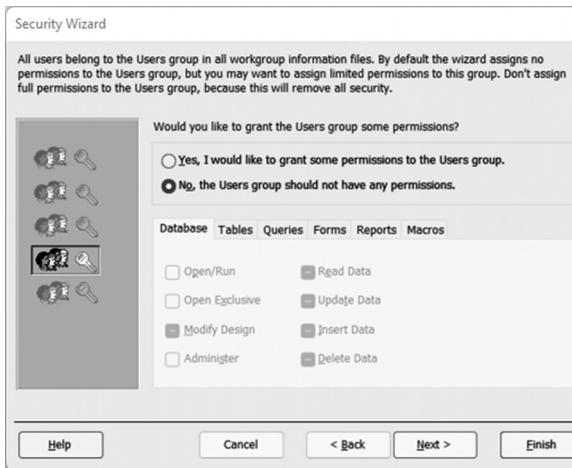


FIGURE 18.5. Security Wizard (screen 5).

- Now the Security Wizard shows a screen (Figure 18.6) where you finally can do a little bit of work. You need to define a new user in your database. This user will function as a new Admin. Let's call this user **Developer** and allow him to log into the database using **chapter18** as a password. Fill in the User name and the Password boxes as shown in Figure 18.6 and click the **Add This User to the List** button. **Developer** should now appear in the users list (see Figure 18.7). Do not leave this screen yet.

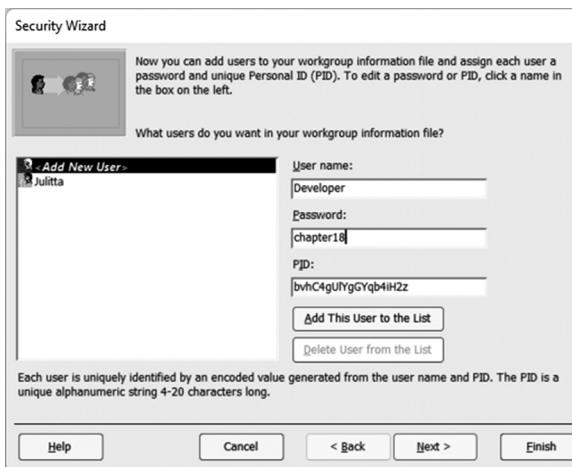


FIGURE 18.6. Security Wizard (screen 6a).

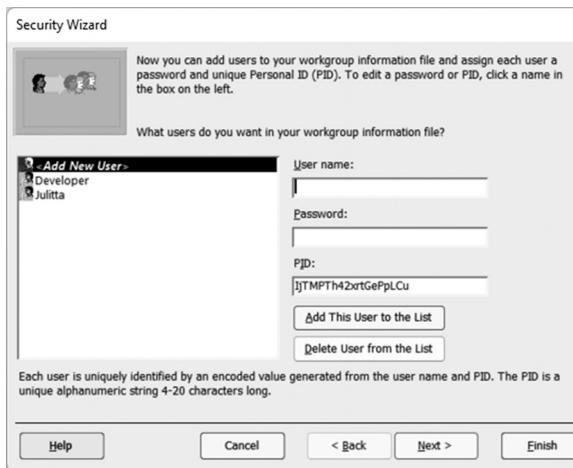


FIGURE 18.7. Security Wizard (screen 6b).

10. Now remove the user account you used to log into Access. In the list of users, select the username you logged in with and click the **Delete User from the List** button. Now **Developer** is the only user in our database, as shown in Figure 18.8. Click **Next** to continue.

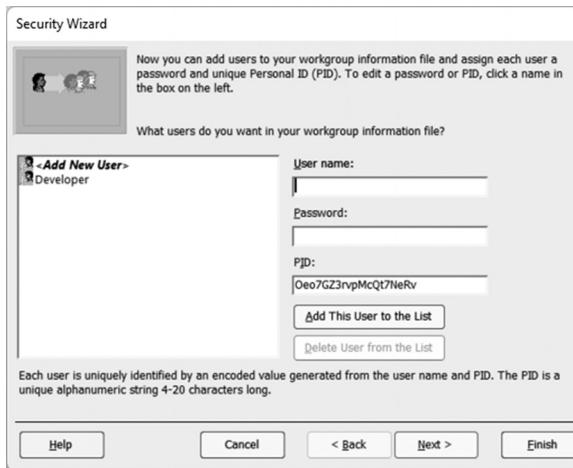


FIGURE 18.8. Security Wizard (screen 6c).

11. The Security Wizard shows the screen where you can assign users to groups in the workgroup information file (Figure 18.9). Notice that the user (**Developer**) you created in Step 8 is a member of the Admins group. Click **Next** to continue.

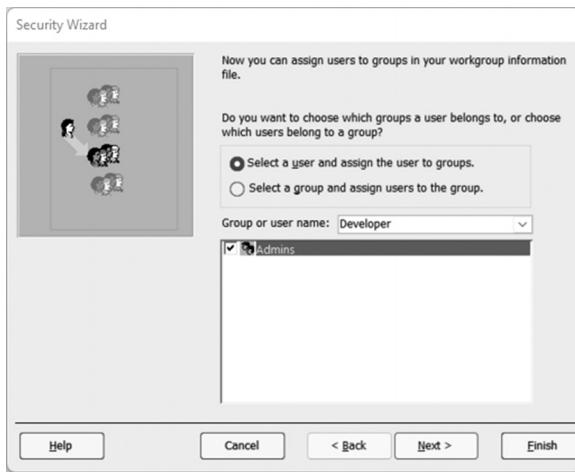


FIGURE 18.9. Security Wizard (screen 7).

12. The Security Wizard has now collected all the required information as shown in Figure 19.10. Click **Finish**.

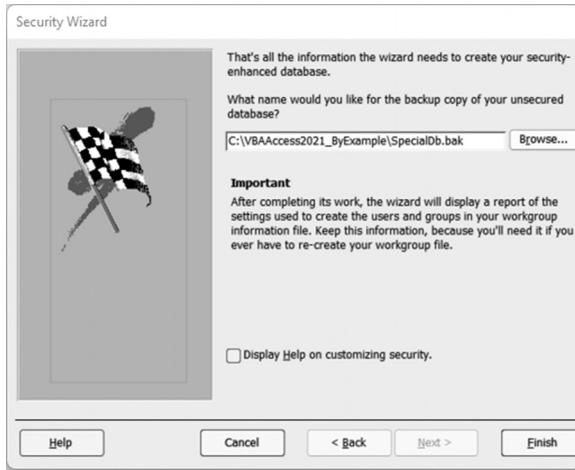


FIGURE 18.10. Security Wizard (screen 8).

13. Access performs its final tasks of securing your database and displays the Security Wizard report (Figure 18.11). If you are connected to a printer, it's a good idea to take a minute now to print this report or save it to the PDF file. You can also magnify the report to read it on screen. When you are done, close the Security Wizard report window.

14. When you close the Print Preview window, the Security Wizard displays a warning message that notifies you that you must have the information contained in this report to recreate your workbook file if your original workbook file is lost or corrupted. Access also asks you whether you would like to save the report as a Snapshot (.snp) file that you can view later. Click Yes. You should see the confirmation message that the Security Wizard has encoded your database and to reopen the database, you must use the new workgroup file you created by closing Access and reopening it. You'll do as suggested in the next section. Click OK to this message.
15. Close the main Access window.

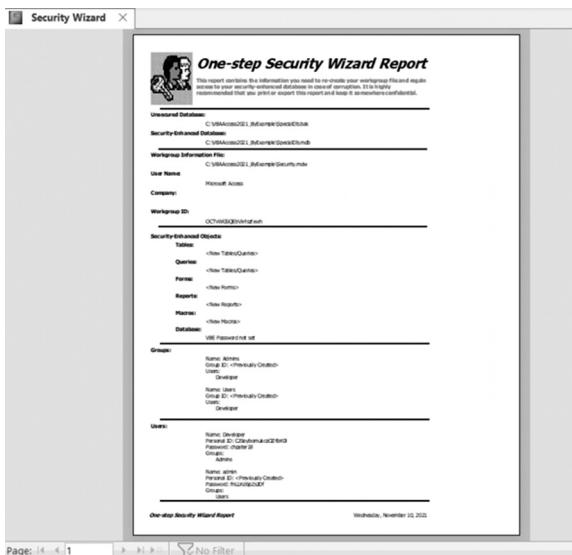


FIGURE 18.11 Security Wizard (screen 8).

## OPENING A SECURED MDB DATABASE

The following four files were added to your **C:\VBAAccess2021\_ByExample** folder when you completed Custom Project 18.1:

- A database file named *SpecialDb.mdb*
- A workgroup information file named *Security.mdw* that stores user and group account information for the SpecialDb database
- A snapshot file named *SpecialDb.snp*
- A backup copy of the SpecialDb database named *SpecialDb.bak*

Also, there is a shortcut on your desktop (created by the Security Wizard) that allows you to quickly start the SpecialDb database using the new workgroup information file (Security.mdw). If you right-click that desktop shortcut and choose Properties, you will see in the Target box the following path:

```
"C:\Program Files\Microsoft Office\root\Office16\MSACCESS.EXE"  
"C:\VBAAccess2021_ByExample\SpecialDb.mdb" /WRKGRP "C:\VBAAC-  
cess2021_ByExample\Security.mdw"
```

Because this path is very long it's shown here on three lines. Notice that the first part of this path is the location of the Access executable program on your computer enclosed by quotation marks. The path to the MSAccess.exe file is followed by a space and the full path of the database file (also in quotation marks). Because this database file is secured, we must also include a space and a command-line switch, /WRKGRP, followed by a space and the name of the accompanying workgroup information file (also In quotation marks).

The /WRKGRP command-line switch tells Access that you want to start a database with a specific workgroup. If you know which user account you want to log on with, you can use the /User and /Pwd command-line switches to avoid being prompted by Access for the username and password:

```
"C:\Program Files\Microsoft Office\root\Office16\MSACCESS.EXE"  
"C:\VBAAccess2021_ByExample\SpecialDb.mdb" /WRKGRP "C:\VBAAC-  
cess2021_ByExample\Security.mdw"  
/User "Developer" /Pwd "chapter18"
```

The information about the username and password follows the name of the workgroup information file and a single space.

Now that you know how the path to a secured database is built, you can create similar shortcuts to other secured databases if they use different workgroup information files.



### Hands-On 18.1 Opening a Secured MDB Database

This hands-on exercise requires prior completion of Custom Project 18.1.

1. On your desktop, double-click the shortcut to **SpecialDb.mdb** to open the database. Because this database is protected, a logon box appears. Enter **Developer** in the Name box and **chapter18** in the Password box and click **OK**. If the password does not work, open the saved or printed Security Wizard report, and check your password that is stored in the Users section. In my case, I misspelled the password **chapter18** as **chpater18**, so I could not open the database until I entered the password that matched the one stored by the

Security Wizard. Once you open the database, you can change the Logon Password for the user by accessing the User and Group Accounts (File | Info | Users and Permissions | User and Group Accounts).

- Now that your secured database file is open, let's take a look at the changes the Security Wizard has made in the Users and Groups accounts. Choose **File | Info | Users and Permissions | User and Group Accounts**. Notice that the Admin user is a member of the Users group (see Figure 18.12). The Security Wizard removed the Admin account from the Admins group. If you open the Name drop-down list in the User area of this screen and select Developer, you will see that Developer is a member of two groups: Admins and Users. Click **Cancel** to exit the User and Group Accounts window.

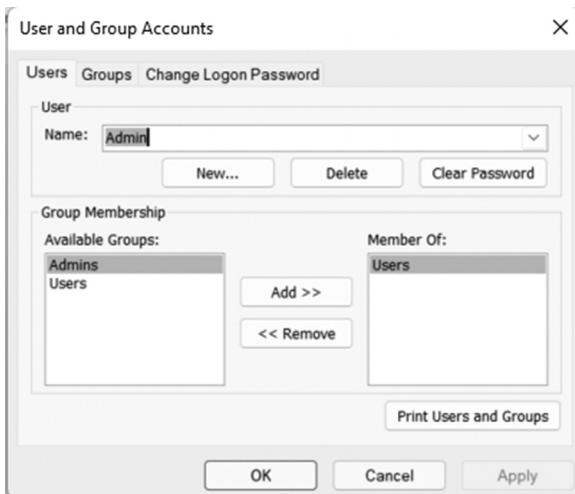


FIGURE 18.12 In Custom Project 18.1, you removed the default Admin user from the Admins group while running the built-in User-Level Security Wizard.

- Having checked the Users and Groups accounts, you can also examine the changes made by the Security Wizard in the group permissions. Choose **File | Info | Users and Permissions | User and Group Permissions**. Right now, the users Developer and Admin don't have permissions on any new objects (see Figure 18.13). To view group permissions, click the **Groups** option button. The Admins group has all the necessary permissions to administer the database while the Users group has no permissions at all. You will learn how to grant and revoke permissions to database objects in the example procedures in this chapter. Now click **Cancel** to exit the User and Group Permissions window.

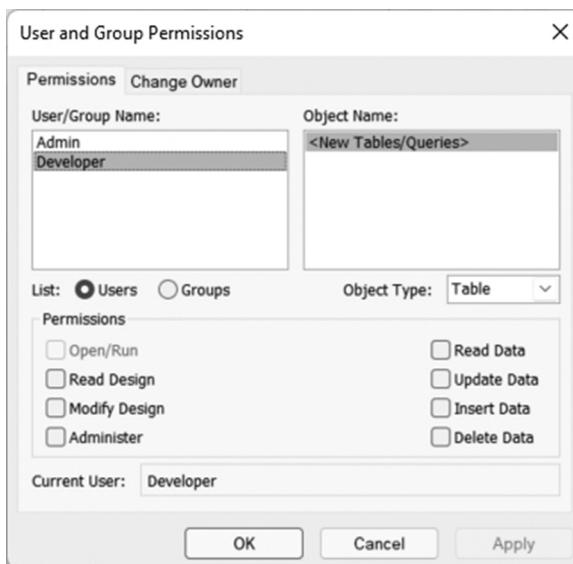


FIGURE 18.13. Use the User and Group Permissions window to check current permissions for the users Admin and Developer after running the User-Level Security Wizard in Custom Project 18.1.

4. Now let's import a couple of objects into this database. We will need them for our tests later in this chapter when we learn to handle permissions for database objects. In the Access window, choose **External Data | New Data Source | From Database | Access**. In the Get External Data dialog box, enter C:\VBAAccess2021\_ByExample\ Northwind.mdb in the File name box and click **OK**.
5. In the Import Objects window, click **Select All** to select all the tables. Click the **Queries** tab, then choose **Select All** to select all the queries. Finally, click **OK** to begin importing. When the import operation is completed, click the **Close** button.
6. The objects you selected in Step 5 have now been added to your database. Close the SpecialDb database and exit Access.

## **CREATING AND MANAGING GROUP AND USER ACCOUNTS**

To create a new group account from a VBA procedure using ADO, open the ADOX Catalog object by specifying the connection to the appropriate database and use the `Append` method of the Catalog object's Groups collection to add a new group account.

To create a new user account, pass the name and password to the `Append` method of the `Users` collection. Specifying a password at this time is optional. You can assign a password later with the `User` object's `ChangePassword` method.

The procedure in Hands-On 18.2 illustrates how to create two group accounts and a user account in the secured database (`SpecialDb.mdb`) that you created in Custom Project 18.1.



## Hands-On 18.2 Creating User and Group Accounts (ADO)

This hands-on exercise requires that you have completed Custom Project 18.1.

1. Start Access and create a new database named **Chap18.accdb** in your **C:\VBAAccess2021\_ByExample** folder.
2. Switch to the Visual Basic Editor window and choose **Insert | Module**.
3. Choose **Tools | References** and click the checkbox next to the following three object libraries:
  - **Microsoft ActiveX Data Objects 6.1 Library**,
  - **Microsoft ADO Ext. 6.0 for DDL and Security Object Library**,

After making these selections, click **OK** to exit the References dialog box.

4. Activate the Immediate window by choosing **View | Immediate Window**. Type the following statement in the Immediate window and press **Enter**:

```
DoCmd.RunCommand acCmdWorkgroupAdministrator
```

When you press Enter, Access loads the Workgroup Administrator tool, which lets you check the path to the workgroup information file that is currently being used. In Access 2021, there is no command in the user interface to access this tool. You must enter the preceding code in the Immediate Window of the Access database in the .mdb file format to use the Workgroup Administrator tool.

***Perform one of the following steps:***

- a. If `System1.mdw` appears in the Workgroup path, click **OK** to exit the Workgroup Administrator dialog box and proceed with Step 5.
- b. If the Workgroup path includes the `Security.mdw` file that was created in Custom Project 18.1, click the **Join** button to join another workgroup. Use the **Browse** button in the Workgroup Information File dialog box to select and open `System.mdw`. Refer to the beginning of this chapter for information on the default location of this file. Once you select the correct file, the dialog box should display its full path. Click **OK** to exit this dialog box. Access will display a message box saying you successfully joined the work-

group defined by the selected information file. Click **OK** to the message and click **OK** in the Workgroup Administrator dialog box to exit. Proceed to Step 5.

5. In the module's Code window, enter the following **Create\_UserAndGroup\_ADO** procedure:

```
Sub Create_UserAndGroup_ADO()
    Dim cat As ADOX.Catalog
    Dim conn As ADODB.Connection
    Dim strPath As String
    Dim strDB As String
    Dim strSysDB As String
    Dim strGrpName1 As String
    Dim strGrpName2 As String
    Dim strUsrName As String

    On Error GoTo ErrorHandler

    strPath = "C:\VBAAccess2021_ByExample\
    strDB = "SpecialDb.mdb"
    strSysDB = "Security.mdw"
    strGrpName1 = "Masters"
    strGrpName2 = "Elite"
    strUsrName = "PowerUser"
    ' open connection to the database
    ' using the specified system database
    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Properties("Jet OLEDB:System Database") = _
            strPath & strSysDB
        .Properties("User ID") = "Developer"
        .Properties("Password") = "chapter18"
        .Open strPath & strDB
    End With

    ' Open the catalog
    Set cat = New ADOX.Catalog
    With cat
        .ActiveConnection = conn
        ' create group accounts
        .Groups.Append strGrpName1
        .Groups.Append strGrpName2
        Debug.Print "Created group accounts."
        ' create a user account
        .Users.Append strUsrName, "star"
    End With
End Sub
```

```
Debug.Print "Created user account."
' Add user to the group
.Users(strUsrName).Groups.Append strGrpName2
Debug.Print strUsrName &
" is a member of the " &
strGrpName2 & " group account."
End With

ExitHere:
Set cat = Nothing
conn.Close
Set conn = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Description
Resume ExitHere
End Sub
```

6. Choose **Run | Run Sub/UserForm** to execute the `Create_UserAndGroup_ADO` procedure.

Upon executing this procedure, two new group accounts named Masters and Elite are established in the secured SpecialDb database you created in Custom Project 18.1. A new user account named PowerUser is added and made a member of the Elite group account. Notice that before opening the database we need to set the Jet OLEDB:System Database property in the Properties collection of the ADO Connection object to specify the path and name of the workgroup information file that should be active when the database is opened. We also set the User ID and Password properties to log onto the database. After opening the database, we open the Catalog object and use the `Append` method of the Catalog's Groups collection to add new group accounts. The Groups collection contains all groups in the specified workgroup information file. The `Append` method of the Catalog's Users collection is used to create a new user account. This user account is then appended to the Groups collection and made a member of a group (Elite).

7. If you'd like to take a moment now, open the SpecialDb database using the shortcut on your desktop. Once the database is open, choose **File | Info | Users and Permissions | User and Group Accounts**. Notice that the database now contains the Masters and Elite groups in addition to the default Admins and Users groups (see Figure 18.14).



FIGURE 18.14. The Elite and Masters group accounts are created by running the procedure in Hands-On 18.2.

8. Close the SpecialDb database and the Access window in which it was displayed. Be careful not to close the Chap18.accdb database you are working with.

### **Deleting User and Group Accounts**

Use the `Delete` method of the Catalog object's `Users` collection to delete a user account. Use the `Delete` method of the Catalog object's `Groups` collection to delete a group account.

The procedure in Hands-On 18.3 deletes the user account named PowerUser and the group account named Masters that were created in Hands-On 18.2.



### **Hands-On 18.3 Deleting User and Group Accounts (ADO)**

This hands-on exercise requires the prior completion of Custom Project 18.1 and Hands-On 18.2.

1. In the Visual Basic Editor window of Chap18.accdb, choose **Insert | Module**.
2. In the module's Code window, enter the following `Delete_UserAndGroup` procedure:

```
Sub Delete_UserAndGroup(UserName As String, _
    GroupName As String)
    Dim cat As ADOX.Catalog
    Dim conn As ADODB.Connection
    Dim strPath As String
```

```
Dim strDB As String
Dim strSysDB As String

On Error GoTo ErrorHandler

strPath = "C:\VBAAccess2021_ByExample\
strDB = "SpecialDb.mdb"
strSysDB = "Security.mdw"

' Open connection to the database using
' the specified system database
Set conn = New ADODB.Connection
With conn
    .Provider = "Microsoft.Jet.OLEDB.4.0"
    .Properties("Jet OLEDB:System Database") = _
        strPath & strSysDB
    .Properties("User ID") = "Developer"
    .Properties("Password") = "chapter17"
    .Open strPath & strDB
End With

' Open the catalog
Set cat = New ADOX.Catalog
With cat
    .ActiveConnection = conn
    ' Delete user
    .Users.Delete UserName
    ' Delete group
    .Groups.Delete GroupName
End With

ExitHere:
    Set cat = Nothing
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Description
    Resume ExitHere
End Sub
```

3. To run this procedure, enter the following statement in the Immediate window and press **Enter** to execute it:

```
Delete_UserAndGroup "PowerUser", "Masters"
```

After running the Delete\_UserAndGroup procedure, the Masters group account and the PowerUser user account are removed from the SpecialDb database.

### **Listing User and Group Accounts**

---

The procedure in Hands-On 18.4 demonstrates how to retrieve the names of all defined group and user accounts from the Groups and Users collections of the Catalog object (see Figure 18.15).



#### **Hands-On 18.4 Listing Group and User Accounts (ADO)**

1. In the Visual Basic Editor window of Chap18.accdb, choose **Insert | Module**.
2. In the module's Code window, enter the **List\_GroupsAndUsers\_ADO** procedure as shown here:

```
Sub List_GroupsAndUsers_ADO()
    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog
    Dim grp As New ADOX.Group
    Dim usr As New ADOX.User
    Dim strPath As String
    Dim strDB As String
    Dim strSysDB As String

    strPath = "C:\VBAAccess2021_ByExample\""
    strDB = "SpecialDb.mdb"
    strSysDB = "Security.mdw"

    ' Open connection to the database using
    ' the specified system database
    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Properties("Jet OLEDB:System Database") = _
            strPath & strSysDB
        .Properties("User ID") = "Developer"
        .Properties("Password") = "chapter18"
        .Open strPath & strDB
    End With

    ' Open the catalog
    Set cat = New ADOX.Catalog
```

```
cat.ActiveConnection = conn
' list group and user accounts
For Each grp In cat.Groups
    Debug.Print "Group: " & grp.Name
Next

For Each usr In cat.Users
    Debug.Print "User: " & usr.Name
Next

Set cat = Nothing
conn.Close
Set conn = Nothing

MsgBox "Groups and users are " & _
    "listed in the Immediate window."
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.  
The procedure result is shown in Figure 18.15.

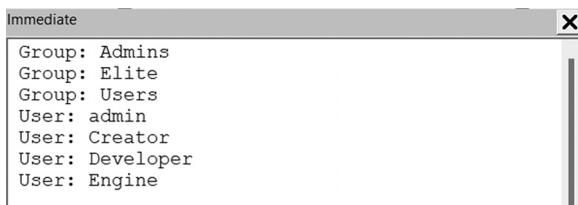


FIGURE 18.15 The names of existing security group and user accounts are written to the Immediate window by the procedure in Hands-On 18.4.

Notice that in addition to the user accounts you have defined, Access reveals the names of its two built-in users: Creator and Engine. To keep these built-in users from showing up in your users listing, use the following conditional statement:

```
If usr.Name <> "Creator" And usr.Name <> "Engine" Then
    Debug.Print "User:" & usr.Name
End If
```

### **Listing Users in Groups**

Sometimes you will need to know which users belong to which groups. The procedure in Hands-On 18.5 demonstrates how to obtain such a list, which is shown in Figure 18.15.



### Hands-On 18.5 Listing Users in Groups (ADO)

1. In the Visual Basic Editor window of Chap18.accdb, choose **Insert | Module**.
2. In the module's Code window, enter the **List\_UsersInGroups** procedure as shown here:

```
Sub List_UsersInGroups()
    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog
    Dim grp As New ADOX.Group
    Dim usr As New ADOX.User
    Dim strPath As String
    Dim strDB As String
    Dim strSysDB As String

    strPath = "C:\VBAAccess2021_ByExample\""
    strDB = "SpecialDb.mdb"
    strSysDB = "Security.mdw"

    ' Open connection to the database using
    ' the specified system database
    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Properties("Jet OLEDB:System Database") = _
            strPath & strSysDB
        .Properties("User ID") = "Developer"
        .Properties("Password") = "chapter18"
        .Open strPath & strDB
    End With

    ' Open the catalog
    Set cat = New ADOX.Catalog
    cat.ActiveConnection = conn
    For Each grp In cat.Groups
        Debug.Print "Group: " & grp.Name
        If cat.Groups(grp.Name).Users.Count = 0 Then
            Debug.Print vbTab & "There are no " & _
                "users in the " & grp & " group."
        End If
        For Each usr In cat.Groups(grp.Name).Users
            Debug.Print vbTab & "User: " & usr.Name
        Next usr
    Next grp

    Set cat = Nothing
```

```
conn.Close
Set conn = Nothing
MsgBox "Groups and Users are listed " & _
"in the Immediate window."
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

The procedure result is shown in Figure 18.16.

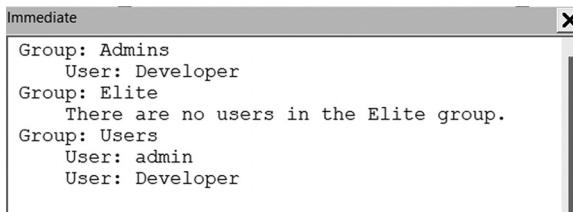


FIGURE 18.16. After running the procedure in Hands-On 18.5, security group account names and the corresponding user accounts are listed in the Immediate window.

## SETTING AND RETRIEVING USER AND GROUP PERMISSIONS

---

Users and groups of users can be granted specific permissions to database objects. For example, a user or an entire group of users can be authorized to only read an object's contents, while other users or groups can have less restrictive access to a database, allowing them to modify or delete objects.

<b>NOTE</b>	<i>It is important to understand that when you set permissions for a group, every user in that group automatically inherits those permissions. Also, keep in mind that while the user and group accounts are stored in the workgroup information file, the permissions that those users and groups have to specific objects are stored in system tables in your database.</i>
-------------	---

The following sections of this chapter will get you started using ADOX to retrieve, list, and set permissions for various database objects.

### Determining the Object Owner

---

The database, and every object in the database, has an owner. The owner is the user who created that particular object. The object owner has special privileges, including the ability to assign or revoke permissions for that object. To retrieve

the name of the object owner, use the `GetObjectOwner` method of a Catalog object. This method takes two parameters: the object's name and the object's type. For example, to determine the owner of a table, use the following syntax:

```
cat.GetObjectOwner (myObjName, adPermObjTable)
```

where `cat` is an object variable representing the ADOX Catalog object, `myObjName` is the name of a database table, and `adPermObjTable` is a built-in ADOX constant specifying the type of object. The constants for the Type parameter can be looked up in the Object Browser, as shown in Figure 18.17.

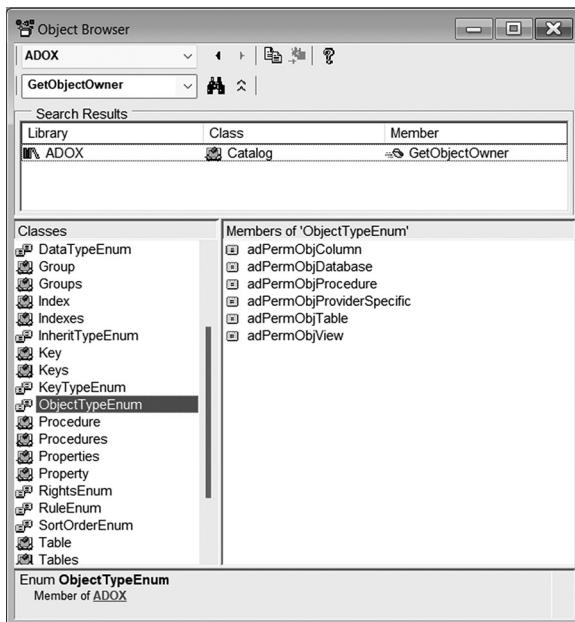


FIGURE 18.17. The Object Browser displays the available constants for the Type parameter of the `GetObjectOwner` method. It can be accessed by pressing F2 or choosing View | Object Browser.



### Hands-On 18.6 Retrieving the Name of the Object Owner (ADO)

1. In the Visual Basic Editor window of Chap18.accdb, choose **Insert | Module**.
2. In the module's Code window, enter the `Get_ObjectOwner` procedure as shown here:

```
Sub Get_ObjectOwner()
    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog
```

```
Dim strObjName As Variant
Dim strPath As String
Dim strDB As String
Dim strSysDB As String

strPath = "C:\VBAAccess2021_ByExample\""
strDB = "SpecialDb.mdb"
strSysDB = "Security.mdw"
strObjName = "Customers"

' Open connection to the database using
' the specified system database
Set conn = New ADODB.Connection
With conn
    .Provider = "Microsoft.ACE.OLEDB.12.0"
    .Properties("Jet OLEDB:System Database") = _
        strPath & strSysDB
    .Properties("User ID") = "Developer"
    .Properties("Password") = "chapter18"
    .Open strPath & strDB
End With

' Open the catalog
Set cat = New ADOX.Catalog
cat.ActiveConnection = conn

' Display the name of the table owner
MsgBox "The owner of the " & strObjName & _
    " table is " & vbCrLf _ 
    & cat.GetObjectOwner(strObjName, _ 
        adPermObjTable) & "."

Set cat = Nothing
conn.Close
Set conn = Nothing
End Sub
```

**3. Choose Run | Run Sub/UserForm to execute the procedure.**

To set the ownership of an object with ADOX, use the `SetObjectOwner` method of the Catalog object like this:

```
cat.SetObjectOwner("Customers", adPermObjTable, "PowerUser")
```

The preceding statement says that the ownership of the Customers table is to be transferred to the user named PowerUser. Note that currently there is no such user in the SpecialDb database. Recall that we created the PowerUser user account in Hands-On 18.2 and deleted it in Hands-On 18.3. If you want to experiment with changing object ownership, you need to make appropriate changes in the example procedure using the information you have already learned.

### **Setting User Permissions for an Object**

---

With ADOX, you set permissions on an object by using the `SetPermissions` method. User-level security can be easier to manage if you set permissions only for groups, and then assign users to the appropriate groups. Recall that permissions set for the group are automatically inherited by all users in that group. The `SetPermissions` method, which can be used for setting both user and group permissions, has the following syntax:

```
GroupOrUser.SetPermissions(Name, ObjectType, Action, Rights[, Inherit] [,ObjectTypeId])
```

- `Name`—The name of the object to set permissions on.
- `ObjectType`—The type of object the permissions are set for. (See Figure 18.17 for the names of the ADOX built-in constants that can be used to specify the `Type` parameter.)
- `Action`—The type of action to perform when setting permissions. Use the `adAccessSet` constant for Access databases to specify that the group of users will have exactly the requested permissions.
- `Rights`—A Long value containing a bitmask indicating the permissions to set. The `Rights` argument can consist of a single permissions constant or several constants combined with the OR operator. See Figure 18.18 for the names of the ADOX built-in constants that can be used in the `Rights` argument to specify the type of permissions to set.

**NOTE**

A **bitmask** is a numeric value intended for a bit-by-bit value comparison with other numeric values, usually to flag options in parameters or return values. In Visual Basic, this comparison is done with bitwise logical operators, such as AND and OR. The ADOX `GetPermissions` and `SetPermissions` methods use the bitwise logical operator OR to retrieve the bitmask for the existing permissions and to add new permissions to the bitmask.

The last two arguments (those in square brackets) are optional:

- `Inherit`—A Long value that specifies how objects will inherit these permissions. The default value is `adInheritNone`.
- `ObjectTypeId`—A Variant value that specifies the GUID (global unique identifier) for a provider object type not defined by OLE DB. This parameter is required if `ObjectType` is set to `adPermObjProviderSpecific` (which is used for setting permissions for forms, reports, and macros); otherwise, it is not used. See Table 18.2 for available GUIDs.

TABLE 18.2 GUIDs for provider objects

Object	GUID
Form	{c49c842e-9dcb-11d1-9f0a-00c04fc2c2e0}
Report	{c49c8430-9dcb-11d1-9f0a-00c04fc2c2e0}
Macro	{c49c842f-9dcb-11d1-9f0a-00c04fc2c2e0}

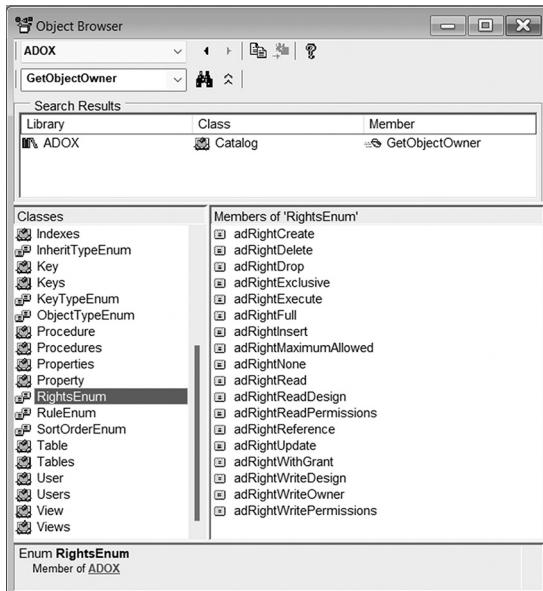


FIGURE 18.18 In ADOX, you can use many security constants for setting permissions to database objects.

The example procedure in Hands-On 18.7 grants a user the permission to read (`adRightRead`), insert (`adRightInsert`), update (`adRightUpdate`), and delete (`adRightDelete`) records.



### Hands-On 18.7 Setting User Permissions for an Object (ADO)

1. In the Visual Basic Editor window of Chap18.accdb, choose **Insert | Module**.
2. In the module's Code window, enter the **Set\_UserObjectPermissions** procedure as shown here:

```
Sub Set_UserObjectPermissions()
    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog
    Dim strPath As String
    Dim strDB As String
    Dim strSysDB As String

    On Error GoTo ErrorHandler

    strPath = "C:\VBAAccess2021_ByExample\
    strDB = "SpecialDb.mdb"
    strSysDB = "Security.mdw"

    ' Open connection to the database using
    ' the specified system database
    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Properties("Jet OLEDB:System Database") = _
            strPath & strSysDB
        .Properties("User ID") = "Developer"
        .Properties("Password") = "chapter18"
        .Open strPath & strDB
    End With

    ' Open the catalog
    Set cat = New ADOX.Catalog
    cat.ActiveConnection = conn
    ' add a user account
    cat.Users.Append "PowerUser", "star"

    ' Set permissions for PowerUser
    ' on the Customers table
    cat.Users("PowerUser").SetPermissions _
        "Customers", _
        adPermObjTable, _
        adAccessSet, _
        adRightRead Or _
        adRightInsert Or _
        adRightUpdate Or _
```

```

adRightDelete
MsgBox "Read, Insert, Update and Delete " & _
vbCrLf & " permissions were set on " & _
"Customers table for PowerUser."
ExitHere:
Set cat = Nothing
conn.Close
Set conn = Nothing
Exit Sub
ErrorHandler:
If Err.Number = -2147467259 Then
    MsgBox "PowerUser user already exists."
    Resume Next
Else
    MsgBox Err.Description
    Resume ExitHere
End If
End Sub

```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

### **Setting User Permissions for a Database**

---

To specify permissions for the database, specify an empty string ("") as the name of the database:

```
cat.Users("PowerUser").SetPermissions "", _
adPermObjDatabase, _
adAccessSet, adRightExclusive
```

This statement gives the user named PowerUser the right to open the database exclusively.

Figure 18.19 displays the permissions for the SpecialDb database that are set when the example procedure in Hands-On 18.8 is run.



### **Hands-On 18.8 Setting User Permissions for a Database (ADO)**

1. In the Visual Basic Editor window of Chap18.accdb, choose **Insert | Module**.
2. In the module's Code window, enter the **Set\_UserDbPermissions\_ADO** procedure as shown here:

```

Sub Set_UserDbPermissions_ADO()
    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog
    Dim strPath As String
    Dim strDB As String

```

```
Dim strSysDB As String

On Error GoTo ErrorHandler

strPath = "C:\VBAAccess2021_ByExample\"  
strDB = "SpecialDb.mdb"  
strSysDB = "Security.mdw"

' Open connection to the database using  
' the specified system database  
Set conn = New ADODB.Connection  
With conn  
    .Provider = "Microsoft.ACE.OLEDB.12.0"  
    .Properties("Jet OLEDB:System Database") = _  
        strPath & strSysDB  
    .Properties("User ID") = "Developer"  
    .Properties("Password") = "chapter18"  
    .Open strPath & strDB  
End With

' Open the catalog  
Set cat = New ADOX.Catalog  
cat.ActiveConnection = conn

' add a user account  
cat.Users.Append "PowerUser", "star"

' Set permissions for PowerUser  
cat.Users("PowerUser").SetPermissions "", _  
    adPermObjDatabase, adAccessSet, _  
    adRightExclusive  
MsgBox "PowerUser has been granted " & _  
    vbCrLf & "permission to open the " & _  
    "database exclusively."  
ExitHere:  
    Set cat = Nothing  
    conn.Close  
    Set conn = Nothing  
    Exit Sub  
ErrorHandler:  
    If Err.Number = -2147467259 Then  
        ' because PowerUser user already exists  
        ' we ignore this statement  
        Resume Next  
    Else
```

```
    MsgBox Err.Description
    Resume ExitHere
End If
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure.

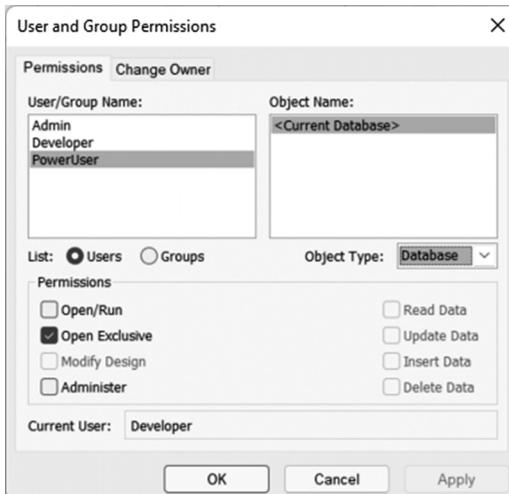


FIGURE 18.19 The settings shown here are found in the User and Group Permissions window for the SpecialDb database after running the Set\_UserDbPermissions\_ADO procedure in Hands-On 18.8.

### Setting User Permissions for Containers

Now that you've learned how to grant permissions to a user for a specific object such as a table or query, you may want to know how to specify permissions for an entire set of objects such as tables, queries, forms, reports, and macros.

Each Database object has a `Containers` collection consisting of built-in Container objects. A Container object groups together similar types of Document objects. You can use the `Containers` collection to set security for all Document objects of a given type. You can set the permissions that users and groups will receive by default on all newly created objects in a database by passing in Null for the object name argument of the ADOX `SetPermissions` method, as shown in the example procedure in Hands-On 18.9.



#### Hands-On 18.9 Setting User Permissions for Containers (ADO)

1. In the Visual Basic Editor window of Chap18.accdb, choose **Insert | Module**.

2. In the module's Code window, enter the **Set\_UserContainerPermissions\_ADO** procedure as shown here:

```
Sub Set_UserContainerPermissions_ADO()
    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog
    Dim strPath As String
    Dim strDB As String
    Dim strSysDB As String

    On Error GoTo ErrorHandler

    strPath = "C:\VBAAccess2021_ByExample\
    strDB = "SpecialDb.mdb"
    strSysDB = "Security.mdw"

    ' Open connection to the database using
    ' the specified system database
    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Properties("Jet OLEDB:System Database") = _
            strPath & strSysDB
        .Properties("User ID") = "Developer"
        .Properties("Password") = "chapter18"
        .Open strPath & strDB
    End With

    ' Open the catalog
    Set cat = New ADOX.Catalog
    cat.ActiveConnection = conn

    ' add a user account
    cat.Users.Append "PowerUser", "star"

    ' Set permissions for PowerUser on
    ' the Tables Container
    cat.Users("PowerUser").SetPermissions Null, _
        adPermObjTable, _
        adAccessSet, _
        adRightRead Or _
        adRightInsert Or _
        adRightUpdate Or _
        adRightDelete, adInheritNone
    MsgBox "You have granted " & vbCrLf & _
        "permissions to PowerUser on " & _
```

```

    "the Tables Container."
ExitHere:
    Set cat = Nothing
    conn.Close
    Set conn = Nothing
    Exit Sub
ErrorHandler:
    If Err.Number = -2147467259 Then
        ' because PowerUser user already exists
        ' we ignore this statement
        Resume Next
    Else
        MsgBox Err.Description
        Resume ExitHere
    End If
End Sub

```

**3. Choose Run | Run Sub/UserForm to execute the procedure.**

This procedure gives the PowerUser account the permission to design, read, update, insert, and delete data for all newly created tables and queries. Notice that Null is passed as the first argument of the `SetPermissions` method to indicate that permissions are to be set only on new objects of the type specified by the second argument of this method.

After executing this procedure, the user account PowerUser has the permissions listed in Figure 18.20 on all newly created Table and Query objects.

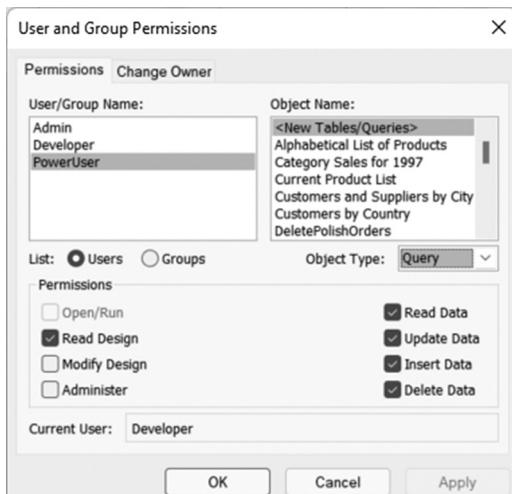


FIGURE 18.20 The settings shown here are found in the User and Group Permissions window after running the `Set_UserContainerPermissions_ADO` procedure in Hands-On 18.9.

## Checking Permissions for Objects

You can retrieve the permissions for a particular user or group on a particular object with the ADOX `GetPermissions` method. Because this method returns a numeric permission value for the specified object, you must write more code to decipher the returned value if you want to display the names of constants representing permissions. The procedure in Hands-On 18.10 demonstrates how to retrieve the permissions set for PowerUser on the Customers table in a sample database (Figure 18.20).

### Hands-On 18.10 Checking Permissions for a Specific Object (ADO)

1. In the Visual Basic Editor window of Chap18.accdb, choose **Insert | Module**.
2. In the module's Code window, enter the `GetObjectPermissions_ADO` procedure as shown here:

```
Sub GetObjectPermissions_ADO(strUserName As String, _
    varObjName As Variant, _
    lngObjType As ADOX.ObjectTypeEnum)

    Dim conn As ADODB.Connection
    Dim cat As ADOX.Catalog
    Dim strPath As String
    Dim strDB As String
    Dim strSysDB As String
    Dim listPerms As Long
    Dim strPermsTypes As String

    On Error GoTo ErrorHandler

    strPath = "C:\VBAAccess2021_ByExample\"
    strDB = "SpecialDb.mdb"
    strSysDB = "Security.mdw"

    ' Open connection to the database using
    ' the specified system database
    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.ACE.OLEDB.12.0"
        .Properties("Jet OLEDB:System Database") = _
            strPath & strSysDB
        .Properties("User ID") = "Developer"
        .Properties("Password") = "chapter18"
        .Open strPath & strDB
    End With
End Sub
```

```
End With

' Open the catalog
Set cat = New ADOX.Catalog
cat.ActiveConnection = conn
' add a user account
cat.Users.Append "PowerUser", "star"

listPerms = cat.Users(strUserName) _
    .GetPermissions(varObjName, lngObjType)
Debug.Print listPerms

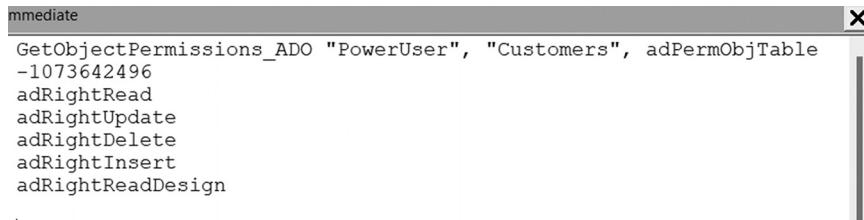
If (listPerms And ADOX.RightsEnum.adRightCreate) = _
    adRightCreate Then
    strPermsTypes = strPermsTypes & _
        "adRightCreate" & vbCrLf
End If
If (listPerms And RightsEnum.adRightRead) = _
    adRightRead Then
    strPermsTypes = strPermsTypes & _
        "adRightRead" & vbCrLf
End If
If (listPerms And RightsEnum.adRightUpdate) = _
    adRightUpdate Then
    strPermsTypes = strPermsTypes & _
        "adRightUpdate" & vbCrLf
End If
If (listPerms And RightsEnum.adRightDelete) = _
    adRightDelete Then
    strPermsTypes = strPermsTypes & _
        "adRightDelete" & vbCrLf
End If
If (listPerms And RightsEnum.adRightInsert) = _
    adRightInsert Then
    strPermsTypes = strPermsTypes & _
        "adRightInsert" & vbCrLf
End If
If (listPerms And RightsEnum.adRightReadDesign) = _
    adRightReadDesign Then
    strPermsTypes = strPermsTypes & _
        "adRightReadDesign" & vbCrLf
End If

Debug.Print strPermsTypes
MsgBox "Permissions are listed in " & _
```

```
"the Immediate Window."  
ExitHere:  
    Set cat = Nothing  
    conn.Close  
    Set conn = Nothing  
    Exit Sub  
ErrorHandler:  
    If Err.Number = -2147467259 Then  
        ' because PowerUser user already exists  
        ' we ignore this statement  
        Resume Next  
    Else  
        MsgBox Err.Description  
        Resume ExitHere  
    End If  
End Sub
```

3. To run the GetObjectPermissions\_ADO procedure, type the following statement in the Immediate window and press **Enter** to execute it:

```
GetObjectPermissions_ADO "PowerUser", "Customers", adPermObjTable
```



The screenshot shows the Microsoft Visual Studio Immediate window. The title bar says "Immediate". The window contains the following text:

```
GetObjectPermissions_ADO "PowerUser", "Customers", adPermObjTable  
-1073642496  
adRightRead  
adRightUpdate  
adRightDelete  
adRightInsert  
adRightReadDesign
```

FIGURE 18.21 The procedure in Hands-On 18.10 writes the permissions found for PowerUser in the Customers table to the Immediate window.

### **Setting a Database Password Using the DbEngine.CompactDatabase Method**

You can implement share-level security by setting a database password. When you set a database password, the password dialog box will appear when you open the database. Only users with a valid password can open the database.

Use the `CompactDatabase` method of the `DbEngine` object to copy and compact a database and specify password. This method has two required parameters:

- a full path and file name of an existing (closed) database
  - a full path and file name of the compacted database you are creating
- There are also several optional parameters. The third parameter that we

will use in our Hands-On is a string specifying the collating order of the compacted database. We will use the `dbLangGeneral` constant for English, German, French, Portuguese, Italian, and Modern Spanish. To set a password on the compacted database, you can concatenate to this argument the password string starting with the “;pwd=” as shown in the next Hands-On 18.11. Notice the use of the semicolon here.

Remember that passwords are case sensitive and should be strong (a mix of upper-and lowercase letters, numbers, and symbols). Check out the Microsoft online documentation on other parameters used with the `CompactDatabase` method:

<https://docs.microsoft.com/en-us/office/client-developer/access/desktop-database-reference/dbengine-compactdatabase-method-dao>

The procedure in Hands-On 18.11 compacts a database and sets the sample database password to “Welcome@18.”



### Hands-On 18.11 Setting a Database Password

1. Create a new Access database in Access 2002-2003 format named `C:\VBAAccess2021_ByExample\PasswordTest.mdb`. Close this database before proceeding to step 2.
2. Reopen the `Chap18.accdb` file if it was closed. Switch to the Visual Basic Editor window and choose **Insert | Module**.
3. In the module’s Code window, enter the `Compact_andSetPwd` procedure as shown here:

```
Sub Compact_andSetPwd()
    Dim strCompactFrom As String
    Dim strCompactTo As String
    Dim strPath As String

    On Error GoTo ErrHandler
    strPath = CurrentProject.Path & "\"

    strCompactFrom = strPath & "PasswordTest.mdb"
    strCompactTo = strPath & "PasswordTest_Compact.mdb"

    DBEngine.CompactDatabase strCompactFrom, strCompactTo, _
        dbLangGeneral & ";pwd=Welcome@18"

    ExitHere:
    Exit Sub

```

```
ErrorHandler:  
    If Err.Number = 3204 Then  
        ' delete the database if it already exists  
        If Dir(strCompactTo) <> "" Then Kill strCompactTo  
        Resume  
    Else  
        MsgBox Err.Number & ":" & _  
            Err.Description  
        Resume ExitHere  
    End If  
End Sub
```

4. Choose **Run | Run Sub/UserForm** to execute the procedure.
5. After you run this procedure, open the **PasswordTest\_Compact.mdb** database file. You should be prompted for the password. Type **Welcome@18** to log in.
6. Close the **PasswordTest\_Compact.mdb** file.

### Setting a Database Password Using the NewPassword Method

---

The Database object in the DAO Object Model has a `NewPassword` method that you can use to change the password of an existing Access database in .accdb or .mdb file format. The `NewPassword` method requires two parameters. The first one specifies the old password, and the second one provides the new password. Both passwords can be up to 20 characters long and can include any characters except the ASCII character 0 (Null). Use a zero-length string ("") for the old password if the database does not have a password. Use a zero-length string ("") for the new password to clear the password. Password operations require that the database is open in exclusive mode. Remember that passwords are case sensitive.

The procedure in Hands-On 18.12 sets the password for the Chap16.accdb database you created earlier in this book.



#### Hands-On 18.12 Setting a Database Password (DAO)

1. In the Visual Basic Editor window of Chap18.accdb, choose **Insert | Module**.
2. In the module's Code window, enter the `Set_DBPassword.DAO` procedure as shown here:

```
Sub Set_DBPassword.DAO()  
    Dim db As DAO.Database  
    Dim strDB As String
```

```
strDB = CurrentProject.Path & "\Chap16.accdb"

Set db = OpenDatabase(strDB, True)

db.NewPassword "", "chapter16"
db.Close
Set db = Nothing
End Sub
```

3. Choose **Run | Run Sub/UserForm** to execute the procedure. The second parameter (`True`) in the `OpenDatabase` method tells VBA to open the database in exclusive mode.
4. Open the **Chap16.accdb** database and notice that you are now prompted to enter a password. Type **chapter16** for the password and click **OK**. When the database opens, close it and close the Access window in which it was opened. Do not close the Chap18.accdb database.

You can unset the password on the Chap16.accdb database by running the following procedure:

```
Sub Unset_DBPassword.DAO()
    Dim db As DAO.Database
    Dim strDB As String

    strDB = CurrentProject.Path & _
        "\Chap16.accdb"
    Set db = OpenDatabase(strDB, True, _
        False, ";pwd=chapter16")

    db.NewPassword "chapter16", ""
    db.Close
    Set db = Nothing
End Sub
```

Let's review the parameters of the `OpenDatabase` method. `True` specifies that the database is to be opened in exclusive mode, and `False` indicates that the database should be opened in read/write mode. Because the Chap16.accdb database has been protected with a password by the `Set_DBPassword.DAO` procedure in Hands-On 18.12, we also had to specify the password in the `connect` parameter. After you run the preceding procedure, you will not be prompted for a password when you open the Chap16.accdb database.

## Changing a User Password

User passwords are stored in the workgroup information file. To change a user's password in VBA code, use the ADOX User object's `ChangePassword` method. This method takes as parameters the user's current password and the new password. If a user does not yet have a password, use an empty string ("") for the user's current password.

The procedure in Hands-On 18.13 demonstrates how to change a password for the Admin user. Recall that Admin is the default user account that has a blank password. In an unsecured Access database, all users are automatically logged on using the Admin account. When establishing user-level security, you should start by changing the password for the Admin user. Changing an Admin password activates the Logon dialog box the next time you start Access. Only users with a valid username and password will be able to log onto the database. Although users are permitted to change their own passwords, only a user who belongs to the Admins group can clear a password that another user has forgotten.



### Hands-On 18.13 Changing a User Password (ADO)

1. Create a new Access database named **AdminPwd.mdb** in your **C:\VBAAccess2021\_ByExample** folder. Close this database before proceeding to step 2.
2. In the Visual Basic Editor window of Chap18.mdb, choose **Insert | Module**.
3. In the module's Code window, enter the **Change\_UserPassword\_ADO** procedure as shown here:

```
Sub Change_UserPassword_ADO()
    Dim cat As ADOX.Catalog
    Dim strDB As String
    Dim strSysDB As String

    On Error GoTo ErrorHandler

    strDB = CurrentProject.Path & "\AdminPwd.mdb"
    ' change the path to use the default
    ' workgroup information file on your computer
    strSysDB = "C:\Users\Julitta\" &
    "AppData\Roaming\Microsoft\Access\System1.mdw"

    ' Open the catalog, specifying the system
    ' database to use
```

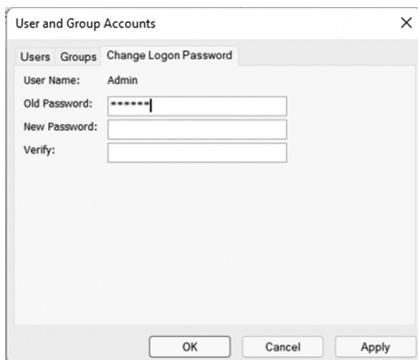
```

Set cat = New ADOX.Catalog
With cat
    .ActiveConnection =
        "Provider='Microsoft.ACE.OLEDB.12.0';" & _
        "Data Source='" & strDB &"';" & _
        "Jet OLEDB:System Database=''" & _
        strSysDB & ";" & _
        "User Id=Admin;Password=;"
    ' Change the password for the Admin user
    .Users("Admin").ChangePassword "", "secret"
End With

ExitHere:
Set cat = Nothing
Exit Sub
ErrorHandler:
MsgBox Err.Description
GoTo ExitHere
End Sub

```

4. Choose **Run | Run Sub/UserForm** to execute the procedure.
5. When you open the AdminPwd.mdb database after running the procedure, a Logon dialog box will appear. Enter **Admin** in the Name text box and **secret** in the Password text box, and click **OK**. Remove the Admin password by choosing **File | Info | Users and Permissions | User and Group Accounts**. Click the **Change Logon Password** tab and type **secret** in the old password. Click the **Apply** button, and click **OK** to exit the User and Group Accounts window (Figure 18.22).



**FIGURE 18.22** You can remove the Admin password via the Change Logon Password tab in the User and Group Accounts window or by modifying the VBA code shown in Hands-On 18.13.

6. After removing the Admin password, reopen the **AdminPwd.mdb** file. The database should now open without prompting you to enter a password.
7. Close the **AdminPwd.mdb** database and exit the Access window in which the file was opened.

## SUMMARY

---

In this chapter, you worked with VBA procedures that implemented share-level and user-level security in Access databases. You found out that to use user-level security, you must create an Access database in the 2002-2003 .mdb file format. This chapter introduced you to workgroup information files that contain group and user information, including passwords. You practiced creating and modifying user and group accounts and setting user permissions to a database and its objects. You also learned how to set a database password on Access databases using the DAO CompactDatabase method of the JET Database Engine.

This chapter concludes Part IV of this book, in which we focused on writing VBA procedures that deal with security aspects of Access databases.



Part

# V

# *VBA PROGRAMMING IN ACCESS FORMS AND REPORTS*

The behavior of Access objects such as forms, reports, and controls can be modified by writing programming code known as an event procedure or an event handler. In this part of the book, you will learn how you can design more effective and visually appealing forms and reports, and make your forms, reports, and controls perform useful tasks by writing event procedures in class modules.

Chapter 19 Enhancing Access Forms

Chapter 20 Using Form Events

Chapter 21 Events Recognized by Form Controls

Chapter 22 Enhancing Access Reports and Using Report Events



# Chapter 19 ENHANCING ACCESS FORMS

Access 2021 offers users a great number of features in the form design area. For example, the Layout view gives form interface a true WYSIWYG: you can see the live data as you design your form without the need to constantly switch between the Design and Form views. The form features include various methods of creating forms, the Split Form, Bound Image controls, the Attachments control, styles and AutoFormats, rich text support as well as various ways of grouping controls by using the layouts. Many features are available in datasheets, including the Date Picker, alternating row colors, the Totals row, truncated number displays, sorting and filtering, and an easy way to add new list items to combo boxes.

The Navigator control allows the creation of modern-looking tab-style navigation forms that replace the old-fashioned switchboard style form frequently seen in older Access applications. With the Navigation control, it is possible to create richer user interfaces with “parent” and “child” navigation forms. You can further enhance your Access forms by linking subreports to the form and making it possible for users to view information related to the record as they navigate the form. You can publish your application to the Web using SharePoint. With the Web Form Designer you can quickly and easily create forms in tabular format that are properly rendered on SharePoint via Access Services.

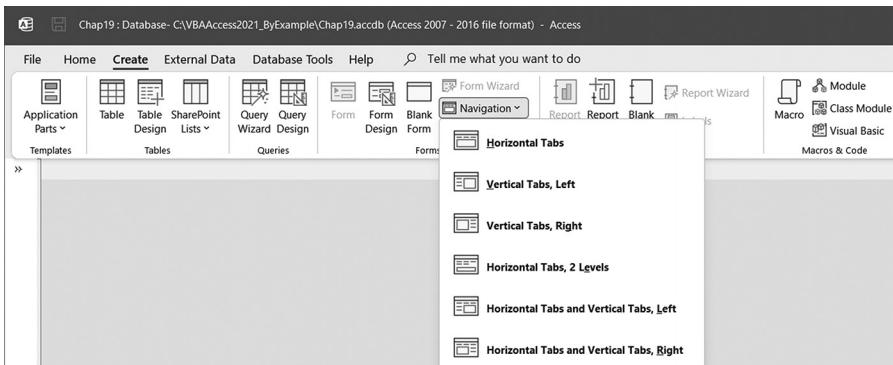
## CREATING ACCESS FORMS

---

The form buttons on the Ribbon's Create tab (see Figure 19.1) enable users to create both simple and advanced forms.

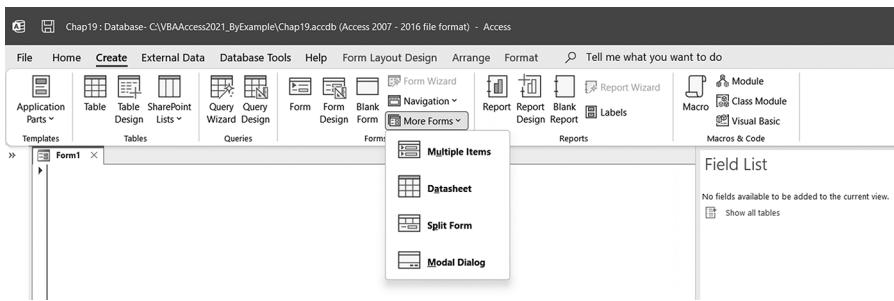
The classic Access form with a columnar layout is generated automatically by selecting a record source (a table or query) in the Navigation pane and clicking the Form button. The next button, called Form Design, is used to create a blank form in Design view. This form is not bound to any data source. Instead, you are presented with a list of tables and queries on which you can base your form. The Blank Form button on the Ribbon can be used to create a custom form from scratch in Layout view. This form is not connected to any data source and can be used to create any form you want. The Form Wizard button allows you to create simple customizable forms. When you use the Form Wizard, Access allows you to specify which fields to include from which table or query and makes it easy to choose from a variety of AutoFormats. Using the wizard lets you choose only the fields you want so you don't have to spend extra time deleting the fields you don't want and repositioning the remaining fields.

The Navigation button is a gallery control that allows you to create forms that browse to other forms and reports. When you click the Navigation button, it will present a selection of different layouts (see Figure 19.1).



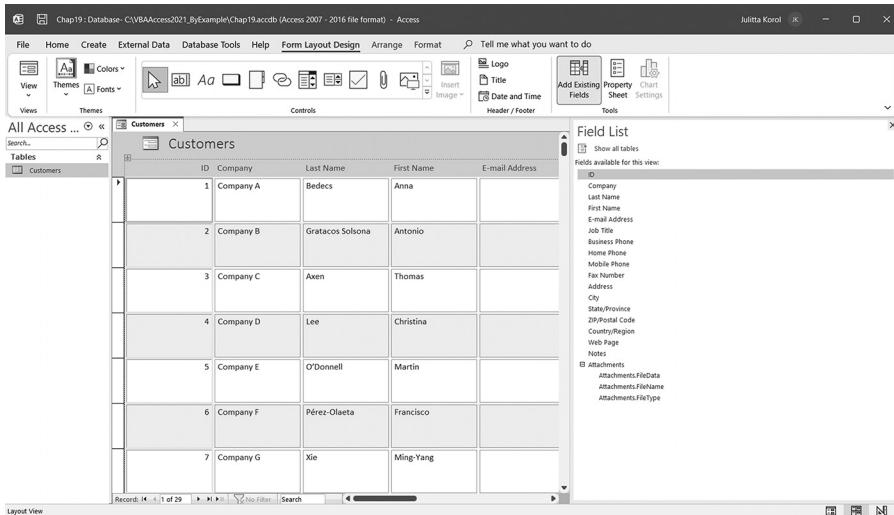
**FIGURE 19.1.** Use the Navigation control button on the Create tab (Forms group) to create customizable navigation forms.

The More Forms button (see Figure 19.2) provides additional types of forms users can create in Access 2021: Multiple Items, Datasheet, Split Form, and Modal Dialog. Notice there are no options for creating PivotChart and Pivot-Table forms.



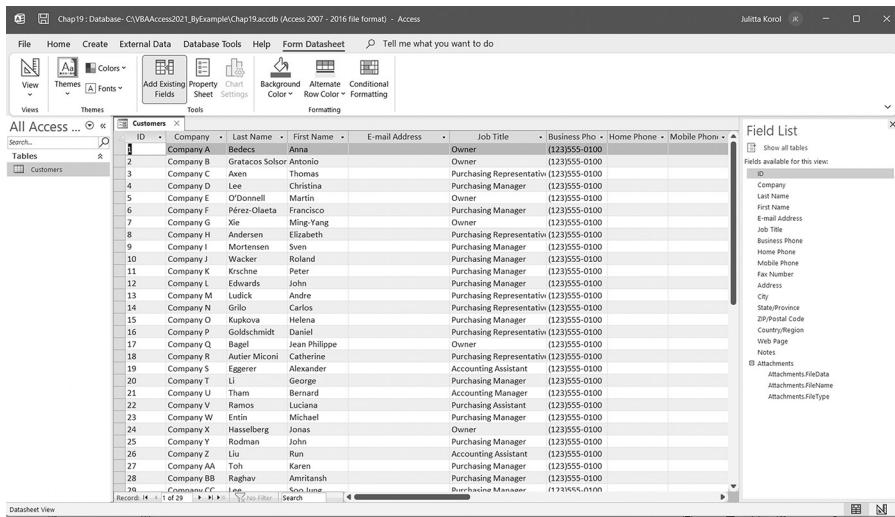
**FIGURE 19.2.** The More Forms button provides many types of forms that you can create in Access 2021.

The Multiple Items form is a standard continuous form used in earlier versions of Access. This form (see Figure 19.3) displays multiple records in a datasheet, with one record per row, and allows you to arrange the controls any way you want. To create this type of form, select the appropriate table or query in the Navigation pane and choose Create | More Forms | Multiple Items.



**FIGURE 19.3** The Multiple Items form in Access 2021.

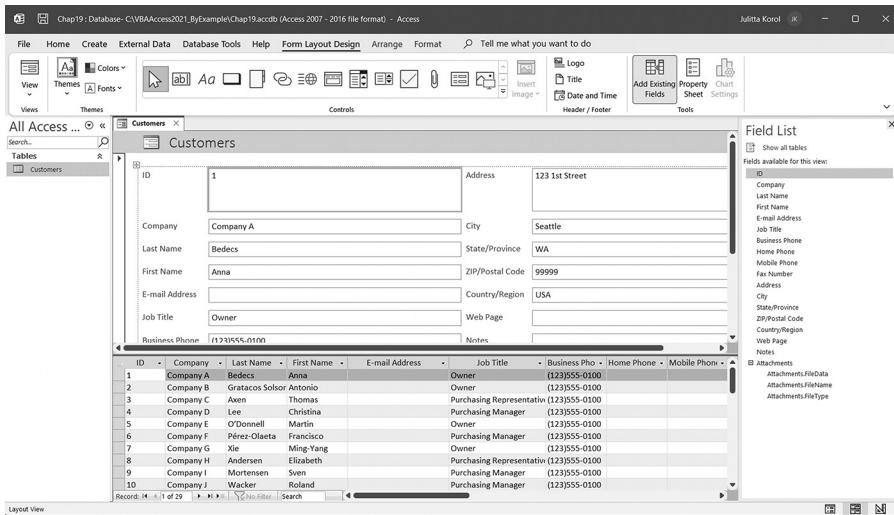
The Datasheet form (see Figure 19.4) organizes data like the Multiple Items form but looks more like an Excel worksheet with one record per row. To create this type of form, select the appropriate table or query in the Navigation pane and choose Create | More Forms | Datasheet.



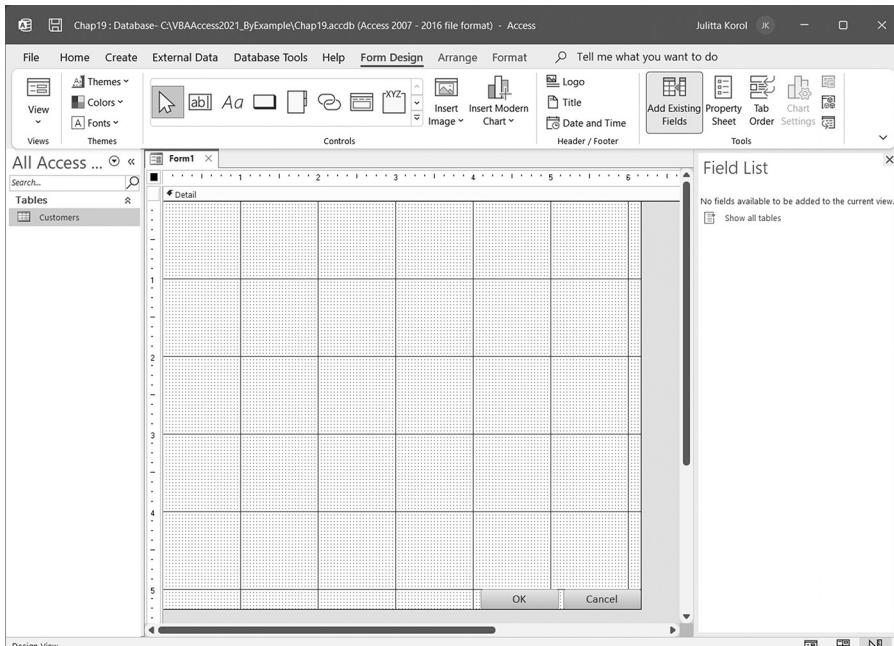
**FIGURE 19.4.** The Datasheet form resembles an Excel worksheet.

The Split Form contains a datasheet and a standard Access form (see Figure 19.5). The datasheet displays multiple records. Simply click on the record in the datasheet and the form will change to show the details for this record, which you can edit. Access can create these types of forms with ease without asking you a single question. To create this type of form, select the appropriate table or query in the Navigation pane and choose Create | More Forms | Split Form.

The Modal Dialog form (see Figure 19.6) opens a form that functions like a modal window. This means that the user will not be able to activate any other object before closing that form. Modal Dialog forms are very useful when you need to gather specific information from users before allowing them to perform other actions. To create this type of form, select the appropriate table or query in the Navigation pane and choose Create | More Forms | Modal Dialog. Notice that Modal form comes with two buttons: OK and Cancel.



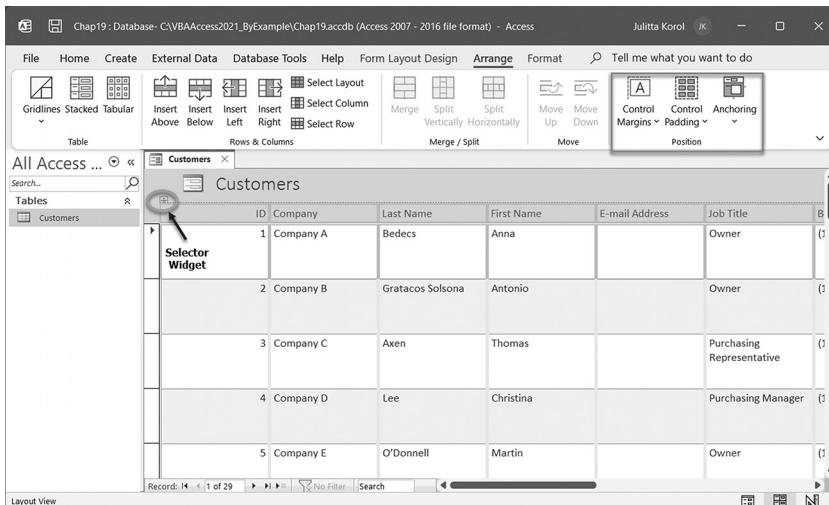
**FIGURE 19.5.** The Split Form is a combination of a standard form in an upper section and a datasheet in a lower section. It allows easy browsing through the records and entering or editing data for the selected record in the standard form.



**FIGURE 19.6** The Modal Form

## GROUPING CONTROLS USING LAYOUTS

In Access, you can group controls through a feature known as Layouts. Layout view enables you to work with entire groups of controls without having to guess whether the controls are properly sized and positioned. Take a look at Figure 19.7 and notice the Selector widget. When you click on the widget, you will see which controls are included in that layout. Using the Layout view you can easily move controls around in the form and resize them. To control the layouts, use the buttons in the Position section of the Ribbon's Arrange tab.



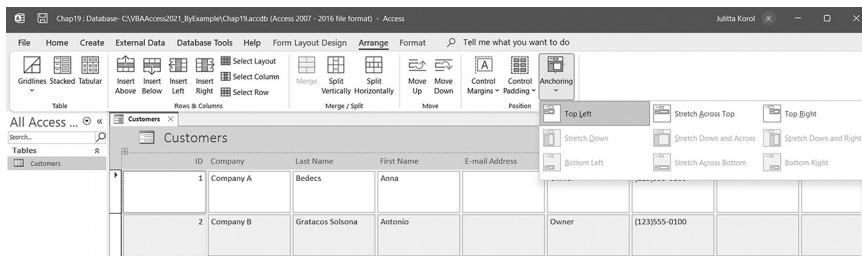
**FIGURE 19.7.** A group of controls on the form can be moved easily by using the anchor point (Selector widget).

Groups of controls can be moved to a new layout in one step. The tabular layout makes it easy to group the controls similar to a spreadsheet, with labels positioned across the top and data displayed in columns below the labels. To do this quickly, select the control group you want to reposition by clicking the anchor (Selector widget), then click the Tabular button in the Table group of the Ribbon. The Stacked button can be used to create a layout similar to a paper form with labels to the left of the data. Removing entire groups of controls is also easily done by clicking the Selector widget and pressing the Delete key.

You can use anchoring to tie a control or a group of controls to a section or another control so it moves into place in accordance with the parent. The Anchoring options in Figure 19.8 show various positions where controls can be

moved and ways they can be stretched to maximize the use of the space available on the form.

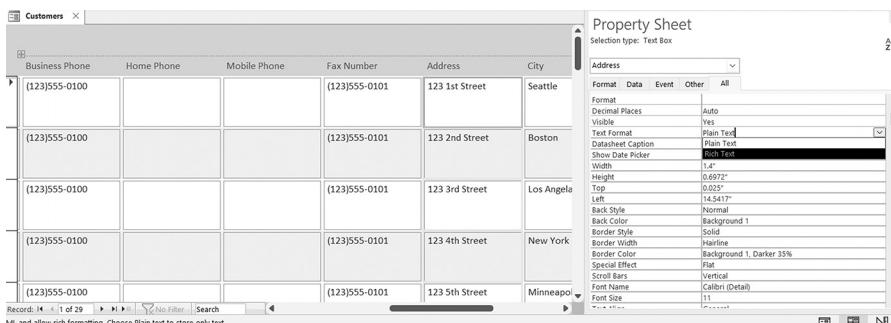
To keep users from making changes to the form, you can disallow the Layout view in the property sheet for the form.



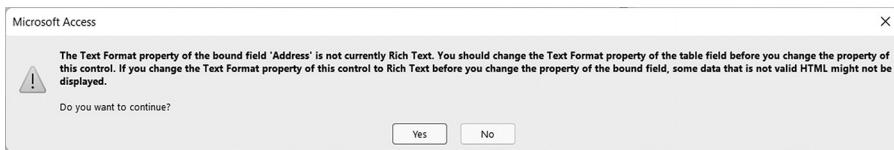
**FIGURE 19.8** The Anchoring button reveals several options for positioning and stretching form controls.

## RICH TEXT SUPPORT IN FORMS

For a nicer user experience, you can enable rich text formatting on Access forms via the Text Box control. For an unbound text box, simply open the property sheet for the text box and set its Text Format property to Rich Text (see Figure 19.9). This change will enable text formatting tools on the Ribbon. At runtime, users will be able to change the font style and color; add bold, italics, and highlighting; and apply other formatting. If the text box is bound to a field in a table, you must first change the Text Format property of the table field before changing the property of the control. If you forget to do this, Access will display the warning message shown in Figure 19.10.



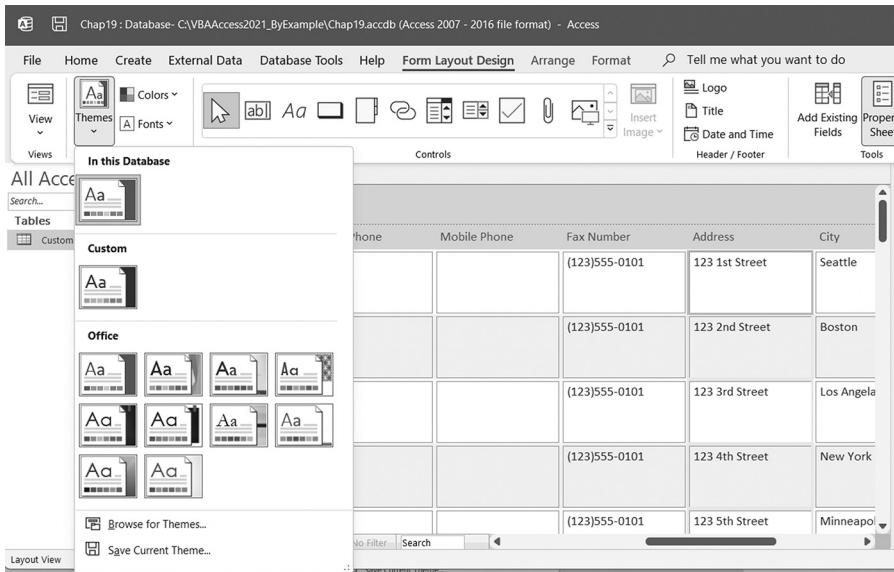
**FIGURE 19.9.** Access allows the Rich Text format to be used in text box controls placed on the form.



**FIGURE 19.10.** Before changing the Text Format of a bound text box control to Rich Text, be sure to change the Text Format property of the table field.

## USING BUILT-IN FORMATTING TOOLS

Access provides a gallery of themes you can use to give your forms a pleasing and consistent look. To preview the available designs, switch to the Form Layout view and click on the Themes button in the Themes section of the Design tab (see Figure 19.11).



**FIGURE 19.11.** The Themes button provides a gallery of quick formats in Access.

## USING IMAGES IN ACCESS FORMS

Access 2007 came with better image support. Earlier versions of Access converted images from their native format and stored them as bitmaps (.bmp). This format caused a significant increase in the size of the database because bitmap files are not compressed. Also, any image transparency features were lost during

the conversion process. If you use the Image control in Access 2007–2021 and specify the image in the Picture property, Access will store the image in its native format with no conversion. Images with transparency work just fine. You can see examples of transparent buttons and pictures in the Northwind 2007 database's forms (see Figure 19.12).

ID	Company	First Name	Last Name	E-mail Address	Business Phone	Job Title
8	Company H	Elizabeth	Andersen		(123)555-0100	Purchasing Repres
18	Company R	Catherine	Autier Miconi		(123)555-0100	Purchasing Repres
3	Company C	Thomas	Axen		(123)555-0100	Purchasing Repres
17	Company Q	Jean Philippe	Bagel		(123)555-0100	Owner
1	Company A	Anna	Bedecs		(123)555-0100	Owner
12	Company L	John	Edwards		(123)555-0100	Purchasing Manager
19	Company S	Alexander	Eggerer		(123)555-0100	Accounting Assistan
23	Company W	Michael	Entin		(123)555-0100	Purchasing Manager
16	Company P	Daniel	Goldschmidt		(123)555-0100	Purchasing Repres
2	Company B	Antonio	Gratacos Solsona		(123)555-0100	Owner
14	Company N	Carlos	Grilo		(123)555-0100	Purchasing Repres
24	Company X	Jonas	Hasselberg		(123)555-0100	Owner

FIGURE 19.12. Access form with transparent images and buttons.

The older MDB databases do not support saving images in the native format, so there is a special database property that lets you choose whether the images should be converted to DIB (device-independent bitmap) or stored in their native format. The default setting is to store images in their native format and convert them to bitmaps for MDB databases. If you want your images to be displayed in previous versions of Access, choose the second option button under the Picture Property Storage Format setting (see Figure 19.13).

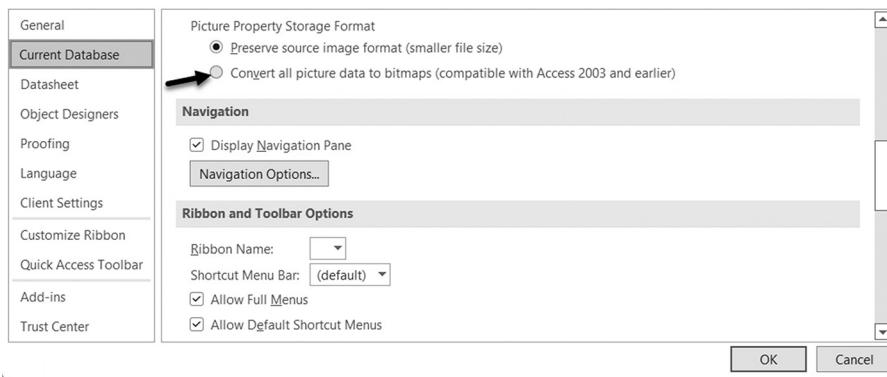


FIGURE 19.13. You can tell Access how to store images in older versions of Access (2003 and earlier) by using the options under Picture Property Storage Format. The Access Options window can be accessed by clicking File | Options.

Access 2007 introduced a bound Image control. Access 2003 and earlier needed lots of VBA code to display images on forms and reports when the images were stored in the directories on disk. The Image control can be bound to the image path. To add an image to your form, place the form in Design view and click the Image button in the Controls group of the Ribbon's Form Design tab. Click Browse and choose the image file. When you click OK, you are returned to the form design and your cursor displays a small image next to it. Click and drag on the form to define the area where you'd like the image to be placed. When you release the mouse button, the selected picture is placed in the selected area on the form. If you activate the property sheet for the Image control, you will see that Access has placed the filename in the Picture property (see Figure 19.14). If you don't like the picture you've chosen, you can simply click the ellipsis button (...) next to the Picture property and choose another image or choose another image (if available) from the Image Gallery accessed via the Insert Image button.

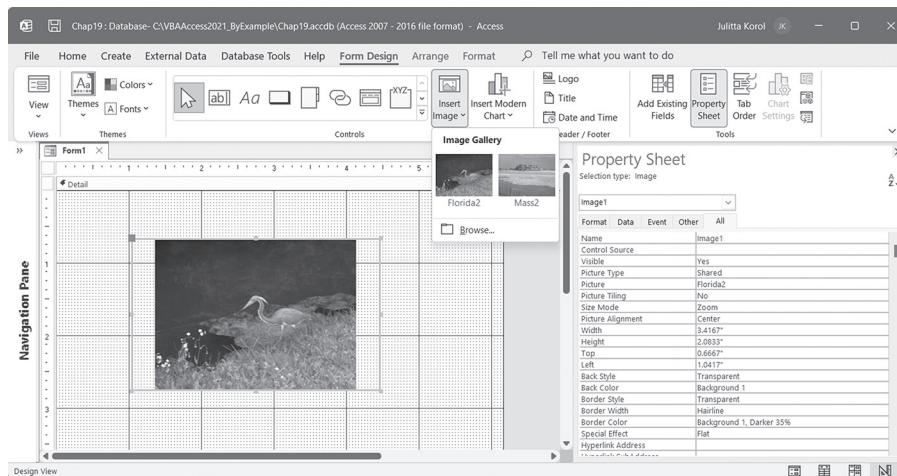


FIGURE 19.14. The picture is shown here using the Image control placed on an Access form.

## USING THE ATTACHMENTS CONTROL

In the Controls group of the Form Design tab you will find an Attachments control (Attachment icon) that enables you to attach a file or a collection of files to any database record. When you click on the Attachments field on the form, Access displays

a small toolbar with three buttons (see Figure 19.15). The Forward and Backward buttons allow you to move through the attached files, and the third button opens the Attachments dialog box. You can also right-click the Attachments field and choose the same options from the shortcut menu.

Recall that in Chapter 11 you wrote a VBA procedure that added an attachment field to an existing table. Let's see how you can work with the attachments in an Access form.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Custom Project 19.1 Working with the Attachments Control

1. Start Access and create a new database named **Chap19.accdb** in your C:\VBAAccess2021\_ByExample folder.
2. Choose **External Data | Access** to import the Customers table. In the File name box, type **C:\VBAAccess2021\_ByExample\Northwind 2007.accdb** and click **OK**. In the Import Objects window, activate the **Tables** tab and select the **Customers** table, then click **OK**. Click **Close** to exit the dialog boxes when the import process has completed.
3. In the Navigation pane, double-click the **Customers** table.
4. Double-click on the paper clip icon for the third record. You should see an empty Attachments dialog box. Click the **Add** button, select the **California1.jpg** and **California2.jpg** images. These files can be found with the companion files in the External Docs folder. When you select the files and click **Open**, the image names should appear the Attachments dialog box.
5. Click **OK** to close the dialog box and press **Ctrl+S** to save the record. Notice that the paper clip column now displays the number of attached files in parentheses next to the paper clip icon for the record.
6. Close the Customers table.
7. Highlight the **Customers** table in the Navigation pane, then click the **Form** button in the Forms group on the Ribbon's Create tab.  
Access will display a form as shown in Figure 19.15.
8. Activate the record for the third customer.
9. On the Customers form, click the **Attachments** control next to the Attachments label; notice a small toolbar with three buttons. Scroll through the attached files by clicking the Forward and Backward buttons.

The screenshot shows an Access form titled "Customers". The form displays a single record for a customer. The fields include: ID (3), Company (Company C), Last Name (Axen), First Name (Thomas), E-mail Address (empty), Job Title (Purchasing Representative), Business Phone ((123)555-0100), Home Phone (empty), Mobile Phone (empty), Fax Number ((123)555-0101), Address (123 3rd Street), City (Los Angeles), State/Province (CA), ZIP/Postal Code (99999), Country/Region (USA), and Web Page (empty). Below the form, there is a "Attachments" section with a thumbnail image and a "Manage Attachments" button. The status bar at the bottom shows "Record: 3 of 29" and "No Filter".

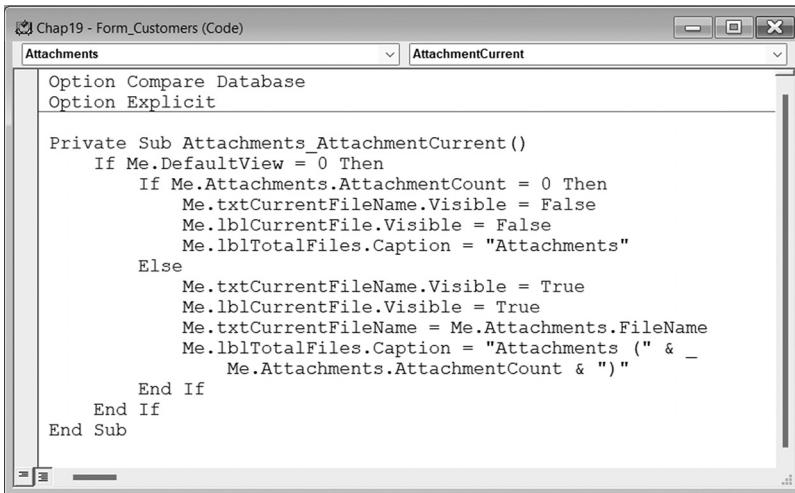
FIGURE 19.15. The Access form uses the Attachments control to show images attached to a record.

- Let's modify the form to display additional information about the attachments.
10. Switch to the Design view of the Customers form and use the **Text Box control** in the Controls group of the Form Design tab to add a text box to the form as shown in Figure 19.16. Change the default label of the text box control to **Current File** as shown.

The screenshot shows the "Customers" form in Design view. A new unbound text box control is added to the form, labeled "Current File", positioned next to the "Mobile Phone" field. The form is divided into sections: "Detail", "1", "2", "3", "4", and "5". Each section contains fields such as ID, Company, Last Name, First Name, E-mail Address, Job Title, Business Phone, Home Phone, Mobile Phone, Fax Number, Address, City, State/Province, ZIP/Postal Code, Country/Region, Web Page, Notes, and Attachments.

FIGURE 19.16 Placing an unbound text box control on the form.

11. In the form grid, click the unbound text box next to Current File. In the property sheet for this text box, click the All tab and type **txtCurrentFileName** in the Name property. Click the Format tab and change the Back Color property to any color you like.
12. In the form grid, click the **Current File** label. In the property sheet for the selected label control, click the All tab and type **lblCurrentFile** in the Name property.
13. In the form grid, click the **Attachments** label. In the property sheet for this label, click the All tab and type **lblTotalFiles** in the Name property.  
Now let's write an event procedure to display information about the attached file.
14. In the form grid, click the **Attachments** control. In the property sheet for this control, click the Event tab, then click the Browse button next to the **On Attachment Current** property. In the Choose Builder dialog box, select **Code Builder** and click OK.  
Access will write the stub of the **Attachments\_AttachmentCurrent** event procedure.
15. Complete the code of the **Attachments\_AttachmentCurrent** procedure as shown in Figure 19.17.



The screenshot shows the Microsoft Access VBA code editor window titled "Chap19 - Form\_Customers (Code)". The code module is named "Attachments". The code itself is as follows:

```
Option Compare Database
Option Explicit

Private Sub Attachments_AttachmentCurrent()
    If Me.DefaultView = 0 Then
        If Me.Attachments.AttachmentCount = 0 Then
            Me.txtCurrentFileName.Visible = False
            Me.lblCurrentFile.Visible = False
            Me.lblTotalFiles.Caption = "Attachments"
        Else
            Me.txtCurrentFileName.Visible = True
            Me.lblCurrentFile.Visible = True
            Me.txtCurrentFileName = Me.Attachments.FileName
            Me.lblTotalFiles.Caption = "Attachments (" & _
                Me.Attachments.AttachmentCount & ")"
        End If
    End If
End Sub
```

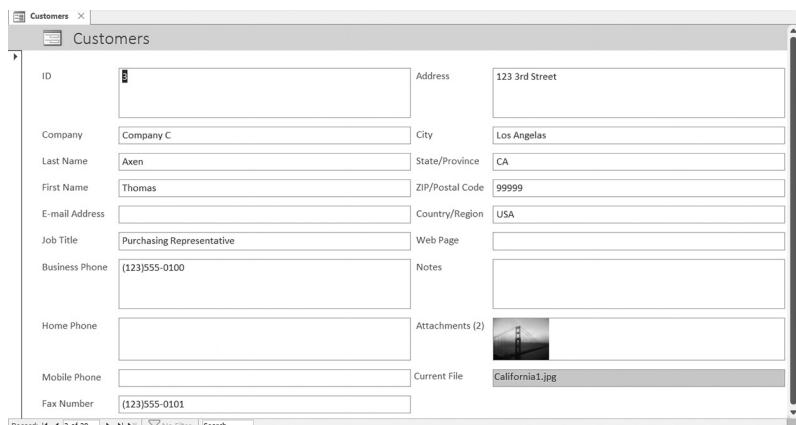
FIGURE 19.17. Use the AttachmentCurrent event procedure for the Attachments control to retrieve information about attachments and load it into your form's controls.

The Attachments control comes with special properties that apply to working with the Attachment data type. The **FileName** property returns the name of the attached file. If you need to display the file extension, use the **FileType**

property. The `AttachmentCount` property returns the number of attachments stored for the record.

The `Attachments` control has a special event called `AttachmentCurrent`. This event is similar to the form's `OnCurrent` event. It is triggered when you move the focus from one attachment to another. The code shown in Figure 19.17 begins by checking whether the form's default view is set to Single Form (0). If `DefaultView` is set to display other types of Access forms, the code in the event procedure will not run. The procedure hides the `txtCurrentFileName` text box control and its label `lblTotalFiles` for all records that do not have any attachments. This is done by setting the `Visible` property of the text box control and label control to False. Next, the procedure fills in the text boxes with the values retrieved from the `AttachmentCount` and `FileName` properties. Notice how the procedure manipulates the `Attachments` label control to display the total number of attachments for records that have them.

16. Press **Alt+F11** to return to the main Access window and activate the **Customers** form in Form view.
17. Scroll to the third customer record.
18. Notice that the `Attachments` label now shows the number of attached files in parentheses. There is also a text box below the attachment listing the current filename (see Figure 19.18). To scroll through the available files, select the `Attachments` field and click the Forward button in the tiny pop-up toolbar. Notice that when the new file loads into the `Attachments` control, the `Current File` box displays the name of the file being viewed.



**FIGURE 19.18.** The Current File text box control added to the form provides information about the attachment filename currently displayed in the `Attachments` control. The `Attachments` label has been modified to include the total number of attached files for records that contain attachments.

19. Press **Ctrl+S** to save changes to the Customers form, and then close this form.

## SUMMARY

---

This chapter presented a quick overview of types of forms you can create with Access 2021 and types of formatting you can apply to make your forms more attractive.

You learned how you can group form controls using the layouts, implement rich formatting in form controls, professionally format your forms using built-in themes, and enhance forms with images.

The chapter's main project focused on using the Attachments control in an Access form and showed you how to write an event procedure to display additional information about the attachments. You may want to treat it as a "warm-up" exercise for the next chapter, which gives you a complete overview and working knowledge of event procedures you can write for Access forms to change or enhance their default behavior.



# Chapter 20 *USING FORM EVENTS*

Chapter 1 provided a quick introduction to events, event properties, and event procedures as well as an example event procedure that changed the background color of a text box control placed on a form. Now is a good time to go back to the beginning of this book and review these topics. Here's a rundown of the terms you need to be familiar with:

- **Event**—Events are things that happen to an object. Events occur when you move a mouse, press a key, make changes to data, open a form, or add, modify, or delete a record, etc. An event can be triggered by the user or by the operating system.
- **Event property**—Forms, reports, and controls have various event properties you can use to trigger desired actions. When an event occurs, Microsoft Access runs a procedure assigned to an event property. Event properties are listed in the Event tab of the object's property sheet. The name of the event property begins with “On” and is followed by the event’s name. Therefore, the On Click event property corresponds to the Click event, and the On Got Focus event property is used for responding to the Got-Focus event.

- **Event procedure**—This is programming code you write to specify how a form, report, or control should respond to a particular event. By writing event procedures you can modify the application's built-in response to an event.
- **Event trapping**—When you assign programming code to an event property, you set an event trap. When you trap an event, you interrupt the default processing that Access would normally carry out in response to the user's keypress or mouse click.
- **Sequence of events**—Events occur in a predefined order. For example, the Click event occurs before the DoubleClick event. When you perform an action, several events occur, one after the other. For instance, the following form and control events occur when you open a form:

Open → Load → Resize → Activate → Current → Enter (control) → GotFocus (control)

Closing the form triggers the following control and form events:

Exit (control) → LostFocus (control) → Unload → Deactivate → Close

To find out whether a particular event is triggered in response to a user action, you may want to place the `MsgBox` statement inside the event procedure for the event you want to test.

Access forms, reports, and controls recognize various events.

Events can be organized by object (form, report, control) or by cause (what caused the event to happen). This chapter contains numerous examples of event procedures you can write to make your forms dynamic.

Access forms can respond to a variety of events. These events allow you to manage entire records and respond to changes in the data. You can determine what happens when records are added, changed, or deleted, or when a different record becomes current. You can decide how the form appears to the user when it is first displayed on the screen and what happens when the form is closed. You can also manage problems that occur when the data is unavailable. As you design your custom forms, you will find that some form events are used more frequently than others. The following sections provide hands-on examples of event procedures you can write for Access forms.

## DATA EVENTS

---

Data events occur when you change the data in a control or record placed on a form, or when you move the focus from one record to another.

## Current

The Current event occurs when the form is opened or re-queried and when the focus moves to a different record. Use the Current event to synchronize data among forms or move focus to a specific control.

The event procedure in Hands-On 20.1 sets the BackColor property of the form's header (Section 1) to red (255) for each discontinued product. The Form\_Current event will occur each time you move to a new record if the specified condition is true.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 20.1 Writing the Form\_Current Event Procedure

1. Start Access and create a new database named **Chap20.accdb** in your C:\VBAAccess2021\_ByExample folder.
2. Import all the tables, queries, forms, reports, macros, and modules from the Northwind.mdb sample database to your Chap20.accdb database. To do this, in the Access window, choose **External Data | Access**. In the File name box, type **C:\VBAAccess2021\_ByExample\Northwind.mdb** and click **OK**. In the Import Objects window, select the **Tables** tab and click the **Select All** button. This will highlight all the tables. Select the **Queries** tab and click the **Select All** button. Select the **Forms** tab and click the **Select All** button. Select the **Reports** tab and click the **Select All** button. Do the same for macros and modules. After selecting all the objects on the specified tabs, click **OK** to begin importing. Click the **Close** button when done. There is no need to save the Import steps.
3. In the Access window of the Chap20.accdb database, right-click on the **Products** form and choose **Design View**. Make sure the form's property sheet is visible and the Selection Type is set to Form. To activate the property sheet, choose **Property Sheet** in the Tools section of the Form Design tab.
4. In the form's property sheet, click the **Event** tab. Click next to the **On Current** event property and choose **[Event Procedure]** from the drop-down box. Click the **Build** button (...).
5. Complete the code of the **Form\_Current** event procedure as shown here:

```
Private Sub Form_Current()
```

```
Dim strPath As String
Dim strImage As String

strPath = "C:\VBAAccess2021_ByExample\External Docs\"
strImage = "Pinelumb.jpg"
If Discontinued = True Then
    Me.Section(1).BackColor = 255
    Me.Picture = ""
Else
    Me.Picture = strPath & strImage
End If
End Sub
```

6. To test this event procedure, activate the **Products** form that is currently open in Design view. You can quickly switch to the selected form from Visual Basic by clicking the View Object button in the Project Explorer window. Next, in the Access window, click the **View** button on the Ribbon to display the form in Form view. Use the record selectors to move to record 5. Because this record is marked as Discontinued, the code in the Form\_Current event will change the form header section's color to red (see Figure 20.1). The records that are not discontinued will appear with the background image specified in the `Else` clause.
7. Close the Products form and save all the changes when prompted.

The screenshot shows the Microsoft Access 'Products' form in Form view. The form has a dark grey header bar with the title 'Products'. Below the header, there are several input fields and dropdown menus for product details. The 'Discontinued' field is checked, which triggers the VBA code to change the header's background color to red. The form also includes buttons for previewing the product list and outputting it as HTML. At the bottom, there are navigation buttons for records and a search bar.

FIGURE 20.1 The Products form displays a red header background when a product is marked as Discontinued.

## BeforeInsert

The BeforeInsert event occurs when the first character is typed in a new record but before the new record is created. Use the BeforeInsert event to verify that the data is valid or to display information about data being added. This event is quite useful for placing default values in the fields at runtime. The BeforeInsert event can be canceled if the data being added does not meet specific criteria. The event procedure in Hands-On 20.2 demonstrates how to enter a default value in the Country field when a user begins to enter data in the form.

### ④ Hands-On 20.2 Writing the Form\_BeforeInsert Event Procedure

For this hands-on exercise, we will create a new form based on the Customers table.

1. Highlight the **Customers** table in the left pane of the Access window. Choose **Create | Form Wizard**.
2. Select the following fields: **CustomerID**, **CompanyName**, **Address**, **City**, **Region**, **PostalCode**, and **Country**. Step through the Form Wizard screens, pressing the **Next** button until you get to the screen where you are asked for the form's title. Type **New Customers** for the form's title, select the **Modify the form's design** option button, and click **Finish**.  
Access opens the New Customers form in Design view.
3. In the property sheet, select **Form** from the drop-down box, and click the **Data** tab. Set the Data Entry property to **Yes**.
4. In the form's property sheet, click the **Event** tab. Click next to the **Before Insert** event property and choose **[Event Procedure]** from the drop-down box. Click the **Build** button (...).  
Access opens the Visual Basic Editor window and writes the stub of the **Form\_BeforeInsert** event procedure.
5. Complete the code of the **Form\_BeforeInsert** event procedure as shown here:

```
Private Sub Form_BeforeInsert(Cancel As Integer)
    Me.Country = "USA"
End Sub
```

6. To test this event procedure, activate the **New Customers** form in Form view.
7. Type **JANIT** in the CustomerID field. Notice that as soon as you start filling in the form's text boxes, the text "USA" appears in the Country field.
8. Press the **Esc** key twice to undo the changes to the form.
9. Close the New Customers form and save all the changes when prompted.

### **AfterInsert**

---

The AfterInsert event occurs when a new record has been inserted. Use this event to re-query the recordset when a new record is added or to display other information. The event procedure in Hands-On 20.3 retrieves the total number of records in the Customers table after a new record has been inserted.



### **Hands-On 20.3 Writing the Form\_AfterInsert Event Procedure**

This hands-on exercise uses the New Customers form created in Hands-On 20.2.

1. In the Visual Basic Editor's Project Explorer window, double-click **Form\_New Customers**.
2. In the Code window, you will see the Form\_BeforeInsert event procedure prepared in Hands-On 20.2. Below this procedure code, enter the **Form\_AfterInsert** event procedure as shown here:

```
Private Sub Form_AfterInsert()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset

    Set db = CurrentDb()
    Set rst = db.OpenRecordset("Customers")

    MsgBox "Total Number of Records: " & _
        rst.RecordCount & "."

    rst.Close
    Set rst = Nothing
    Set db = Nothing
End Sub
```

3. To test this event procedure, open the **New Customers** form in Form view. Type TRYIT in the Customer ID text box and Test Events in the Company Name text box. Now use the record selector to move to the next record. Access executes the code in the Form\_AfterInsert event procedure and displays the total number of records.
4. Close the New Customers form and save all the changes if prompted.

### **BeforeUpdate**

---

The BeforeUpdate event occurs after a record has been edited but before it is written to the table. This event is triggered by moving to another record or

attempting to save the current record. The BeforeUpdate event takes place after the BeforeInsert event. Use this event to validate the entire record and display a message to confirm the change. The BeforeUpdate event can be canceled if the record cannot be accepted. The event procedure in Hands-On 20.4 will supply the value for the CustomerID field before the newly entered record is saved.



### Hands-On 20.4 Writing the Form\_BeforeUpdate Event Procedure

This hands-on exercise uses the New Customers form created in Hands-On 20.2.

1. In the Visual Basic Editor's Project Explorer window, double-click **Form\_New Customers**.
2. In the Code window, other event procedures prepared in Hands-On 20.2 and 20.3 will be listed. Enter the following **Form\_BeforeUpdate** event procedure below the code of the last procedure:

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
    If Not IsNull(Me.CompanyName) Then
        Me.CustomerID = Left(CompanyName, 3) & _
            Right(CompanyName, 2)
        MsgBox "You just added Customer ID: " & _
            Me.CustomerID
    Else
        MsgBox "Please enter Company Name.", _
            vbOKOnly, "Missing Data"
        Me.CompanyName.SetFocus
        Cancel = True
    End If
End Sub
```

3. To test this event procedure, open the **New Customers** form in Form view. Type **Event Enterprises** in the Company Name box. Click the record selector to move to the next record. The BeforeUpdate event procedure code will run at this point and you will see a message box with the custom-generated Customer ID. Click **OK** to the message. Another message will appear with the number of total records. This message box is generated by the AfterInsert event procedure that was prepared in Hands-On 20.3. Click **OK** to this message.
4. Close the New Customers form and save changes to the form if prompted.

### AfterUpdate

The AfterUpdate event occurs after the record changes have been saved in the database. It is also invoked when a control loses focus and after the data in the

control has changed. Use the `AfterUpdate` event to update data in other controls on the form or to move the focus to a different record or control. The event procedure in Hands-On 20.5 creates an audit trail for all newly added records, as illustrated in Figure 20.2.

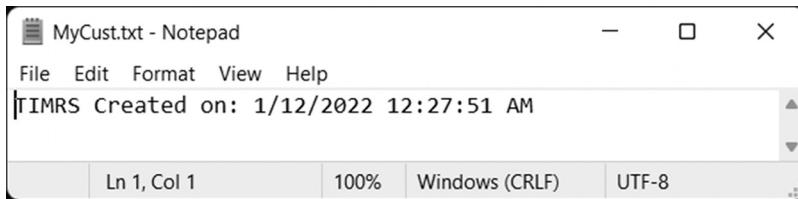


FIGURE 20.2. The `Form_AfterUpdate` event procedure is used here to store information about newly added records in a text file.



### Hands-On 20.5 Writing the `Form_AfterUpdate` Event Procedure

This hands-on exercise requires the New Customers form that was created in Hands-On 20.2.

1. In the Visual Basic Editor window, choose **Tools | References**. Locate and select **Microsoft Scripting Runtime** in the Available References list and click **OK**.
2. In the Project Explorer window, double-click **Form\_New Customers**.
3. Other procedures that were prepared so far in this chapter will be listed in the Code window. Enter the following `Form_AfterUpdate` event procedure below the code of the last procedure:

```
Private Sub Form_AfterUpdate()
    Dim fso As FileSystemObject
    Dim objFile As Object
    Dim strFileName As String
    Dim strPath As String
    Dim strFullPath As String

    On Error Resume Next

    strPath = "C:\VBAAccess2021_ByExample\
    strFileName = "MyCust.txt"
    strFullPath = strPath & strFileName

    Set fso = New FileSystemObject
    Set objFile = fso.GetFile(strFullPath)
```

```
If Err.Number = 0 Then
    ' open text file
    Set objFile = fso.OpenTextFile(strFullPath, 8)
Else
    ' create a text file
    Set objFile = fso.CreateTextFile(strFullPath)
End If

objFile.WriteLine UCase(Me.CustomerID) & _
    " Created on: " & Date & " " & Time
objFile.Close
Set fso = Nothing
MsgBox "This record was logged in: " & strFullPath
End Sub
```

This event procedure first checks whether the specified text file exists on your computer. If the file is found, then the `Err.Number` statement returns zero. At this point you want to open the file. The “8” represents the open mode for appending. Use “2” if you want to replace the contents of a file with the new data.

4. To test the event procedure, open the **New Customers** form in Form view. Type **Time Organizers** in the Company Name box. Click the record selector to move to the next record. The `BeforeUpdate` event procedure code you prepared in Hands-On 20.4 will run at this point and you should see a message box that displays the custom-generated Customer ID. Click **OK** to the message. The next message box notifies you about the location of the audit trail (the result of the `AfterUpdate` event procedure prepared in this exercise). Click **OK** to the message. Another message will appear with the number of total records. This message box is generated by the `AfterInsert` event procedure that was prepared in Hands-On 20.3. Click **OK** to this message.

As you enter more customer records using the New Customers form, events are executed in the following order:

BeforeInsert (Hands-On 20.2)  
BeforeUpdate (Hands-On 20.4)  
AfterUpdate (Hands-On 20.5)  
AfterInsert (Hands-On 20.3)

5. Close the New Customers form and save changes to the form if prompted.
6. Open the log file you created in this Hands-On and check its output with Figure 20.2.

## Dirty

The Dirty event occurs when the contents of a form or the text portion of a combo box changes. This event will be triggered by an attempt to enter a character directly in the form's text box or combo box. Use this event to determine if the record can be changed. The event procedure in Hands-On 20.6 disallows changes to form data when the CategoryID is less than or equal to 4.



### Hands-On 20.6 Writing the Form\_Dirty Event Procedure

1. Highlight the **Categories** table in the left pane of the Access window. Choose **Create | Form Wizard**.
2. Add all the fields as listed in the Categories table. Step through the Form Wizard screens, clicking the **Next** button until you get to the screen where you are asked for the form's title. Type **Product Categories** for the form's title, select the **Modify the form's design** option button, and click **Finish**.  
Access opens the Product Categories form in Design view.
3. In the property sheet, select **Form** from the drop-down box, and click the **Event** tab. Click next to the **On Dirty** event property and choose **[Event Procedure]** from the drop-down box. Click the **Build** button (...).  
Access opens the Visual Basic Editor window and writes the stub of the Form\_Dirty event procedure.
4. Complete the code of the **Form\_Dirty** event procedure as shown here:

```
Private Sub Form_Dirty(Cancel As Integer)
    If CategoryID <= 4 Then
        MsgBox "You cannot make changes in this record."
        Cancel = True
    End If
End Sub
```

5. To test this event procedure, open the **Product Categories** form in Form view. Try to make any changes to the original records. You will not be able to make changes to the data if the product's CategoryID is less than or equal to 4.
6. Close the Product Categories form and save changes to the form when prompted.

## OnUndo

---

The OnUndo event occurs when the user undoes a change to a combo box control, form, or text box control. By setting the `Cancel` argument to `True`, you can cancel the undo operation and leave the control or form in its edited state. The

Undo event for forms is triggered when the user clicks the Undo button, presses the Esc key, or calls the `Undo` method.

## Delete

The Delete event occurs when you select one or more records for deletion and before the records are actually removed from the table. Use this event to place restrictions on the data that can be deleted. When deleting multiple records, the Delete event occurs for each record. This enables you to confirm or cancel each deletion in your event procedure code. You can cancel the deletion in the Delete or BeforeDelConfirm events by setting the `Cancel` argument to `True`.

The event procedure in Hands-On 20.7 demonstrates how to disallow deletion of records when `CategoryID` is less than or equal to 8 and asks the user to confirm the deletion for other records.



### Hands-On 20.7 Writing the Form\_Delete Event Procedure

This hands-on exercise uses the Product Categories form created in Hands-On 20.6.

1. In the Visual Basic Editor's Project Explorer window, double-click **Form\_Product Categories**, which was created in Hands-On 20.6.
2. In the Code window, you will see the **Form\_Dirty** event procedure that was prepared in Hands-On 20.6. Below this procedure code, enter the **Form\_Delete** event procedure as shown here:

```
Private Sub Form_Delete(Cancel As Integer)
    If CategoryID <= 8 Then
        MsgBox "You can't delete the original categories."
        Cancel = True
    Else
        If MsgBox("Do you really want to delete " & _
                  "this record?", vbOKCancel, _
                  "Delete Verification") = vbCancel Then
            Cancel = True
        End If
    End If
End Sub
```

3. To test this event procedure, open the **Product Categories** form in Form view. Click on the record selector to the left of the first record and press the **Delete** key. At this point Access will execute the code of the **Form\_Delete** event procedure. You should see the message that you cannot delete original product categories.

4. Click the **New** button on the Ribbon to add a new record to the form. Enter a new category named **Organic Food** and save the record. Now press the **Delete** button on the Ribbon to delete this record. If there is no code in the **Form\_BeforeDelConfirm** event procedure (see Hands-On 20.8), you will be prompted twice to confirm the deletion. Go ahead with the deletion by clicking **OK** to the first message and **Yes** to the second.
5. Close the Product Categories form and save changes to the form when prompted.

### **BeforeDelConfirm**

---

The **BeforeDelConfirm** event occurs after the Delete event but before the Delete Confirm message box is displayed. If you don't write your own **BeforeDelConfirm** event, Access will display a standard delete confirmation message as described in Hands-On 20.7. You can use this event to write a custom deletion confirmation message. The event procedure in Hands-On 20.8 demonstrates how to suppress the default message.



### **Hands-On 20.8 Writing the Form\_BeforeDelConfirm Event Procedure**

This hands-on exercise uses the Product Categories form created in Hands-On 20.6.

1. In the Visual Basic Editor's Project Explorer window, double-click **Form\_Product Categories**, which was created in Hands-On 20.6 and modified in Hands-On 20.7.
2. In the Code window, two event procedures are shown that were prepared in Hands-On 20.6 and 20.7. Enter the following **Form\_BeforeDelConfirm** event procedure below the code of the last procedure:

```
Private Sub Form_BeforeDelConfirm(Cancel _  
As Integer, Response As Integer)  
    Response = acDataErrContinue  
End Sub
```

In this procedure code, the statement `Response = acDataErrContinue` will suppress the default message box that Microsoft Access normally displays when you attempt to delete a record.

3. To test this event procedure, open the **Product Categories** form in Form view. Click the **New** button on the Ribbon to add a new record, save it, and then delete it. The **Form\_Delete** event procedure prepared in Hands-On 20.7 will be executed at this point, and you will see a dialog with your custom prompt to confirm the deletion. Click **Yes**. Notice that Access does not display its default

- message asking you to confirm the deletion of the specified number of records.
4. Close the Product Categories form and save changes to the form when prompted.

<b>NOTE</b>	<p><i>Instead of writing your custom confirmation message in the Form_Delete event procedure, you can place it in the Form_BeforeDelConfirm event procedure as shown here:</i></p> <pre>Private Sub Form_BeforeDelConfirm(Cancel As Integer, _     Response As Integer)     ' remove the default Access message box     ' that prompts to confirm deletion     Response = acDataErrContinue     If MsgBox("Do you really want to delete this record?", _         vbOKCancel) = vbCancel Then         Cancel = True     End If End Sub</pre>
-------------	---

### AfterDelConfirm

The AfterDelConfirm event occurs after the record is deleted or after deletion is canceled in the BeforeDelConfirm event procedure. Use the AfterDelConfirm event to move to another record or to display a message indicating whether the deletion was successful. The `Status` argument allows you to check whether deletion progressed normally or was canceled by the user or Visual Basic. The following constants can be used for the `Status` argument in the AfterDelConfirm event procedure: `acDelete` (6), `acDeleteCancel` (1), `acDeleteOK` (0), or `acDeleteUserCancel` (2).

The event procedure in Hands-On 20.9 displays a message when a record is successfully deleted.



### Hands-On 20.9 Writing the Form\_AfterDelConfirm Event Procedure

This hands-on exercise uses the Product Categories form created in Hands-On 20.6.

1. In the Visual Basic Editor's Project Explorer window, double-click **Form\_Product Categories**.
2. The Code window appears with several event procedures that were prepared in previous hands-on exercises. Enter the following **Form\_AfterDelConfirm** event procedure below the code of the last procedure:

```
Private Sub Form_AfterDelConfirm(Status As Integer)
    MsgBox "The selected record was deleted."
    Debug.Print "Status = " & Status
End Sub
```

3. To test this event procedure, open the **Product Categories** form in Form view. Add a new record, save it, and delete it. Access will execute the code in the Form\_Delete event procedure (Hands-On 20.7) that displays a message box asking you whether you want to delete the record. Click **Yes**. Access will then check the code in Form\_BeforeDelConfirm (Hands-On 20.8). The statement Response = acDataErrContinue will cause Access to suppress its default Delete Confirm message box and you will not be prompted again to reconfirm the deletion. Finally, Form\_AfterDelConfirm will run and you will see a message about the successful deletion.
4. Close the Product Categories form and save changes to the form when prompted.

## FOCUS EVENTS

---

Focus events occur when a form becomes active or inactive and when a form or form control loses or gains the focus.

### Activate

---

The Activate event occurs whenever the form gains the focus and becomes the active window. This situation occurs when the form is first opened and when the user activates the form again by clicking on the form or one of its controls. Use this event to display or hide supporting forms.

The event procedure in Hands-On 20.10 will hide the tab labeled Personal Info when the Employees form is displayed. Notice that the tabs are numbered beginning with 0, hence the second tab in the tab control placed on the form has an index value of 1.



### Hands-On 20.10 Writing the Form\_Activate Event Procedure

1. In the Visual Basic Editor's Project Explorer window, double-click **Form\_Employees**.
2. The Code window contains several event procedures and functions already written for this form. Enter the following **Form\_Activate** event procedure below the code of the last procedure:

```
Private Sub Form_Activate()
    Me.TabCtl0.Pages(1).Visible = False
End Sub
```

3. To test this event procedure, open the **Employees** form in Form view. Notice that only the tab labeled Company Info is shown.
4. Close the Employees form and save changes to the form when prompted.

## Deactivate

---

The Deactivate event occurs when the user switches to another form or closes the form. Use this event to display or hide supporting forms. The event procedure in Hands-On 20.11 will display a message when the focus moves to a different form.

### Hands-On 20.11 Writing the Form\_Deactivate Event Procedure

1. In the Visual Basic Editor's Project Explorer window, double-click **Form\_Employees**.
2. The Code window contains several event procedures and functions already written for this form. Enter the following **Form\_Deactivate** event procedure below the code of the last procedure:

```
Private Sub Form_Deactivate()
    MsgBox "You are leaving the " & Me.Name & " form."
    If Me.Dirty Then
        DoCmd.Save acForm, Me.Name
        MsgBox "Your changes have been saved."
    End If
End Sub
```

3. To test this event procedure, open the **Products** form in Form view. Next, activate the **Employees** form in Form view and change the phone extension in the first employee record. Now go back to the Products form. You should get two messages as programmed in the Form\_Deactivate event procedure. Click **OK** to each message.
4. Close the Employees and the Products forms.

## GotFocus

---

The GotFocus event happens when a form receives the focus, if there are no visible or enabled controls on the form. The GotFocus event is frequently used for controls placed on the form and rarely used for the form itself.

### **LostFocus**

---

The LostFocus event happens when a form loses focus, provided there are no visible or enabled controls on the form. This event is frequently used for controls placed on the form and rarely used for the form itself.

## **MOUSE EVENTS**

---

Mouse events occur when you move a mouse or click any of the available mouse buttons.

### **Click**

---

The Click event occurs when you click a mouse button on a blank area of a form, a form's record selector, or a control placed on the form.

The event procedure in Hands-On 20.12 will cause a text box control to move one inch to the right when you click the record selector.



### **Hands-On 20.12 Writing the Form\_Click Event Procedure**

1. Create a new form with two text boxes. Position both text boxes starting at 1 inch on the horizontal ruler. Save the form as **Mouse Test**.
2. In the form's property sheet, make sure **Form** is selected and click the **Event** tab. Click next to the **On Click** event property and choose **[Event Procedure]** from the drop-down box. Click the **Build** button (...).  
Access opens the Visual Basic Editor window and writes the stub of the **Form\_Click** event procedure.
3. Complete the code of the **Form\_Click** event procedure as shown here:

```
Private Sub Form_Click()  
    MsgBox "Form Click Event Occurred."  
    Me.Text0.Left = Text0.Left + 1440  
End Sub
```

The first text box control placed on the form is automatically named **Text0**. The **Left** property is used to specify an object's location on a form or report. This procedure moves a text box control one inch to the right. Screen measurements are expressed in units called *twips*, and there are 1440 twips per inch. Thus, to calculate the new position of the text box, you must add 1440 to the current position.

4. To test this event procedure, open the **Mouse Test** form in Form view. Click on the record selector (a bar to the left of a record). This will cause the Form\_Click event procedure code to execute and you will see a message box. After clicking **OK** in response to the message, the first text box control will move one inch to the right as illustrated in Figure 20.3.
5. Close the Mouse Test form and save changes to the form when prompted.



FIGURE 20.3. The Form\_Click event procedure has moved the first text box to the right.

## DblClick

The DblClick event occurs when you double-click on a blank area of the form, the form's record selector, or a control placed on the form.

## MouseDown

The MouseDown event occurs when you click and hold on a blank area of the form, the form's record selector, or a control placed on the form. This event occurs before the Click event. The MouseDown event has four arguments:

- **Button**—Identifies the state of the mouse buttons. Use `acLeftButton` to check for the left mouse button, `acRightButton` to check for the right mouse button, and `acMiddleButton` to check for the middle mouse button.
- **Shift**—Specifies the state of the Shift, Ctrl, and Alt keys when the button specified by the `Button` argument was pressed or released. Use `acShiftMask` (1) to test for the Shift key, `acCtrlMask` (2) to test for the Ctrl key, and `acAltMask` (4) to test for the Alt key. You can test for any combination of buttons. For example, to specify that Ctrl and Alt were pressed, use the value of 6 (2+4) as the `Shift` argument.

- **X**—Specifies the horizontal (x) position from the left edge of the form or control.
- **Y**—Specifies the vertical (y) position from the top edge of the form or control.

The event procedure in Hands-On 20.13 displays two messages when the form's **MouseDown** event is fired. The first message tells whether you pressed the Alt, Ctrl, or Shift key, and the second one announces which mouse button was used.



### Hands-On 20.13 Writing the Form\_MouseDown Event Procedure

1. Create a new form based on the Products table adding all the available fields to the form. Save this form as **Products Test**.
2. In the form's property sheet, make sure **Form** is selected and click the **Event** tab. Click next to the **On Mouse Down** event property and choose **[Event Procedure]** from the drop-down box. Click the **Build** button (...).
3. Enter the following code in the **Form\_MouseDown** event:

```
Private Sub Form_MouseDown(Button As Integer, _
    Shift As Integer, _
    X As Single, _
    Y As Single)
    Debug.Print "Mouse Down"

    Select Case Shift
        Case 0
            MsgBox "You did not press a key."
        Case 1 ' or acShiftMask
            MsgBox "You pressed SHIFT."
        Case 2 ' or acCtrlMask
            MsgBox "You pressed CTRL."
        Case 3
            MsgBox "You pressed CTRL and SHIFT."
        Case 4 ' or acAltMask
            MsgBox "You pressed ALT."
        Case 5
            MsgBox "You pressed ALT and SHIFT."
        Case 6
            MsgBox "You pressed CTRL and ALT."
        Case 7
            MsgBox "You pressed CTRL, ALT, and SHIFT."
    End Select

    If Button = 1 Then ' acLeftButton
```

```
    MsgBox "You pressed the left button."
    ElseIf Button = 2 Then ' acRightButton
        MsgBox "You pressed the right button."
    ElseIf Button = 4 Then ' acMiddleButton
        MsgBox "You pressed the middle button."
    End If
End Sub
```

4. To test this event procedure, switch to the **Products Test** form and open it in Form view. Click on the record selector while holding down any mouse button and pressing the Shift, Ctrl, or Alt keys or combinations of these keys.
5. Close the Products Test form and save changes to the form when prompted.

### MouseMove

---

The MouseMove event occurs when you move the mouse over a blank area of the form, the form's record selector, or a control placed on the form. The MouseMove event occurs before the Click event and has the same arguments as the MouseDown event.

### MouseUp

---

The MouseUp event occurs when you release the mouse button. It occurs before the Click event and uses the same arguments as the MouseDown and MouseMove events.

### MouseWheel

---

The MouseWheel event occurs in Form view or Datasheet view when the user rotates the mouse wheel on a mouse device that has a wheel. This event takes the following two arguments:

- **Page**—Returns True if the page was changed.
- **Count**—Specifies the number of lines that were scrolled with the mouse wheel.

<b>NOTE</b>	<i>Because there is no Cancel argument, you cannot use the MouseWheel event to prevent users from using the mouse wheel to scroll through records on a form.</i>
-------------	--

## KEYBOARD EVENTS

---

Keyboard events occur when you hold down, press, or release a key on the keyboard or send a keystroke by using the `SendKeys` statement in Visual Basic or the `SendKeys` action in a macro.

The keyboard events occur in the following sequence:

`KeyDown` → `KeyPress` → `KeyUp`

If the form's `KeyPreview` property is set to `Yes`, all keyboard events occur first for the form, and then for the control that has the focus. When you press and hold down the key, the `KeyDown` and `KeyPress` events occur repeatedly. When you release the key, the `KeyUp` event occurs.

### **KeyDown**

---

The `KeyDown` event occurs when you press a key while a form or control has the focus. This event is also triggered by using the `SendKeys` statement in Visual Basic or the `SendKeys` action in a macro. If the form's `KeyPreview` property is set to `Yes`, all keyboard events occur first for the form, and then for the control that has the focus.

The `KeyDown` event takes the following two arguments:

- **KeyCode**—Determines which key was pressed. To specify keycodes, use members of the `KeyCodeConstants` class in the VBA Object Library in the Object Browser. To prevent an object from receiving the keystroke, set `KeyCode` to zero (0).
- **Shift**—Determines if the Shift, Ctrl, or Alt key was pressed. Use `acShiftMask(1)` to test for the Shift key, `acCtrlMask(2)` to test for the Ctrl key, and `acAltMask(4)` to test for the Alt key. You can test for any combination of buttons. For example, to specify that Ctrl and Alt were pressed, use the value of 6 (2+4) as the `Shift` argument.

The event procedure in Hands-On 20.14 displays a message when you press one of the following keys: F1, Home, Tab, Shift, Ctrl, Alt, or Delete.



### **Hands-On 20.14 Writing the Form\_KeyDown Event Procedure**

1. Open the **Products** form in Design view. In the form's property sheet, make sure **Form** is selected and click the **Event** tab. Set the **Key Preview** property to **Yes**.
2. Save the **Products** form.

3. Click next to the **On Key Down** event property and choose **[Event Procedure]** from the drop-down box. Click the **Build** button (...).

In the Code window there are a couple of event procedures already written for this form. Access adds the stub of the Form\_KeyDown event procedure for you.

4. Enter the following **Form\_KeyDown** event procedure code.

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    Select Case KeyCode
        Case vbKeyF1
            MsgBox "You pressed the F1 key."
        Case vbKeyHome
            MsgBox "You pressed the Home key."
        Case vbKeyTab
            MsgBox "You pressed the Tab key."
    End Select

    Select Case Shift
        Case acShiftMask
            MsgBox "You pressed the SHIFT key."
        Case acCtrlMask
            MsgBox "You pressed the CTRL key."
        Case acAltMask
            MsgBox "You pressed the ALT key."
    End Select

    If KeyCode = vbKeyDelete Then
        MsgBox "Delete Key is not allowed."
        KeyCode = 0
    End If
End Sub
```

5. To test this event procedure, open the **Products** form in Form view. Press one of the following keys: **F1**, **Home**, **Tab**, **Shift**, **Ctrl**, **Alt**, or **Delete**. Click **OK** to the message.
6. Close the Products form and save changes to the form when prompted.

### KeyPress

---

The KeyPress event occurs when you press and release a key or a key combination. This event is also triggered by using the `SendKeys` statement in Visual Basic or the `SendKeys` action in a macro. If the form's `KeyPreview` property is set to `Yes`, all keyboard events occur first for the form, and then for the control that has the focus.

The KeyPress event responds only to the ANSI characters generated by the keyboard, the Ctrl key combined with a character from the standard alphabet or a special character, and the Enter or Backspace key. Other keystrokes are handled by the KeyDown and KeyUp event procedures. `KeyAscii` is a read/write argument that specifies which ANSI key was pressed. To cancel the keystroke in the KeyPress event, set the `KeyAscii` argument to 0. The KeyPress event treats uppercase and lowercase letters as different characters.

The event procedure in Hands-On 20.15 prints the ASCII code and the value of the pressed key to the Immediate window. Upon pressing the Escape key (`KeyAscii=27`), the user is prompted to save changes. Clicking Yes to the message will cause the form to be closed. All other keystrokes are ignored.



### Hands-On 20.15 Writing the Form\_KeyPress Event Procedure

1. Open the **Suppliers** form in Design view. In the property sheet, make sure that **Form** is selected and click the **Event** tab. Set the **Key Preview** property to **Yes**. Click next to the **On keypress** event property and choose **[Event Procedure]** from the drop-down box. Click the **Build** button (...).  
In the Code window there are a couple of event procedures already written for this form. Access adds the stub of the Form\_KeyPress event procedure for you.
2. Enter the following **Form\_KeyPress** event procedure code.

```
Private Sub Form_KeyPress (KeyAscii As Integer)
    Debug.Print "keypress: KeyAscii = " & KeyAscii & _
        Space(1) & "=" & Chr(KeyAscii)
    If KeyAscii = 27 Then
        If MsgBox("Save changes to this form?", _
            vbYesNo) = vbYes Then
            DoCmd.Close acForm, Me.Name, acSaveYes
        Else
            KeyAscii = 0
        End If
    Else
        KeyAscii = 0
    End If
End Sub
```

The statement `KeyAscii = 0` will disable any input to all the controls on the form. Recall that a form's Key Preview property determines whether form keyboard events are invoked before control keyboard events. To prevent keystrokes from going to the form's controls, the KeyPreview property must be set to Yes.

Note that the KeyPress event is not triggered by the Delete key. You can delete any data on this form if there is no custom VBA code written in the KeyDown or KeyUp event procedure that blocks the use of this key.

3. To test this event procedure, open the **Suppliers** form in Form view. Try to edit a field by typing some text. Because the input to all the controls on the form has been disabled by the Form\_KeyPress event procedure, you cannot see any input. However, when you switch to the Immediate window, you will see the complete listing of keys that you pressed. Switch back to the Suppliers form and press the **Escape** key. If you agree to save changes to this form, the form will be closed.

### **KeyUp**

---

The KeyUp event occurs when you release a key while a form or control has the focus. This event is also triggered by using the `SendKeys` statement in Visual Basic or the `SendKeys` action in a macro. If the form's `KeyPreview` property is set to Yes, all keyboard events occur first for the form, and then for the control that has the focus.

The KeyUp event takes the following two arguments:

- **KeyCode**—Determines which key was pressed. To specify keycodes, use members of the `KeyCodeConstants` class in the VBA Object Library in the Object Browser. To prevent an object from receiving the keystroke, set `KeyCode` to zero (0).
- **Shift**—Determines if the Shift, Ctrl, or Alt key was pressed. Use `acShiftMask` (1) to test for the Shift key, `acCtrlMask` (2) to test for the Ctrl key, and `acAltMask` (4) to test for the Alt key. You can test for any combination of buttons. For example, to specify that Ctrl and Alt were pressed, use the value of 6 (2+4) as the `Shift` argument.

The event procedure in Hands-On 20.16 will print to the Immediate window the keycode and the value of the key that was released. Also, the information about `KeyCode` and the state of the Shift key will be shown in the form's caption.



### **Hands-On 20.16 Writing the Form\_KeyUp Event Procedure**

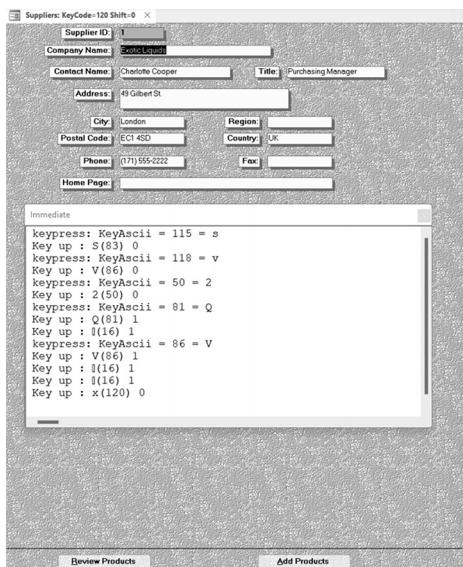
1. Open the **Suppliers** form in Design view. In the form's property sheet, make sure **Form** is selected and click the **Event** tab. Make sure the **Key Preview** property is set to **Yes**.
2. Click next to the **On Key Up** event property and choose **[Event Procedure]** from the drop-down box. Click the **Build** button (...).

The Code window contains a couple of event procedures already written for this form. Access adds the stub of the Form\_KeyUp event procedure for you.

3. Enter the following code for the **Form\_KeyUp** event procedure:

```
Private Sub Form_KeyUp(KeyCode As Integer, _
    Shift As Integer)
    Debug.Print "Key up : " & Chr(KeyCode) & _
    "(" & KeyCode & ")" & _
    Shift
    Me.Caption = Me.Name
    Me.Caption = Me.Caption & ":" & KeyCode = " & _
    KeyCode & " " & "Shift=" & Shift
End Sub
```

4. To test this event procedure, open the **Suppliers** form in Form view. Press various keys on the keyboard and notice the key information in the form's caption (see Figure 20.4).
5. Switch to the Visual Basic Editor window and activate the Immediate window.  
You should see a listing of the keys that were pressed and released while performing step 4.
6. Close the Suppliers form and save changes to the form when prompted.



**FIGURE 20.4.** The Form\_KeyUp event procedure writes out the Key information to the form's caption and Immediate window.

## ERROR EVENTS

---

The Error event is triggered by runtime errors generated either in the Access interface or by the Microsoft Jet/ACE database engine. The Error event does not trap VBA errors.

### Error

---

The Error event occurs when there is a problem accessing data for the form. Use this event to suppress the standard error messages and display a custom error message instead.

The Error event takes the following two arguments:

- **DataErr**—Contains the number of the Microsoft Access error that occurred.
- **Response**—Determines whether error messages should be displayed. It may be one of the following constants:
  - **acDataErrContinue**—Ignore the error and continue without displaying the default Microsoft Access error message.
  - **acDataErrDisplay**—Display the default Microsoft Access error message. This is the default.

The event procedure in Hands-On 20.17 displays a custom message when an attempt is made to add a new record with a customer ID that already exists. The standard Microsoft Access error message is not displayed.



### Hands-On 20.17 Writing the Form\_Error Event Procedure

1. Create a new form based on the Customers table. Add all the fields from the Customers table and save the new form as **Customers Data Entry**.
1. Activate the **Customers Data Entry** form in Design view. In the property sheet, make sure **Form** is selected and click the **Data** tab. Set the form's **DataEntry** property to **Yes**.
2. Click the **Event** tab, set the **On Error** property to **[Event Procedure]**, and press the **Build** button (...).  
Access will create the event procedure stub.
3. Enter the following **Form\_Error** event procedure:

```
Private Sub Form_Error(DataErr As Integer, _  
    Response As Integer)
```

```
Dim strMsg As String
Dim custId As String

Const conDuplicateKey = 3022
custId = Me.CustomerID

If DataErr = conDuplicateKey Then
    ' Don't show built-in error messages
    Response = acDataErrContinue
    strMsg = "Customer " & custId & " already exists."
    ' Show a custom error message
    MsgBox strMsg, vbCritical, "Duplicate Value"
End If
End Sub
```

4. Open the **Customers Data Entry** form in Form view. Enter **ALFKI** in the CustomerID field and **Alfred Fiki** in the Company Name field. Click the **Save** button. When you try to save this record, the Form\_Error event procedure code will cause a message box to appear, saying that the customer already exists. Click **OK** to the message. Press **Esc** to cancel the changes to this record.
5. Close the Customers Data Entry form and save the changes to the form.

## FILTER EVENTS

---

Filter events are triggered by opening or closing a filter window or when you are applying or removing a filter.

### Filter

---

The Filter event occurs when you design a filter to limit the form's records to those matching specified criteria. This event takes place when you select the Filter by Form or Advanced Filter/Sort options. Use this event to remove the filter that was previously set, to enter initial settings for the filter, or to call your own custom filter dialog box. To cancel the filtering command, set the **Cancel** argument for the event procedure to **True**.

The event procedure in Hands-On 20.18 allows the use of the Filter by Form option but disallows the use of the Advanced Filter/Sort option.



### Hands-On 20.18 Writing the Form\_Filter Event Procedure

This hands-on exercise uses the Product Categories form created in Hands-On 20.6.

1. In the Visual Basic Editor's Project Explorer window, double-click **Form\_Product Categories**.
2. The Code window shows other event procedures already written for this form. Enter the following **Form\_Filter** event procedure below the code of the last procedure:

```
Private Sub Form_Filter(Cancel As Integer, _
    FilterType As Integer)
    Select Case FilterType
        Case acFilterByForm
            MsgBox "You selected to filter records " & _
                "by form.", vbOKOnly + vbInformation, _
                "Filter By Form"
            Me.CategoryName.SetFocus
            Me.CategoryID.Enabled = False
        Case acFilterAdvanced
            MsgBox "You are not authorized to use " & _
                " Advanced Filter/Sort.", _
                vbOKOnly + vbInformation, _
                "Advanced Filter By Form"
            Cancel = True
    End Select
End Sub
```

3. To test this event procedure, open the **Product Categories** form in Form view.
4. In the Sort & Filter area of the Ribbon, choose **Advanced | Filter by Form**. The code in the Form\_Filter event procedure runs and you will see a message box. Click **OK**. The Filter by Form dialog box appears with the Category ID text box disabled. You can disable certain controls on the form if you don't want the user to filter by them.
5. Filter the form to display only records for **Seafood or Meat/Poultry**. Be sure to click **Toggle Filter** in the Sort & Filter area of the Ribbon after setting up filter criteria.
6. Now, remove the filter by clicking **Toggle Filter** again.
7. Choose **Advanced | Advanced Filter/Sort**. You will not be able to use the advanced filter for this form because the form's Filter event has disabled this action.
8. Close the Product Categories form and save changes to the form when prompted.

## ApplyFilter

The **ApplyFilter** event occurs when you apply the filter to restrict the records. This event takes place when you select the Apply Filter/Sort, Filter by Selection, or Remove Filter/Sort options. Use this event to change the form display before the filter is applied or undo any changes made when the Filter event occurred.

The **ApplyType** argument can be one of the predefined constants shown in Table 20.1.

**TABLE 20.1** *ApplyType* argument constants

Constant Name	Constant Value
acShowAllRecords	0
acApplyFilter	1
acCloseFilterWindow	2
acApplyServerFilter	3
acCloseServerFilterWindow	4

The event procedure in Hands-On 20.19 displays a different message depending on whether the user has made a selection in the Filter by Form dialog box.



### Hands-On 20.19 Writing the Form\_ApplyFilter Event Procedure

This hands-on exercise uses the Product Categories form created in Hands-On 20.6.

1. In the Visual Basic Editor's Project Explorer window, double-click **Form\_Product Categories**.
2. The Code window shows other event procedures already written for this form. Enter the following **Form\_ApplyFilter** event procedure below the code of the last procedure:

```
Private Sub Form_ApplyFilter(Cancel As Integer, _
    ApplyType As Integer)

    Dim Response As Integer

    If ApplyType = acApplyFilter Then
        If Me.Filter = "" Then
            MsgBox "You did not select any criteria.", _
                vbOKOnly + vbCritical, "No Selection"
            GoTo ExitHere
        End If
        Response = MsgBox("The selected criteria " & _

```

```
"is as follows:" & vbCrLf &_
Me.Filter, vbOKCancel + vbQuestion, _
"Filter Criteria")
End If

If Response = vbCancel Then
    Cancel = True
End If
If ApplyType = acShowAllRecords Then
    Me.Filter = ""
    MsgBox "Filter was removed."
End If
If ApplyType = acCloseFilterWindow Then
    Response = MsgBox("Are you sure you " & _
        "want to close the Filter window?", vbYesNo)
    If Response = vbNo Then
        Cancel = True
    End If
End If
ExitHere:
With Me.CategoryID
    .Enabled = True
    .SetFocus
End With
End Sub
```

3. To test this event procedure, open the **Product Categories** form in Form view. From the Sort & Filter area of the Ribbon, choose **Advanced | Filter by Form**. The Form\_Filter event will be triggered (see Hands-On 20.18). Click **OK** to the message box.
4. Select a category from the **Category Name** combo box and click **Toggle Filter** on the Ribbon. This action will trigger the Form\_ApplyFilter event procedure. Experiment with the form filter, testing other situations such as clicking Toggle Filter when the filtering criteria were not specified or closing the Filter by Form dialog box.
5. Close the Product Categories form and save changes to the form when prompted.

## TIMING EVENTS

---

Timing events occur in response to a specified amount of time passing.

## Timer

The Timer event occurs when the form is opened. The duration of this event is determined by the value (milliseconds) entered in the `TimerInterval` property located on the Event tab of the form's property sheet. Use this event to display a splash screen when the database is opened. The Timer event is helpful in limiting the time the record remains locked in multiuser applications.

The event procedure in Hands-On 20.20 will flash the button's text, "Preview Product List" (or the entire button if you use the commented code instead). For the code to work, you must start the timer by changing the `TimerInterval` property from 0 (stopped) to the desired interval. A timer interval of 1,000 will invoke a timer event every second. The form's Load event procedure sets the form's `TimerInterval` property to 250, so the button text (or the entire button) is toggled once every quarter second. You may change the timer interval manually by typing the value next to the form's `TimerInterval` property in the property sheet or by placing the following statement in the `Form_Load` event:

```
Me.TimerInterval = 250
```



### Hands-On 20.20 Writing the Form\_Timer Event Procedure

1. In the Visual Basic Editor's Project Explorer window, double-click the **Products** form.
2. The Code window shows other event procedures already written for this form.
3. Enter the following **Form\_Timer** event procedure below the code of the last procedure:

```
Private Sub Form_Timer()
    Static OnOff As Integer

    If OnOff Then
        Me.PreviewReport.Caption = "Preview Product List"
        ' Me.PreviewReport.Visible = True
    Else
        Me.PreviewReport.Caption = ""
        ' Me.PreviewReport.Visible = False
    End If
    OnOff = Not OnOff
End Sub
```

4. Activate the **Products** form in Design view. In the property sheet, make sure **Form** is selected and click the **Event** tab. Enter **250** for the `TimerInterval` property.

5. Switch the form to Form view. Notice the flashing effect of the Preview Product List button's text.
6. Close the Products form and save changes to the form when prompted.

**NOTE**

*To make the entire button flash, uncomment the commented lines of code and comment the original lines. Next, open the Products form in Form view and notice that the entire button is now flashing.*

## EVENTS RECOGNIZED BY FORM SECTIONS

---

In addition to trapping events for the entire form, you can write event procedures for the following form sections: Detail, FormHeader, FormFooter, PageHeader, and PageFooter.

Form sections respond to the following events: Click, DblClick, MouseDown, MouseUp, andMouseMove.

### DblClick (Form Section Event)

---

The DblClick event occurs when you double-click inside the form's header or footer section.

The example procedure in Hands-On 20.21 demonstrates how to randomly change the background color for each of the form's sections every time you double-click anywhere within the form's Detail section.



#### Hands-On 20.21 Writing the Detail\_DblClick Event Procedure

In the Navigation pane of the Chap20.accdb database, open the **Product Categories** form in Design view. Recall that you created this form in Hands-On 20.6.

1. Increase the size of the header and footer so that they are visible when you run the form.
2. In the property sheet, choose **Detail** from the drop-down box. Click the **Event** tab and select **[Event Procedure]** next to the **DblClick** property name. Click the **Build** button (...).
3. In the Code window, you should have the stub of the **Detail\_DblClick** event procedure already written for you. Complete this procedure as shown here:

```
Private Sub Detail_DblClick(Cancel As Integer)
    With Me
        .Section(acHeader).BackColor = _
```

```

RGB(Rnd * 128, _
Rnd * 256, _
Rnd * 255)
.Section(acDetail).BackColor = _
RGB(Rnd * 128, _
Rnd * 256, _
Rnd * 255)
.Section(acFooter).BackColor = _
RGB(Rnd * 128, _
Rnd * 256, _
Rnd * 255)
End With
End Sub

```

4. To test this event procedure, open the **Product Categories** form in Form view. Double-click anywhere in the Detail section of the form and see the colors of the Detail, Header, and Footer sections change.
5. Close the Chap20.accdb database.

## **UNDERSTANDING AND USING THE OPENARGS PROPERTY**

It's been over a decade since Microsoft introduced in Access an extremely useful property of the Form and Report objects called `OpenArgs`. Using the `OpenArgs` property you can pass parameters to the form or report when you open it with the `DoCmd` command. The `OpenArgs` property also comes in handy when:

- You want to pass values from one form to another
- You want to move the focus to a specific record when the form opens
- You want to automatically populate a control on the form
- You want to restrict access to certain forms

**NOTE**

*To use the `OpenArgs` property with the Access reports, turn to Chapter 22.*

The `OpenArgs` property is a string expression. It can be used both in macros and in VBA code. Only one `OpenArgs` string can be used in the `OpenForm` or `OpenReport` command; however, by combining values into one string separated by a unique character and using the built-in VBA `Split`

function, you can overcome this limitation. Before we delve into a practical example, let's look at the complete syntax of the `OpenForm` method:

```
DoCmd.OpenForm FormName, View, FilterName, WhereCondition, DataMode, WindowMode, OpenArgs
```

The parameter definitions are listed in Table 20.2.

**TABLE 20.2** Parameters used with the OpenForm method of the DoCmd object.

Parameter Name	Data Type	Description
<b>FormName</b> (This parameter is required.)	Variant	A string expression containing the name of a form in the current database.
<b>View</b>	acFormView	The acFormView constant specifies the view in which the form should open. The default is acNormal.
<b>FilterName</b>	Variant	A string expression containing the name of a query in the current database.
<b>WhereCondition</b>	Variant	A string expression containing the SQL WHERE clause without the word WHERE.
<b>DataMode</b>	acFormOpenDataMode	An acFormOpenDataMode constant specifies the data entry mode for the form and applies only to forms open in the Form view or Datasheet view. The default is acFormPropertySettings.
<b>WindowMode</b>	acWindowMode	An acWindowMode constant specifies the window mode in which the form opens. The default is acWindowNormal.
<b>OpenArgs</b>	Variant	A string expression used to set the form's OpenArgs property in a VBA code or in a macro.

The Hands-On 20.22 demonstrates the use the OpenArgs property for passing values from a custom form (frmOpenArgs) to the Northwind 2007 database built-in form (Employee List).

### **Hands-On 20.22 Passing Values to a Form Using the OpenArgs Property**

1. Copy the **Northwind 2007\_Revised.accdb** database from the companion files to your **VBAAccess2021\_ByExample** folder.

2. Open the **Northwind 2007\_Revised.accdb** database. Cancel the login dialog box upon loading of the database.
3. In the Navigation pane on the left, double click the **frmOpenArgs** to open it in Form view (see Figure 20.5).
4. Select the last value from the drop-down box and click the Execute button. Access displays the Employee List form as shown in Figure 20.5.
5. Switch to the Visual Basic Editor window and analyze the VBA code in the Form\_Employee List form class module, Form\_frmEmpOpenArgs form class module, and in the OpenArgs\_Demo module. Use the debugging techniques that you acquired earlier in this book to step line by line through the code sections.

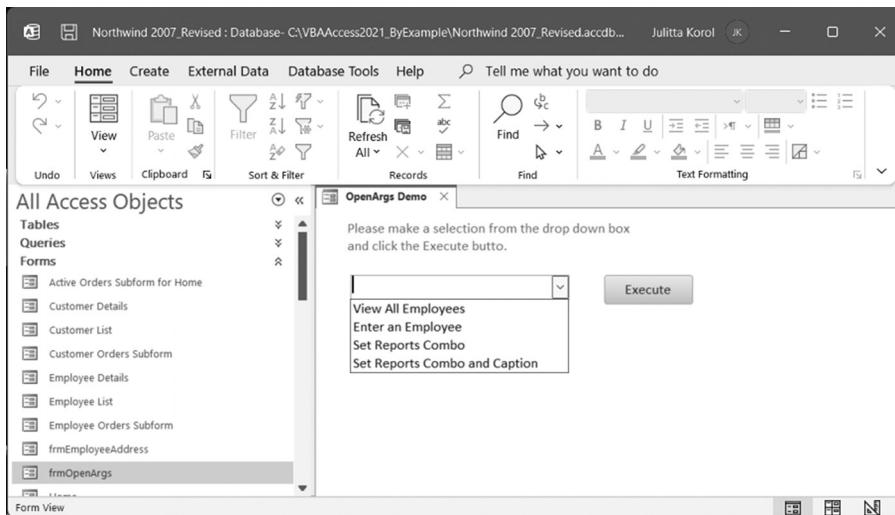


FIGURE 20.5 Working with the OpenArgs Demo (frmOpenArgs form).

Notice that the example form contains a combo box control with four items. Every time you select an item from the combo box and click the Execute button, an Employee List form is loaded with a slightly different effect. The code attached to the click event of the Execute button is shown below:

```

Private Sub cmdOpenEmpList_Click()
On Error GoTo Err_cmdOpenEmpList_Click

Dim strFormToOpen As String
Dim strUserSelection As String
strFormToOpen = "Employee List"

```

```
If IsOpenForm(strFormToOpen) Then
    DoCmd.Close acForm, strFormToOpen
    DoEvents
End If

If Not IsNull(cboSelection) Then
    strUserSelection = cboSelection.Value

Select Case cboSelection
    Case "View All Employees"
        DoCmd.OpenForm FormName:=strFormToOpen, _
            View:=acNormal, WindowMode:=acWindowNormal, _
            OpenArgs:=strUserSelection
    Case "Enter an Employee"
        DoCmd.OpenForm FormName:=strFormToOpen, _
            View:=acNormal, DataMode:=acFormAdd, _
            OpenArgs:=strUserSelection
    Case "Set Reports Combo"
        DoCmd.OpenForm strFormToOpen, acNormal, _
            , , , acWindowNormal, "Customer Address Book"
    Case "Set Reports Combo and Caption"
        DoCmd.OpenForm strFormToOpen, acNormal, _
            , , , acWindowNormal, _
            Me.Name & "|" & "Customer Phone Book"

End Select
Else
    MsgBox "Please make a selection from the combo box."
End If
Exit_cmdOpenEmpList_Click:
    Exit Sub
Err_cmdOpenEmpList_Click:
    MsgBox Err.Description
    Resume Exit_cmdOpenEmpList_Click
End Sub
```

Notice how this event procedure uses the `OpenArgs` property of the form to send different values to the Employee List form. To open a form, we simply use the `OpenForm` method of the `DoCmd` object and pass the name of the form as well as other parameters that define the type of view, data mode, window mode, and the `OpenArgs`. The parameters used with the `DoCmd` object are listed in Table 20.2. These parameters can be passed by name (as shown in the first two `Select Case` statements, or in line (as shown in the last two `Select Case` statements).

The Form\_Load event procedure of the Employee List form reads the values placed in the OpenArgs property and makes changes to the specified form controls:

```

Private Sub Form_Load()
    Dim aArgs() As String
    Dim counter As Integer

    If Not IsNull(Me.OpenArgs) Then
        If Me.OpenArgs = "Customer Address Book" Then
            Me.cboReports = Me.OpenArgs
            Me.cboReports.Width = 2800
            Exit Sub
        End If

        If DelimFound(Me.OpenArgs, "|") Then
            MsgBox "Passing multiple values."
            aArgs() = Split(Me.OpenArgs, "|")
            For counter = 0 To UBound(aArgs)
                If aArgs(counter) = "frmOpenArgs" Then
                    Me.Auto_Title0.Caption =
                        Me.Auto_Title0.Caption & _
                        " called from " & aArgs(counter)
                End If
                If aArgs(counter) = "Customer Phone Book" Then
                    Me.cboReports = aArgs(counter)
                    Me.cboReports.Width = 2800
                End If
                Debug.Print counter & ":" & aArgs(counter)
            Next counter
        Else
            Me.Auto_Title0.Caption = "Employee List"
        End If
    End If
End Sub

Private Sub Form_Unload(Cancel As Integer)
    Me.Auto_Title0.Caption = "Employee List"
End Sub

```

This procedure begins by checking whether the OpenArgs property contains any values. If the property is not Null, Access will run the remaining code prior to loading the form. Notice that to determine whether the

`OpenArgs` property is passing more than one value, we make a call to the custom `DelimFound` function (see the second code excerpt below). We pass two values to the `DelimFound` function. The first value is the contents of the `OpenArgs` property; the second value is the delimiter. In this example, we are using the Pipe character ( | ) as the delimiter. If the delimiter is found, we need to extract the values from the `OpenArgs` property by using the `Split` function:

```
aArgs() = Split(Me.OpenArgs, "|")
```

The extracted values are stored in the `aArgs` array variable (refer to Chapter 7 if you are new to arrays). The For...Next loop is then used to iterate through the array and assign the values to the form controls. In this process we assign corresponding values to the `Auto_Title0` and the `cboReports` controls.

In the `Form_Unload` event procedure, we make sure that the form's caption is reset to the "Employee List."

The supplemental function procedures that the `Click` event procedure and the `Load` event procedure call are placed in a standard module called `OpenArgs_Demo`.

The `IsOpenForm` function procedure returns `true` if the Employee List form is open and `false` if it is closed. If the form is open, the `cmdOpenEmpList_Click` event procedure will close it prior to executing the remaining code.

```
Function IsOpenForm(strFormName As String) _
As Boolean

    IsOpenForm = Application.CurrentProject. _
        AllForms(strFormName).IsLoaded

End Function
```

The `DelimFound` function checks if the specified delimiter can be found in the string passed in the `OpenArgs` property. This is done by using the built-in `InStr` function.

```
Function DelimFound(strOpenArgs As String, _
    strDelim As String) As Boolean
    If InStr(1, strOpenArgs, strDelim) Then
        DelimFound = True
    Else
        DelimFound = False
```

```

End If

End Function

```

ID	First Name	Last Name	E-mail Address	Business Phone	Company	Job Title
2	Andrew	Cencini	andrew@northwindtraders.com	(123)555-0100	Northwind Trader	Vice President, Sales
1	Nancy	Freehafer	nancy@northwindtraders.com	(123)555-0100	Northwind Trader	Sales Rep
8	Laura	Giussani	laura@northwindtraders.com	(123)555-0100	Northwind Trader	Sales Coordinator
9	Anne	Hellung-Larsen	anne@northwindtraders.com	(123)555-0100	Northwind Trader	Sales Rep
3	Jan	Kotas	jan@northwindtraders.com	(123)555-0100	Northwind Trader	Sales Rep
6	Michael	Neipper	michael@northwindtraders.com	(123)555-0100	Northwind Trader	Sales Rep
4	Mariya	Sergienko	mariya@northwindtraders.com	(123)555-0100	Northwind Trader	Sales Rep
5	Steven	Thorpe	steven@northwindtraders.com	(123)555-0100	Northwind Trader	Sales Manager
7	Robert	Zare	robert@northwindtraders.com	(123)555-0100	Northwind Trader	Sales Rep
*	(New)					
	Total	9				

Record: 1 4 1 of 9 > >> No Filter Search

**FIGURE 20.6.** When you select the last value from the OpenArgs Demo drop-down list (see Figure 20.5), Access displays the Employee List form with changes made to the form caption and the Reports drop-down list.

## SUMMARY

---

In this chapter, you learned that numerous events can occur on an Access form and that you can react to a specific form event by writing an event procedure. If you don't write your own code to handle a particular form event, Access will use its default handler for the event. You have also learned how to use the Form's OpenArgs property to pass values from one form to another.

After trying out numerous hands-on exercises presented in this chapter, you should have a good understanding of how to write event procedures for an Access form. You should also be able to recognize the importance of form events in an Access application.

For more hands-on experience with event programming, proceed to the next chapter, which discusses the events recognized by controls placed on an Access form.

# Chapter 21 *EVENTS RECOGNIZED BY FORM CONTROLS*

In addition to the events for forms introduced in Chapter 20, you can control a great many events that occur for labels, text boxes, combo boxes, list boxes, option buttons, checkboxes, and other controls installed by default with an Access application. These events make it possible to manage what happens on a field level.

The best way to learn about form, report, or control events is to develop an application that addresses specific needs. For example, the AssetsDataEntry.accdb database keeps track of computer assets in various companies. We will use this database to further experiment with event programming. The main data entry form is divided into four easy-to-maintain sections as illustrated in Figure 21.1.

FIGURE 21.1 Custom data entry form.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*

**Hands-On 21.1 Launching the Custom Access Application**

1. Copy the AssetsDataEntry.accdb file from the **companion files** to your C:\VBAAccess2021\_ByExample folder.
2. To access the database source code, double-click the C:\VBAAccess2021\_ByExample\AssetsDataEntry.accdb file while holding down the Shift key. Access loads the database and the Navigation pane displays all Access objects. Now we are ready to examine various form and control events.

**ENTER (CONTROL)**

The Enter event occurs before a control actually receives the focus from another control on the same form. The Enter event applies to text boxes, combo boxes, list boxes, option buttons, checkboxes, option groups, command buttons, toggle

buttons, bound and unbound object frames, and subform and subreport controls. You can use the Enter event to display a message directing the user to first fill in another control on the form.

For example, when a user attempts to make a selection from the combo box controls located in the Room Information and Project Information sections of the Asset Management form (Figure 21.1) without first specifying the Company ID, the Enter event procedures may be triggered for: cboRooms\_Enter, cboRoomType\_Enter, cboOS\_Enter, and cboProject\_Enter.



### Hands-On 21.2 Using the Enter Event Procedure for the Combo Box Control

1. Open the **frmDataEntryMain** form in Form View.
2. Click inside the combo box control located to the right of **Room No.** This action will fire the following Enter event procedure:

```
Private Sub cboRooms_Enter()
    If Me.cboCompanyID = "" Or _
        IsNull(Me.cboCompanyID) Then
        MsgBox "Please select Company ID.", _
            vbInformation + vbOKOnly, _
            "Missing Company ID"
        Me.cboCompanyID.SetFocus
        Exit Sub
    End If
End Sub
```

3. Click **OK** to the information message generated by the cboRooms\_Enter event procedure. Notice that the cursor has been positioned inside the combo box control containing Company ID. Don't make any selections from the Company ID combo box at this time.
4. Click on the combo box control next to **Room Type**. This action will fire the following Enter event procedure:

```
Private Sub cboRoomType_Enter()
    If Me.cboCompanyID = "" Or _
        IsNull(Me.cboCompanyID) Then
        MsgBox "Please select Company ID.", _
            vbInformation + vbOKOnly, _
            "Missing Company ID"
        Me.cboCompanyID.SetFocus
        Exit Sub
    End If
    If Me.cboRooms = "" Or IsNull(Me.cboRooms) Then
        MsgBox "Please specify or " & _
```

```
"select Room number.", _  
    vbInformation + vbOKOnly, _  
    "Missing Room Number"  
Me.cboRooms.SetFocus  
Exit Sub  
End If  
End Sub
```

When you click the cboRoomType combo box control, the Enter event checks whether the cboCompanyID combo box control or cboRooms combo box control is empty. If no selection has been made in these controls, a message box is displayed, and the focus is moved to the appropriate combo box control.

5. Click **OK** to the information message generated by the cboRoomType\_Enter event procedure and notice that the cursor has again been positioned inside the Company ID combo box control.

## **BEFOREUPDATE (CONTROL)**

---

The BeforeUpdate event occurs when you attempt to save the record or leave the control after making changes. This event applies to text boxes, combo boxes, list boxes, option buttons, checkboxes, and bound object frames. Use this event to validate the entry.

For example, the combo box control in the Company Information section of the Asset Management form causes Access to display a custom message if the value of the cboCompanyID combo box control is Null. To cancel the Update event, the Cancel argument has been set to `True`.



### **Hands-On 21.3 Using the BeforeUpdate Event Procedure for the Combo Box Control**

1. Press **Alt+F11** to switch to the Visual Basic Editor window.
2. In the Project Explorer window, double-click the **frmDataEntryMain** form.
3. From the Object drop-down box at the top-left side of the Code window, select **cboCompanyID**. In the Procedure drop-down box at the top-right side of the Code window, select **BeforeUpdate**.

The Code window should display this event procedure:

```
Private Sub cboCompanyID_BeforeUpdate _  
    (Cancel As Integer)  
Dim strMsg As String, strTitle As String  
Dim intStyle As Integer
```

```
If IsNull(Me!cboCompanyID) Or _  
    Me!cboCompanyID = "" Then  
    strMsg = "You must pick a value " & _  
        "from the Company ID list."  
    strTitle = "Company ID Required"  
    intStyle = vbOKOnly  
    MsgBox strMsg, intStyle, strTitle  
    Cancel = True  
End If  
End Sub
```

Let's walk through this procedure using the debugging skills acquired in Chapter 9.

4. Position the cursor on the line with the `If` statement, then press **F9** or choose **Debug | Toggle Breakpoint**.
5. Activate the **frmDataEntryMain** form in Form view and make a selection from the Company ID combo box.

When you make your selection, the `BeforeUpdate` event procedure is fired, and the Code window appears in break mode with the yellow highlight on the breakpoint line. Press **F8** to step through the code line by line. Because you have not set up more breakpoints, you cannot see that two other events (`cboCompanyID_AfterUpdate` and `cboRooms_Enter`) were triggered when you made a selection from the Company ID combo box.

6. When the procedure finishes executing, activate the **frmDataEntryMain** form. You should see the text boxes filled with a company name and address and the cursor positioned inside the Room No combo box and ready for the next selection or data entry.

## AFTERUPDATE (CONTROL)

---

The `AfterUpdate` event occurs after the data in the control has been modified. It applies to text boxes, combo boxes, list boxes, option buttons, checkboxes, and bound object frames. Unlike the `BeforeUpdate` event, the `AfterUpdate` event cannot be canceled. Use this event to fill in other controls on the form based on the newly entered or selected value.

For example, after updating the `cboCompanyID` combo box in the Company Information section of the Asset Management form, the following event procedure is executed:

```
Private Sub cboCompanyID_AfterUpdate()  
    With Me
```

```
.txtCompanyName = Me.cboCompanyID.Column(1)
.txtAddress = Me.cboCompanyID.Column(2)
.txtCity = Me.cboCompanyID.Column(3)
.txtRegion = Me.cboCompanyID.Column(4)
.txtPostalCode = Me.cboCompanyID.Column(5)
.txtCountry = Me.cboCompanyID.Column(6)
.cboRooms.Value = vbNullString
.cboRooms.Requery
.txtRoomDescription = vbNullString
.cboRoomType = vbNullString
.cboOS = vbNullString
.txtOperatingSystem = vbNullString
.cboProject = vbNullString
.txtPID = vbNullString
End With
If Me.cboRooms.ListCount = 0 Then
    'do not display column headings
    Me.cboRooms.ColumnHeads = False
Else
    Me.cboRooms.ColumnHeads = True
End If
Me.cboRooms.SetFocus
End Sub
```

In the preceding procedure, the company address information is filled in based on the contents of the cboCompanyID columns. For example, to fill in the street address, you can read the value of the Columns() property of the cboCompanyID control, even though this column is not visible when you view the combo box:

```
Me.txtAddress = Me.cboCompanyID.Column(2)
```

Note that because the combo box column numbering begins with zero (0), this statement actually reads the contents of the third column. Next, the combo box labeled Room No is re-queried and a number of other controls on the form are cleared.

Also, note how the intrinsic constant named `vbNullString` is used here instead of an empty string ("") to clear text boxes or combo boxes on a form. The final procedure code segment contains the `If...Then...Else` statement that sets the `ColumnHeads` property of the `cboRooms` control to `False` if there are no rooms associated with the selected Company ID.

The last line of the code:

```
Me.cboRooms.SetFocus
```

moves the focus to the combo box control with the room numbers. When this code is executed, the `cboRooms_Enter` event procedure will be triggered.



#### Hands-On 21.4 Using the AfterUpdate Event Procedure for the Combo Box Control

1. In the Code window of the **frmDataEntryMain** form, ensure that the Object drop-down box displays `cboCompanyID`.
2. Choose the **AfterUpdate** event from the Procedure drop-down box, then set a breakpoint on the first line of this procedure.
3. In the Code window of the **frmDataEntryMain** form, ensure that the Object drop-down box displays `cboRooms`.
4. Choose the **Enter** event from the Procedure drop-down box, then set a breakpoint on the first line of this procedure.
5. Switch to Form view for the **frmDataEntryMain** form, then make another selection from the **Company ID** combo box. When the Code window appears in break mode, step through the code line by line by pressing **F8**. Notice that the following three event procedures are run:

```
cboCompanyID_BeforeUpdate  
cboCompanyID_AfterUpdate  
cboRooms_Enter
```

6. Choose **Debug | Clear All Breakpoints** to remove the breakpoint you set in this and the previous hands-on exercise.

### NOTINLIST (CONTROL)

The **NotInList** event is triggered if the user enters a value that is not in the list when the **LimitToList** property of a combo box control is set to **True**. The **NotInList** event procedure can take the following two arguments:

- **NewData**—A string that Access uses for passing the user-entered text to the event procedure.
- **Response**—An integer specifying what Access should do after the procedure executes. This argument can be set to one of the following constants:
  - `acDataErrAdded`—Set the **Response** argument to `acDataErrAdded` if the event procedure enters a new value in the combo box. This constant tells Access to re-query the combo box, adding the new value to the list.

- acDataErrDisplay—Set the Response argument to acDataErrDisplay if you want Access to display the default error message when a user attempts to add a new value to the combo box. The default Access message requires the user to enter a valid value from the list.
- acDataErrContinue—Set the Response argument to acDataErrContinue if you display your own message in the event procedure. Access will not display its default error message.

The NotInList event applies only to combo boxes. Use this event to display a custom warning message or to trigger a custom function that allows the user to add a new item to the list.

For example, after attempting to enter a nonexistent value in the combo box labeled Room Type in the Room Information section of the Asset Management form, this event procedure is executed:

```
Private Sub cboRoomType_NotInList _  
    (NewData As String, _  
     Response As Integer)  
    MsgBox "Please select a value " & _  
        "from the list.", _  
        vbInformation + vbOKOnly, _  
        "Invalid entry"  
    ' Continue without displaying  
    ' default error message.  
    Response = acDataErrContinue  
End Sub
```

The cboRoomType\_NotInList code displays a custom message if a user attempts to type an invalid entry in the cboRoomType combo box control on the form.



### Hands-On 21.5 Using the NotInList Event Procedure for the Combo Box Control

1. In the Code window of the **frmDataEntryMain** form, ensure that the Object drop-down box displays **cboRoomType**.
2. Choose the **NotInList** event from the Procedure drop-down box. Access will create the procedure stub for you. Complete this procedure as shown in the previous section.
3. Open the **frmDataEntryMain** form in Form view.
4. Select a company ID from the **Company ID** combo box.
5. Select a room number from the **Room No** combo box, or type a value in

this box if the value is not available.

6. Type a new value in the **Room Type** combo box, then click on the **Operating System** combo box. This will trigger the `cboRoomType_NotInList` event procedure code to run. Your custom error message should appear. Click **OK** to the message box. Notice that Access does not display its own default message because we set the `Response` argument to `acDataErrContinue`.
7. Select any value from the **Room Type** combo box.

## **CLICK (CONTROL)**

---

The Click event occurs when the user clicks a control with the left mouse button or presses an Enter key when a command button placed on a form has its Default property set to Yes. The Click event applies only to forms, form sections, and controls on a form. The Asset Management data entry form contains several command buttons that allow the user to add new values to appropriate combo box selections. For example, when the user clicks the button labeled Add New Company, the following Click event procedure is triggered:

```
Private Sub cmdNewCompany_Click()
    On Error GoTo Err_cmdNewCompany_Click

    Dim stDocName As String
    Dim stLinkCriteria As String

    stDocName = "frmAddCompany"
    DoCmd.OpenForm stDocName, , , stLinkCriteria

    Exit_cmdNewCompany_Click:
        Exit Sub
    Err_cmdNewCompany_Click:
        MsgBox Err.Description
        Resume Exit_cmdNewCompany_Click
End Sub
```

This event procedure opens a window titled New Company Data Entry Screen (Figure 21.2), where the user can enter new company information.

---

New Company Data Entry Screen

---

**Enter New Company Information**

Company ID:	<input type="text"/>				
Company Name:	<input type="text"/>				
Street Address:	<input type="text"/>				
City:	<input type="text"/>	Region:	<input type="text"/>	Postal Code:	<input type="text"/>
Country:	<input type="text"/>				
<input type="button" value="Save"/> <input type="button" value="Cancel"/>					

---

FIGURE 21.2. This data entry form is used for adding new companies to the database.

When the user clicks the Save button on the New Company Data Entry Screen window, the Click event procedure attached to this button ensures that:

- All text boxes have been filled in
- The Company ID does not contain more than five characters
- The Postal Code text box contains a five-digit zip code for the United States
- The Company ID does not already exist in the table

Notice that the New Company Data Entry form is unbound, which means that it isn't connected to a record source such as a table, query, or SQL statement. After successful data validation, the procedure uses the `AddNew` method of the ADO Recordset object to create a new record. This record is added to the `tblCompanies` table that provides the record source for the Company ID combo box control on the Asset Management data entry form. Next, the `cboCompanyID` combo box control on the Asset Management form is re-queried so that the new Company ID can be accessed from the drop-down list when the user returns to the form.

```
Private Sub cmdSaveCompanyInfo_Click()
    Dim conn As ADODB.Connection
    Dim rst As New ADODB.Recordset
    Dim ctrl As Control
    Dim count As Integer
```

```
On Error GoTo Err_cmdSaveCompanyInfo_Click

'validate data prior to save

For Each ctrl In Me.Controls
    If ctrl.ControlType = acTextBox And IsNull(ctrl) _
        Or IsEmpty(ctrl) Then
        count = count + 1
        If count > 0 Then
            MsgBox "All text fields must be filled in.", _
                vbInformation + vbOKOnly, _
                "Missing Data"
            ctrl.SetFocus
            Exit Sub
        End If
    End If
Next

If Len(Me.txtAddCompanyID) <> 5 Then
    MsgBox "The Company ID requires 5 characters"
    Me.txtAddCompanyID.SetFocus
    Exit Sub
End If

'check the zipcode field
If Len(Me.txtAddPostalCode) <> 5 And _
    UCase(Me.txtAddCountry) = "USA" Then
    MsgBox "Please enter a five-digit zip code " & _
        "for the United States.", _
        vbInformation + vbOKOnly, "Invalid Zip Code"
    Me.txtAddPostalCode.SetFocus
    Exit Sub
End If

'are any alphabetic characters in zip code?
If Not IsNumeric(Me.txtAddPostalCode) And _
    UCase(Me.txtAddCountry) = "USA" Then
    MsgBox "You can't have letters in Zip Code.", _
        vbInformation + vbOKOnly, "Invalid Zip Code"
    Me.txtAddPostalCode.SetFocus
    Exit Sub
End If

'save the data
Set conn = CurrentProject.Connection
```

```
With rst
    .Open "SELECT * FROM tblCompanies", _
        conn, adOpenKeyset, adLockOptimistic
    'check if the CompanyID is not a duplicate
    .Find "CompanyID=''' & Me.txtAddCompanyID & '''"
    'if Company already exists then get out

    If Not rst.EOF Then
        MsgBox "This Company is already in the list : " _
            & rst("CompanyID"), _
            vbInformation + vbOKOnly, "Duplicate Company ID"
        Me.txtAddCompanyID.SetFocus
        Exit Sub
    End If

    .AddNew
    !CompanyID = Me.txtAddCompanyID
    !CompanyName = Me.txtAddCompanyName
    !Address = Me.txtAddAddress
    !City = Me.txtAddCity
    !Region = Me.txtAddRegion
    !PostalCode = Me.txtAddPostalCode
    !Country = Me.txtAddCountry
    .Update
    .Close
End With
Set rst = Nothing
conn.Close
Set conn = Nothing

'requery the combo box on the main form
Forms!frmDataEntryMain.cboCompanyID.Requery
'close the form
DoCmd.Close

Exit_cmdSaveCompanyInfo_Click:
    Exit Sub
Err_cmdSaveCompanyInfo_Click:
    MsgBox Err.Description
    Resume Exit_cmdSaveCompanyInfo_Click
End Sub
```



### Hands-On 21.6 Using the Click Event Procedure for the Command Button Control

1. Open the frmDataEntryMain form in Form view.
2. Click the **Add New Company** command button.
3. When the New Company Data Entry Screen window appears, enter the information shown in Figure 21.3.

New Company Data Entry Screen



FIGURE 21.3. After saving the new company information in this window, the Company ID will appear in the Company ID combo box on the main form.

4. Click the **Save** button to save the company information. Access will run the cmdSaveCompanyInfo\_Click event procedure, as shown earlier. If you have not entered data according to the criteria listed in this event procedure, Access will not allow you to save data until you correct the problem.
5. Back on the main form, select the newly added company (**GOSPO**) from the Company ID combo box.

Notice how the data entry form displays several icons with a question mark. Each icon is actually a command button with a Click event attached to it. When you click on the question mark button, a simple form will appear with help information pertaining to the form's section of the data entry screen.

For example, the following Click event procedure is executed upon clicking the question mark button in the Room Information section on the Asset Management data entry form:

```
Private Sub cmdRoomInfoSec_Click()
    Dim stDocName As String
```

```
Dim stLinkCriteria As String

On Error GoTo Err_cmdRoomInfoSec_Click

stDocName = "frmHelpMe"
stLinkCriteria = "HelpId = 2"
DoCmd.OpenForm stDocName, , , stLinkCriteria

Exit_cmdRoomInfoSec_Click:
    Exit Sub
Err_cmdRoomInfoSec_Click:
    MsgBox Err.Description
    Resume Exit_cmdRoomInfoSec_Click
End Sub
```

This procedure loads the appropriate help topic into the text box control, as illustrated in Figure 21.4.

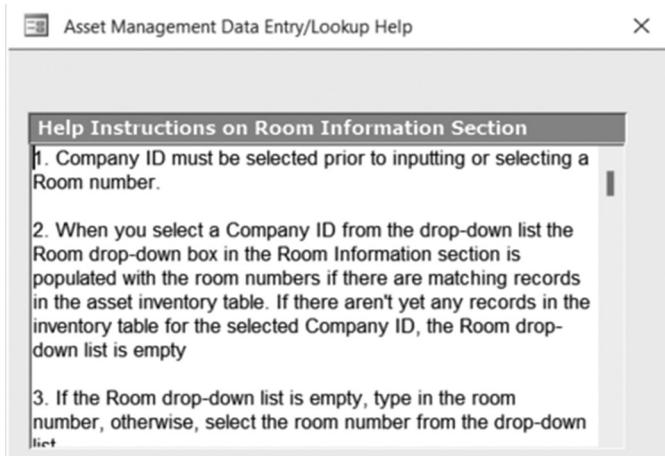


FIGURE 21.4. By clicking the question mark button in each section of the data entry form, users can get detailed guidelines on how to work with the form section.

## **DBLCLICK (CONTROL)**

The DblClick event occurs when the user double-clicks the form or control. This event applies only to forms, form sections, and controls on a form, not controls on a report. Hands-On 21.7 demonstrates how the user of the Asset Management application can delete an asset by double-clicking on its name.



### Hands-On 21.7 Using the DblClick Event Procedure for the Listbox Control

1. Open the **frmDataEntryMain** form in Form view.
2. Make appropriate selections on the Asset Management data entry form.
3. Click the **Add New Asset Type** button in the Hardware Information section. If this button cannot be clicked, you have not made all the necessary selections in the upper part of the form.  
The Add New Asset Type Data Entry Screen window will appear, as shown in Figure 21.5.

Add New Asset Type Data Entry Screen

ID	Description
5	Imation
4	ITV
11	Laptop
7	MiniTower
9	MiniTower Monitor
2	Monitor
3	Printer
8	Scanner
6	Server
10	Server Monitor
1	WS

**FIGURE 21.5.** This form allows the user to add a new entry to the Asset Type column in the Hardware Information section of the Asset Management form or delete the asset entry by double-clicking on the entry in the Available Assets listbox.

4. In the Add New Asset Type Data Entry Screen window, enter **iPad** in the Asset Type text box and click the **Save** button.
5. In the main form, open the combo box in the **Asset Type** column and scroll down to view the newly added asset type—iPad. Do not make any selection in this combo box.
6. Click the **Add New Asset Type** button in the Hardware Information section to return to the Add New Asset Type Data Entry Screen window.  
The left side of the data entry screen (see Figure 21.6) displays a listbox with the currently available assets. When the user double-clicks any item in the list,

the following DblClick event procedure will determine whether the item can be deleted:

```
Private Sub lboxCategories_DblClick _
    (Cancel As Integer)
    Dim conn As ADODB.Connection
    Dim myAsset As String
    Dim myAssetDesc As String
    Dim Response As String
    Dim strSQL As String

    myAsset = Me.lboxCategories.Value
    myAssetDesc = Me.lboxCategories.Column(1)

    If myAsset >= 1 And myAsset <= 11 Then
        MsgBox "Cannot Delete - " & _
            "This item is being used.", _
            vbOKOnly + vbCritical, _
            "Asset Type: " & myAsset
        Exit Sub
    End If

    If (Not IsNull(DLookup("[AssetType]", _
        "tblProjectDetails", _
        "[AssetType] = " & myAsset))) Or _
        Not IsNull(DLookup("[EquipCategoryID]", _
        "tblEquipInventory", _
        "[EquipCategoryID] = " & myAsset)) Then

        MsgBox "This item cannot be deleted.", _
            vbOKOnly + vbCritical, _
            "Asset Type: " & myAsset
    Else
        Response = MsgBox("Do you want to " & _
            "delete this Asset?", _
            vbYesNo, "Delete - " & myAssetDesc & " ?")
        If Response = 6 Then
            Set conn = CurrentProject.Connection
            strSQL = "DELETE * FROM " & _
                "tblEquipCategories Where EquipCategoryID = " &_
                myAsset
            conn.Execute (strSQL & myAsset)
            conn.Close
            Set conn = Nothing
            Me.lboxCategories.Requery
        End If
    End If
```

```
End If

DoCmd.Close

'requery the combo box on the subform
Forms!frmDataEntryMain.frmSubProjectDetails.Form.EquipCatId.
Requery
End Sub
```

Add New Asset Type Data Entry Screen

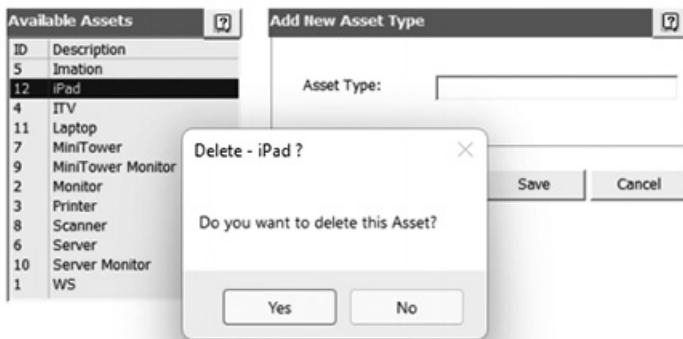


FIGURE 21.6 You can delete an item from the Available Assets list only if the item has not yet been used during the data entry.

7. Double-click on **iPad** in the Available Assets listbox. The DblClick event procedure attached to the listbox will ask you whether you want to delete this asset. Click **Yes** to the message.  
Notice that the iPad entry disappears from the Available Assets listbox.
8. Click the **Cancel** button to exit the Add New Asset Type Data Entry Screen window.

## CHAPTER SUMMARY

In this chapter, you worked with a custom Access application and examined event procedures for various controls placed on an Access form. There are other event procedures not discussed here that control how the Asset Management form and its controls respond to the user's actions. As you explore this application on your own, you will start noticing the areas where writing additional event handlers would prove beneficial to the application's users. So get to it! Tear

this sample application apart. Rebuild it. Add new features. Change the user interface if you want. Learn how to handle whatever event may come your way. Be prepared, because events happen frequently in an Access application, and sooner or later you'll need to respond to them.

The next chapter focuses on working with Access reports and controlling report behavior with event programming.

# Chapter 22 *ENHANCING ACCESS REPORTS AND USING REPORT EVENTS*

Reports have always been a very popular and widely used feature in Access. Access reporting is very interactive thanks to a Report view. With the Report view you can easily perform data searches, sorting, and filtering. You can also copy the data. Many of the Access form features are also available for reports. For example, to make long tabular reports easier to read, you can apply alternating row shading just by changing the Alternate Back Color property in the report's Detail section. Like forms, reports can utilize bound Image controls, rich text formatting, and filtering and sorting features. The Layout view makes it easier to design reports; because you are working with the live data in Layout view, you do not need to switch between Design and Report views to see how the final report will look. The Layout view allows formatting report sections and controls, adding new fields, applying AutoFormats, grouping and sorting data, and changing many of the report's properties. Layouts can be used in reports for resizing and moving groups of controls together or adding grid lines that grow or shrink with the data. It is very easy to design objects in Layout view. You can drop any control anywhere within the layout. Your controls can span multiple rows and columns. Like forms, reports can be enhanced by applying a consistent style using themes. The reports distribution is easy with the portable document format (.pdf) and XML Paper Specification (.xps) format.

The feature most appreciated by all Access users is the ability to view a report as a subform on a form. Additionally, you can specify the name of a report by using the `SourceObject` property of a subform control on a form.

## CREATING ACCESS REPORTS

---

The Access Report Wizard will walk you through the report creation process by presenting various options to choose from, such as selecting a data source for your report, determining criteria such as grouping and sorting, and offering formatting options (layout, orientation, and style). The Report wizard makes it easy to create a report based on multiple tables.

If you need more control over creating a report, you may want to try Report Designer. You can bind your report to a table, query, or SQL statement, and add VBA code behind a report as demonstrated later in this chapter. When using Report Designer, you will not be able to see the actual data from tables and views at design time. You need to switch to Print Preview to view the entire report. To overcome this limitation, try working with the Layout view. In Layout view one can add different types of controls, as well as functionality for sorting, grouping, and calculating totals. You can also apply different formatting to the Layout view while viewing the actual data from your tables and queries.

In addition to creating Access reports via these simple built-in tools, you can create an Access report programmatically by using the `CreateReport` method of the Application object. This process is discussed further in this chapter.

## USING REPORT EVENTS

---

When an Access report is run, a number of events can occur. The following examples demonstrate how to control what happens not only when the report is opened, activated, deactivated, or closed, but also when there are no records for the report to display or the report record source simply does not exist.

### Open

---

The Open event for a report occurs when the report is opened. Use this event to display support forms or custom buttons, or to change the record source for the report. The event procedure in Hands-On 22.1 demonstrates how to change a report's record source on the fly.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### **Hands-On 22.1 Writing the Report\_Open Event Procedure**

1. Start Access and create a new database named **Chap22.accdb** in your **C:\VBAAccess2021\_ByExample** folder.
2. Import all the tables, queries, forms, reports, macros, and modules from the **Northwind.mdb** sample database to your **Chap22.accdb** database.
  - To do this, in the Access window, choose **External Data | New Data Source | From Database | Access**.
  - In the File name box, type **C:\VBAAccess2021\_ByExample\Northwind.mdb** and click **OK**.
  - In the Import Objects window, select the **Tables** tab and click the **Select All** button. This will highlight all the tables. Select the **Queries** tab and click the **Select All** button. Select the **Forms** tab and click the **Select All** button. Select the **Reports** tab and click the **Select All** button. Do the same for macros and modules. After selecting all the objects on the specified tabs, click **OK** to begin importing.
  - Click the **Close** button when done.
3. In the Access window's Navigation pane, select the **Customers** table and choose **Create | Report Wizard**. Select all the fields for the report and click the **Next** button. Continue clicking **Next** until you get to the Report Wizard screen where you can specify the title for your report. Type **rptCustomers** for the title and click **Finish**. Access opens the report in Print Preview.
4. Right-click the report tab and choose **Design view** from the context menu.
5. In the Report Header area, click the report title (label control) to select it. Resize the control to allow for longer text that will be entered dynamically by the event procedure in step 7. In the property sheet for the selected label control, click the **All** tab and enter **lblCustomers** as the Name property and enter **Customers** as the Caption property, overwriting the previous value.
6. In the property sheet, select **Report** from the drop-down box and click the **Event** tab. Click next to the **On Open** event property and choose **[Event Procedure]** from the drop-down box. Click the **Build** button (...).

7. Access opens the Visual Basic Editor window and writes the stub of the Report\_Open event procedure. Complete the code of the following Report\_Open event procedure:

```
Private Sub Report_Open(Cancel As Integer)
    Dim strCustName As String
    Dim strSQL As String
    Dim strWHERE As String
    Dim ctrl As TextBox

    On Error GoTo ErrHandler
    strSQL = "SELECT * FROM Customers"

    strCustName = InputBox("Type the first letter " & _
        " of the Company Name or type an asterisk (*) " & _
        " to view all companies.", "Show All /Or Filter")

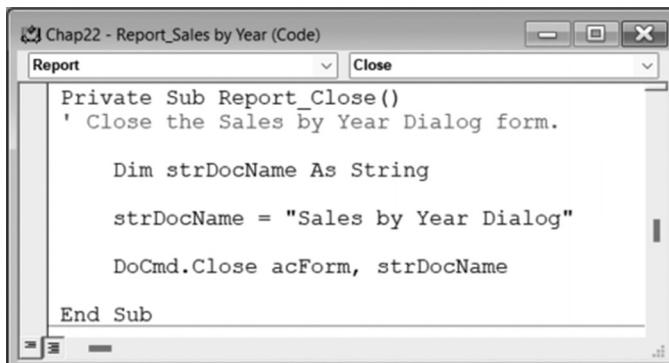
    If strCustName = "" Then
        Cancel = True
    ElseIf strCustName = "*" Then
        Me.RecordSource = strSQL
        Me.lblCustomers.Caption = "All Customers"
    Else
        strCustName = "!" & Trim(strCustName) & "!"
        strWHERE = " WHERE CompanyName Like " & _
            strCustName & ""
        Debug.Print strSQL
        Debug.Print strWHERE
        Me.RecordSource = strSQL & strWHERE
        Me.lblCustomers.Caption = "Selected Customers" & _
            "(" & UCase(strCustName) & ")"
    End If

    For Each ctrl In Me.Detail.Controls
        If ctrl.BackStyle = 1 Then ctrl.BackStyle = 0
    Next
    Exit Sub
ErrHandler:
    MsgBox Err.Description
End Sub
```

8. Switch to the rptCustomers report's Design view and choose **Home | View | Print Preview**. A message box will appear where you can enter an asterisk (\*) to view all customers or the first letter of a company name if you'd like to limit your records. To cancel the report, click **Cancel** or press the **Esc** key.

## Close

The Close event occurs when you close the report. Use this event to close supporting forms or to perform other cleanup operations. You cannot cancel the Close event. Figure 22.1 shows the Report\_Close event procedure for the Report\_Sales by Year report. To run this report, double-click the Sales by Year report name in the Navigation pane. In the Sales by Year Dialog form that opens, specify the report beginning and ending dates and click OK. Access will run the report and hide the form. The Sales by Year Dialog form remains open while the report is open and is closed during the Report\_Close event.



```
Private Sub Report_Close()
    ' Close the Sales by Year Dialog form.

    Dim strDocName As String
    strDocName = "Sales by Year Dialog"
    DoCmd.Close acForm, strDocName
End Sub
```

FIGURE 22.1 The Report\_Close event procedure is often used to close supporting forms.

## Activate

The Activate event occurs when the report is opened right after the Open event but before the event for the first section of the report. The procedure in Hands-On 22.2 displays a message when the report is open in Print Preview and returns the name of the default printer to the Immediate window.



### Hands-On 22.2 Writing the Report\_Activate Event Procedure

This hands-on exercise uses the rptCustomers report created in Hands-On 22.1.

1. In the Visual Basic Editor's Project Explorer window, double-click the **rptCustomers** report. In the Code window, below the previous procedure code, enter the following Report\_Activate event procedure:

```
Private Sub Report_Activate()
    If Me.CurrentView = acCurViewPreview Then
        MsgBox "Activating Print Preview of " _
```

```

& Me.Name & " report."
Debug.Print "Default Printer: " & _
Application.Printer.DeviceName
End If
End Sub

```

Notice how the CurrentView property is used to determine the current view of an object. Table 22.1 lists the CurrentView property constants.

TABLE 22.1 CurrentView property names and values

CurrentView Property Name	Value	Description
acCurViewDesign	0	The object is in Design view.
acCurViewFormBrowse	1	The object is in Form view.
acCurViewDatasheet	2	The object is in Datasheet view.
acCurViewPivotTable	3	The object is in PivotTable view.
acCurViewPivotChart	4	The object is in PivotChart view.
acCurViewPreview	5	The object is in Print Preview.
acCurViewReportBrowse	6	The object is in Report view.
acCurViewLayout	7	The object is in Layout view.

2. In the Navigation pane of the Access window, right-click the **rptCustomers** report and choose **Print Preview**. Enter your report criteria when prompted. Upon activation of the report, the Report\_Activate event will fire with a message. Click **OK** to the message, and then switch to the Immediate window to check out the name of your default printer.
3. Close the **rptCustomers** report and save changes to the report when prompted.

### Deactivate

---

The Deactivate event occurs when a report loses the focus to a table, query, form, report, macro, module, or database window. This event occurs before the Close event for the report.

### NoData

---

The NoData event occurs when the record source for the report contains no records. This event allows you to cancel the report when no records are available. The event procedure in Hands-On 22.3 displays a message when the user enters criteria that are not met.



### Hands-On 22.3 Writing the Report\_NoData Event Procedure

This hands-on exercise uses the **rptCustomers** report created in Hands-On 22.1.

1. In the Visual Basic Editor's Project Explorer window, double-click the **Report\_rptCustomers** report. In the Code window, enter the following **Report\_NoData** event procedure:

```
Private Sub Report_NoData(Cancel As Integer)
    MsgBox "There is no data for the criteria " & _
        "you entered."
    Cancel = True
End Sub
```

2. Switch to the Access window and open the **rptCustomers** report. Request to see customers with a company name starting with the letter “X.” Because there aren't any company names beginning with “X,” a message box will be displayed, saying that there is no data for the criteria entered, and the report will be canceled.

## Page

---

The Page event occurs after a page is formatted but before it is printed. Use the Page event to customize the appearance of your printed reports by adding lines, circles, and graphics. The event procedure in Hands-On 22.4 will draw a red border around the report pages.



### Hands-On 22.4 Drawing a Page Border Using the Report\_Page Event Procedure

This hands-on exercise uses the **rptCustomers** report created in Hands-On 22.1.

1. In the Visual Basic Editor's Project Explorer window, double-click the **rptCustomers** report. In the Code window, enter the following **Report\_Page** event procedure:

```
Private Sub Report_Page()
    Me.DrawLine (0, 0)-(Me.ScaleWidth, Me.ScaleHeight), vbRed, B
    Me.DrawWidth = 15 ' pixels
End Sub
```

Notice that the `DrawWidth` method specifies the thickness of the line and the `Line` method draws a line with the upper-left corner at (0, 0) and the lower-right corner at (Me.ScaleWidth, Me.ScaleHeight). The `ScaleWidth` and `ScaleHeight` properties specify the width and height of the report.

2. Switch to the Access window and open the **rptCustomers** report in Print Preview. Notice that when the report appears on the screen, a red border surrounds the pages (see Figure 22.2).
3. Close the rptCustomers report and save changes when prompted.



FIGURE 22.2. You can frame your Access report pages with a red line by implementing the Report\_Page event procedure shown in Hands-On 22.4.

## Error

The Error event is triggered by errors in accessing the data for the report. Use this event to replace the default error message with your custom message. The Error event takes the following two arguments:

- **DataErr**—Contains the number of the Microsoft Access error that occurred.
- **Response**—Determines whether error messages should be displayed. It may be one of the following constants:
  - **acDataErrContinue**—Ignore the error and continue without displaying the default Microsoft Access error message.
  - **acDataErrDisplay**—Display the default Microsoft Access error message. This is the default.

The Report\_Error event procedure in Hands-On 22.5 illustrates how to use the value of the `DataErr` argument together with the `AccessError` method to determine the error number and its descriptive string.

The statement:

```
Response = acDataErrContinue
```

will prevent the standard Microsoft Access error message from appearing. The Error event for reports works the same as the Error event for forms—but only Microsoft Access ACE or Jet Engine errors can be trapped here.

To trap errors in your VBA code, use the `On Error GoTo` statement to direct the procedure flow to the location of the error-handling statements in your procedure.



### Hands-On 22.5 Writing the Report\_Error Event Procedure

This hands-on exercise uses the `rptCustomers` report created in Hands-On 22.1.

1. In the Navigation pane, rename the Customers table **Customers2**.
2. In the Visual Basic Editor's Project Explorer window, double-click the `rptCustomers` report. In the Code window, enter the following `Report_Error` event procedure:

```
Private Sub Report_Error(DataErr As Integer, _
    Response As Integer)
    ' obtain information about the error
    MsgBox Application.AccessError(DataErr), _
        vbOKOnly, "Error Number: " & DataErr
    If DataErr = 3078 Then
```

```
Response = acDataErrContinue  
MsgBox "Your custom error message goes here."  
End If  
End Sub
```

3. Switch to the Access window and open the **rptCustomers** report. When the input box appears prompting you for the criteria, type any letter and press **OK**. At this point the Report\_Error event will fire because the underlying data for the **rptCustomers** report does not exist. Because you renamed the **Customers** table that this report uses for its data source, Microsoft Access cannot locate the data and generates the error.
4. In the Navigation pane, change the **Customers2** table's name back to **Customers** and open the **rptCustomers** report to ensure that it does not produce unexpected errors.
5. Close the report when finished and save the changes when prompted.

## EVENTS RECOGNIZED BY REPORT SECTIONS

---

An Access report can contain various sections such as Report Header/Footer, Page Header/Footer, the Detail section, and Group Headers/Footers. All report sections can respond to the Format and Print events. These events occur when you print or preview a report. In addition, the Report Header/Footer and the Detail sections recognize the Retreat event that occurs when Access returns to a previous section during report formatting.

### **Format (Report Section Event)**

---

A Format event occurs for each section in a report before Access formats the section for previewing or printing. This event takes the following two arguments:

- **Cancel**—Determines if the formatting of the section occurs. To cancel the section formatting, set this argument to **True**.
- **FormatCount**—Is an integer that specifies whether the Format event has occurred more than once for a section. If a section does not fit on one page and the rest of the section needs to be moved to the next page of the report, the **FormatCount** argument is set to 2.

Use the Format event in the appropriate report section for changes that affect page layout, as described in Table 22.2. For changes that don't affect page layout, use the Print event for the report section.

**TABLE 22.2** Effect of the Format event on report sections

Report Sections	Description of Event
Detail	The Format event occurs for each record in the section just before Access formats the data in the record. You can access the data in the current record using the event procedure.
Group Headers	The Format event occurs for each new group. You can access the data in the Group Header and the data in the first record in the Detail section using the event procedure.
Group Footers	The Format event occurs for each new group. You can access the data in the Group Footer and the data in the last record in the Detail section via an event procedure.

The event procedure in Hands-On 22.6 demonstrates how to make reports easier to read by shading alternate rows.



### **Hands-On 22.6 Shading Alternate Rows Using the Detail\_Format Event Procedure**

This hands-on exercise uses the rptCustomers report created in Hands-On 22.1.

1. In the Visual Basic Editor's Project Explorer window, double-click the **rptCustomers** report. In the Code window, enter the following **Detail\_Format** event procedure. Do not type the `Option Compare Database` and `Option Explicit` statements if they are already present at the top of the Code window.

```
Option Compare Database
Option Explicit
```

```
Dim shaded As Boolean
```

```
Private Sub Detail_Format(Cancel As Integer, _
    FormatCount As Integer)
    If shaded Then
        Me.Detail.BackColor = vbYellow
    Else
        Me.Detail.BackColor = vbWhite
    End If
    shaded = Not shaded
End Sub
```

Notice that at the top of the module sheet (in the module's Declarations area), we have placed the following declaration statement:

```
Dim shaded As Boolean
```

This statement declares the global variable of the Boolean type to keep track of the alternate rows.

When you run the report, upon printing the Detail section, Access will check the value of the `shaded` variable. If the value is True, it will change the background of the formatted row to yellow (which produces a light gray background when printed on a noncolor printer). The shaded value will then be set to False for the next row by using the following statement:

```
shaded = Not shaded
```

This statement works as a toggle. If `shaded` was True, it will be False now, and so on.

**2.** Modify the **Report\_Open** event procedure as follows:

- a. Add the following statement just below the other variable declarations that are already present inside this procedure:

```
Dim ctrl As TextBox
```

- b. Enter the following code before the `Exit Sub` statement:

```
For Each ctrl In Me.Detail.Controls  
    If ctrl.BackStyle = 1 Then ctrl.BackStyle = 0  
Next
```

The `For Each` loop will iterate through the controls in the detail section of the `rptCustomers` report and set the Back Style property of each text box control to 0 (Transparent).

**3.** Switch to the Access window and open the **rptCustomers** report in Print Preview. When the input box prompts you for the criteria, type an asterisk (\*) and press **OK**. You should see the same data as shown in Figure 22.2 with alternate shading.

**NOTE**

*In Access 2007–2021 you can shade alternate rows by setting the `AlternateBackColor` property for the Detail section instead of writing VBA code for the Format event of the Detail section.*

**4.** Close the `rptCustomers` report, saving changes when prompted.

The next hands-on exercise demonstrates how to suppress the Page Footer on the first page of your report by placing code in the `PageFooterSection_Format` event procedure.



### Hands-On 22.7 Suppressing the Page Footer Using the PageFooterSection\_Format Event Procedure

1. Using the Report Wizard, create a report called **rptProducts** based on the Products table. Choose the following fields for this report: **ProductID**, **ProductName**, **UnitPrice**, and **UnitsInStock**. On the last page of the Report Wizard, select the **Modify the report's design** option button.
2. In the Design view of the **rptProducts** report, select **PageFooterSection** in the property sheet. Click the **Event** tab. Click next to the **On Format** property and select **[Event Procedure]** from the drop-down list. Click the **Build** button (...) to activate the Code window.
3. In the Code window for **rptProducts**, enter the following **PageFooterSection\_Format** event procedure:

```
Private Sub PageFooterSection_Format(Cancel As Integer, _  
FormatCount As Integer)  
Dim ctrl As Control  
  
For Each ctrl In Me.PageFooterSection.Controls  
    If Me.Page = 1 Then  
        ctrl.Visible = False  
    Else  
        ctrl.Visible = True  
    End If  
Next ctrl  
End Sub
```

4. Switch to the Access window and open the **rptProducts** report in Print Preview. Notice that the Footer section does not appear on the first page of the report.
5. Close the **rptProducts** report and save changes to the report when prompted.

### Print (Report Section Event)

The Print event occurs after the data in a report section has been formatted but before the data is printed. The Print event occurs only for sections that are actually printed, as described in Table 22.3. To access data from sections that are not printed, use the Format event.

You can use the **PrintCount** argument to check whether the Print event has occurred more than once for a record. If part of a record is printed on one page and the rest is printed on the next page, the Print event will occur twice, and the **PrintCount** argument will be set to 2. You can use the **Cancel** argument to cancel the printing of a section.

TABLE 22.3 Effect of the Print event on report sections

Report Section	Description of Event
Detail	The Print event occurs for each record in the Detail section just before Access prints the data in the record.
Group Headers	The Print event occurs for each new group.
Group Footers	The Print event occurs for each new group.

The event procedure in Hands-On 22.8 demonstrates how to print a record range indicator in the report's Footer. This indicator will display the range of records printed on each page. You can easily modify this example procedure to print the first and last customer ID on the page (see the discussion that follows this hands-on exercise).



### Hands-On 22.8 Displaying a Record Range in the Report's Footer Using the Detail\_Print Event Procedure

This hands-on exercise uses the rptCustomers report you created in Hands-On 22.1.

1. Open the **rptCustomers** report in Design view and place two unbound text boxes in the report's Page Footer section. You may need to make other controls in the footer area smaller to make more room.
2. Change the **Name** property of the first box to **txtPage** and set its **Visible** property to **No**. Delete the label control in front of this text box.
3. Name the second text box **txtRange** and set the **Caption** property of its label control to **Records**.
4. In the property sheet for the **txtRange** and **Records** controls, set the **Display When** property to **Print Only** (see the note at the end of this exercise).
5. In the property sheet, select **Detail** from the drop-down box and click the **Event** tab. Set the **On Print** property of the Detail section to **[Event Procedure]** and write the code for the **Detail\_Print** event as shown here:

```

Private Sub Detail_Print(Cancel As Integer, _
PrintCount As Integer)
    Static rCount As Integer
    Static start As Integer
    Static firstID As String
    Static lastID As String

    If Me.Page <> Me.txtPage Then
        start = Me.CurrentRecord
        firstID = CustomerID
        Me.txtPage = Me.Page
    End If
End Sub

```

```

rCount = 0
End If
rCount = rCount + 1
lastID = CustomerID
If start <= rCount Then
    Me.txtRange = start & "-" & rCount
    ' Me.txtRange = UCASE(firstID) &
    "—" & UCASE(lastID)
Else
    rCount = Me.CurrentRecord
    lastID = CustomerID
End If
End Sub

```

The Detail\_Print event procedure is triggered for each record. It uses the `start` and `rCount` variables to keep track of the first and last items on the page.

- In the Code window, enter the **PageHeaderSection\_Print** event procedure as shown here:

```

Private Sub PageHeaderSection_Print(Cancel As Integer, _
PrintCount As Integer)
    Me.txtPage = 0
End Sub

```

- To test the event procedures, switch to the Access window and open the **rptCustomers** report in Print Preview displaying all customers. Notice the record range indicator at the bottom of the report page (Figure 22.3).
  - Close the **rptCustomers** report and save changes when prompted.
- The PageHeaderSection\_Print event procedure will set the value of the unbound `txtPage` text box to zero (0) whenever the Print event occurs for a new page.

FISS FISSA Fabrica Int	Diego Roel	Accounting	C/ Moralzarzal, 86	Madri	2803	Spain	(91) 555 9	(91)
FOLI Folies gourmand	Martine Ran	Assistant Sal	184, chaussée de Tourna	Lille	5900	Franc	20.16.10.	20.16
FOL Folk och fâ HB	Maria Larss	Owner	Åkergratan 24	Bräck	S-844	Swed	0695-34	6
FRA Frankenversand	Peter Franke	Marketing M	Berliner Platz 43	Münc	8080	Germ	089-0877	089-
FRA France restaurati	Carine Schmi	Marketing M	54, rue Royale	Nante	4400	Franc	40.32.21.	40.32
FRA Franchi S.p.A.	Paolo Accort	Sales Repres	Via Monte Bianco 34	Torin	1010	Italy	011-4988	011-
FUR Furia Bacalhau e	Lino Rodrigu	Sales Manag	Jardim das rosas n. 32	Lisboa	1675	Portu	(1) 354-25	(1) 3
GAL Galeria del gastr	Eduardo Sau	Marketing M	Rambla de Cataluña, 23	Barcel	0802	Spain	(93) 203 4	(93)
GO Godos Cocina Tí	José Pedro F	Sales Manag	C/ Romero, 33	Sevilla	4110	Spain	(95) 555 8	

Monday, January 17, 2022      Records 1-30      Page 1 of 3

FIGURE 22.3. This report displays the record range indicator at the bottom of the page (see Hands-On 22.8).

You can modify the event procedure in Hands-On 22.8 to print the first and last customer IDs on the page as shown in Figure 22.4. Simply replace the following statement in the Detail\_Print event procedure:

```
Me.txtRange = start & "-" & rCount
```

with the following line of code:

```
Me.txtRange = UCASE(firstID) & "-" & UCASE(lastID)
```

FOLI Follies gourmand Martine Ran Assistant Sal 184, chaussée de Tournai Lille	5900 Franc 20.16.10. 20.16
FOL Folk och fä HB Maria Larsson Owner Åkergratan 24	Bräck S-844 Swed 0695-34 6
FRA Frankenversand Peter Franke Marketing M Berliner Platz 43	Münch 8080 Germ 089-0877 089-
FRA France restaurati Carine Schmid Marketing M 54, rue Royale	Nantes 4400 Franc 40.32.21. 40.32
FRA Franchi S.p.A. Paolo Accort Sales Repres Via Monte Bianco 34	Torino 1010 Italy 011-4988 011-
FUR Funia Bacalhau e Lino Rodrigues Sales Manager Jardim das rosas n. 32	Lisboa 1675 Portu (1) 354-25 (1) 3
GAL Galería del gastr Eduardo Saa Marketing M Rambla de Cataluña, 23	Barcelona 0802 Spain (93) 203 4 (93)
GO Godos Cocina Tí José Pedro Fernández Sales Manager C/ Romero, 33	Sevilla 4110 Spain (95) 555 8

Monday, January 17, 2022

Records

ALFKI-GODOS

Page 1 of 3

FIGURE 22.4. This report displays the first and last Customer ID for a specific page at the bottom of each printed page.

<b>NOTE</b>	<p><i>When you open the rptCustomers report in Report view instead of in Print Preview, you will notice that there is no calculated value in the Records text box at the bottom of the page. The reason for this is that in Report view there aren't any pages. The entire report is one big continuous page. Since there aren't any pages Access cannot calculate any values that depend on the Page or Pages properties. Also, it's important to remember that the Print event for report sections does not fire in Report view. You can tell Access to display certain controls only in Print Preview by changing the Display When property of the control to Print Only in the property sheet.</i></p>
-------------	--

### Retreat (Report Section Event)

The Retreat event occurs when Access returns to previous sections of the report during report formatting. For example, after formatting a report section, if Access discovers that the data will not fit on the page, it will go back to the necessary location in the report to ensure that the section can properly begin on the next page.

The Retreat event occurs after the Format event but before the Print event. This event applies to all report sections except Page Headers and Footers. The Retreat event occurs for Group Headers and Footers whose KeepTogether property

has been set to Whole Group or With First Detail. This event is also triggered in subreports whose CanGrow or CanShrink properties have been set to True.

The Retreat event makes it possible to undo any changes made during the Format event for the section. The Retreat event is demonstrated in the sample Northwind.mdb database's Sales by Year report that you imported earlier to the Chap22.accdb database. Figure 22.5 shows the GroupFooter1\_Retreat event procedure.

```
Private Sub GroupFooter1_Retreat()
' If ShowDetails check box on Sales by Year Dialog form is checked,
' set value of Show text box to True so that page header will print on
' next page.

    If Forms![Sales by Year Dialog]!ShowDetails Then Me!Show.Value = True

End Sub
```

FIGURE 22.5 The Sales by Year report in the Northwind.mdb database uses the GroupFooter1\_Retreat event procedure to control printing of a page header.

## USING THE REPORT VIEW

---

Reports have an interactive view called Report view, as shown in Figure 22.6. This is the default view for all new reports created in Access 2007–2021. In this view you can easily copy data by selecting it and then clicking the Copy button in the Clipboard group of the Home tab or pressing Ctrl+C. If you need to find particular data in the report, use the Find button in the Find group of the Home tab or press Ctrl+F. Access will pop up the standard Find dialog box in which you can enter your search criteria. Filtering and sorting is also enabled for the Report view via the buttons located in the Sort & Filter section of the Home tab. A report open in Report view isn't divided into pages; it is a single big page. If you have any calculations that depend on the Page or Pages properties of the report, they may not return the correct results. Certain report events, such as Print and Format, will not be triggered when the report is displayed in the Report view. The Report view has its own new event called Paint that is used with sections in Report view. This event fires whenever a section needs to be drawn on the screen. Use this event to conditionally format controls in that view, as shown in Hands-On 22.9.

<b>NOTE</b>	<i>The Paint event fires multiple times for each section of the report because Access paints various elements of the given section separately at different times. The calculated controls and the items that require a change in background or foreground colors are each painted separately.</i>
-------------	---

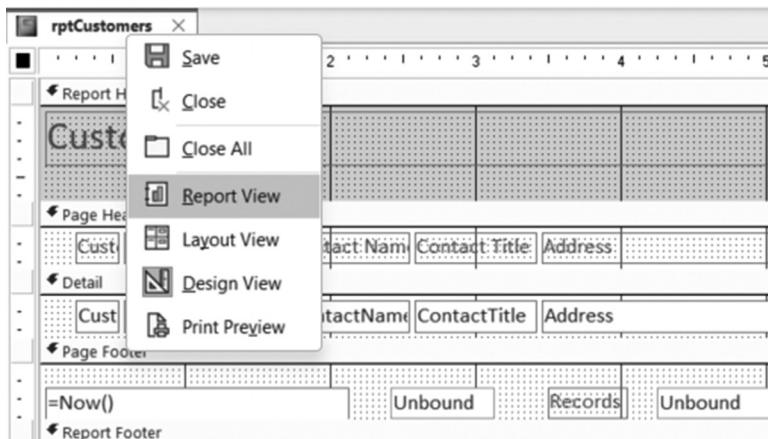


FIGURE 22.6. Access reports can be displayed using four different views: Report View, Layout View, Design View, and Print Preview. The Layout view may not be available for some reports.



### Hands-On 22.9 Conditionally Formatting a Control in Report View

- In the Navigation pane, right-click the **rptCustomers** report created earlier and choose **Design View**.
  - Click the **Detail** section and activate the property sheet.
  - In the property sheet of the Detail section, click the **Event** tab, select **Event Procedure** from the drop-down box next to the **On Paint** property, then click the **Ellipsis** button (...).
- Access activates the Code window and writes the stub of the **Detail\_Paint** event procedure.
- Complete the code of the **Detail\_Paint** procedure as shown here:

```
Private Sub Detail_Paint()
    If Me.City.Value = "London" Then
        Me.City.ForeColor = vbBlue
    Else
        Me.City.ForeColor = vbBlack
    End If
End Sub
```

This event procedure will set the **ForeColor** property for a control called **City** to blue when the city name is London and display the names of all other cities in black. This procedure will be triggered when you open the report in Report or Layout view.

- Press **Ctrl+S** to save the changes in the Code window.

6. Press **Ctrl+F11** to return to the main Access window.
7. Click the **View** button in the Views group of the Report Design tab.
8. Enter \* (asterisk) to view all customers. Press **Ctrl+F** to activate the Find dialog box. Enter **London** (change Look In field to Current document) and click **Find Next**. Access locates the first customer who lives in London. Click **Find Next** again to locate the next customer. Notice that all the occurrences of “London” are shown in blue.
9. Close the rptCustomers report.

## SORTING AND GROUPING DATA

---

Access offers users a convenient interface for grouping data, adding totals, and filtering. These features are available from a separate Group, Sort, and Total pane as shown in Figure 22.7. To work with this pane, open the report in Layout view, click the Report Layout Design tab, and select the Group & Sort button in the Grouping & Totals section. When you click on the Add a Group or Add a Sort buttons in the pane at the bottom of the report, Access will walk you through the steps required to create new report groups, add totals, or sort (Figure 22.8).

Cust ID	Company Name	Contact Name	Contact Title	Address	City	Region	Postal Code	Country	Phone	Fax
ALFK	Alfreds Futterkiste	Maria Anders	Sales Rep	Reise	Obere Str. 57	Dresden	12205	Germany	(030)-00743	(030)-0
ANAT	Ana Trujillo Empa	Ana Trujillo	Owner		Avda. de la Constitución 2	México City	05021	Mexico	(5) 555-47	(5) 55
ANTO	Antonio Moreno	Antonio Moreno	Owner		Mata	México City	05022	Mexico	(5) 555-39	
ARO	Around the Horn	Thomas Hard	Sales Rep	Reise	120 Hanover Sq.	Dover	WA1	UK	(171) 555-	(171)
BERI	Berglunds snabbköp	Christina Berg	Order Admin		Berguvsvägen 8	Luleå	S-958	Sweden	(0921)-12 34	(0921)-
BLAI	Blauer See Delika	Hanna Moen	Sales Rep	Reise	Forsterstr. 57	Mannheim	6830	Germany	(0621)-0846	(0621)-
BLO	Blondel père et fil	Frédérique C	Marketing M	Reise	24, place Kléber	Strasbourg	6700	France	88.60.15.3	88.60
BOLB	Bólido Comidas p	Martin Sommer	Owner		C/ Araquil, 67	Madrid	28002	Spain	(91) 555-22	(91) 5
BONB	Bon app'	Laurence Leb	Owner		12, rue des Bouchers	Marseille	13008	France	91.24.45.4	91.24
BOTB	Bottom-Dollar M	Elizabeth Lin	Accounting M	Reise	23 Tsawassen Blvd.	Tsawwassen	BC	Canada	(604) 555-	(604)
BSBI	B's Beverages	Victoria Ash	Sales Rep	Reise	Fauntleroy Circus	London	EC2	UK	(171) 555-	
CAC	Cactus Comidas p	Patricia Simpkins	Sales Agent		Cerrito 333	Bueno	1010	Argentina	(1) 135-55	(1) 13

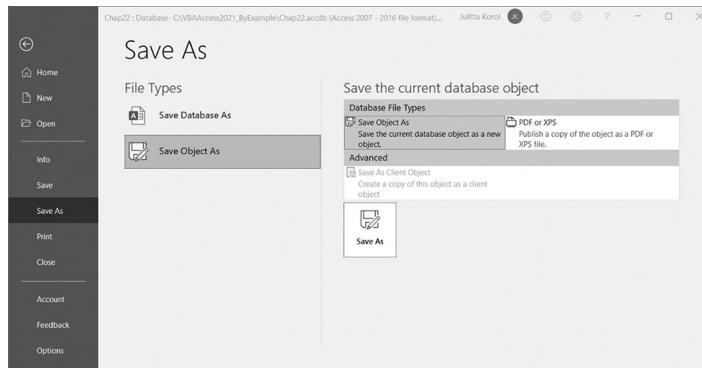
FIGURE 22.7. The Group, Sort, and Total pane provides a quick way to group and sort data, and add calculations in Access reports.

The screenshot shows a Microsoft Access report titled "All Customers". The report displays a table of customer information, including Cust. ID, Company Name, Contact Name, Contact Title, Address, City, Region, Postal Code, Country, and Phone/Fax numbers. The data is grouped by Country, as indicated by the grouping symbols in the report body. At the bottom of the screen, the "Group, Sort, and Total" pane is open, showing the current grouping settings: "Group on Country" with "A on top" selected. It also includes buttons for "Add a group" and "Add a sort".

**FIGURE 22.8** The Group, Sort, and Total pane indicates that the report is grouped on Country.

## SAVING REPORTS IN .PDF OR .XPS FILE FORMAT

Access reports can be saved to the .pdf or .xps format, as shown in Figure 22.9. The Portable Document Format (.pdf) preserves document formatting and makes files easy to distribute and print. Reports distributed as .pdf files retain their format and are protected so that the data may not be copied or changed. Another format that you can use for your report distribution is the XML Paper Specification (.xps) format, which also retains the format of the original document.



**FIGURE 22.9.** To save your report to .pdf or .xps file format, open it in the Report view, and click File | Save As. Select Save Object As, select PDF or XPS, and click the Save As button. Access will display the Publish As PDF or XPS dialog box where you can specify the required file format as well as the file name and destination folder.

## USING THE OPENARGS PROPERTY OF THE REPORT OBJECT

Like forms, Access reports have a very useful property called `OpenArgs` that you can use from a VBA code or a macro to pass a value to a report as the report is opened. Use the `OpenReport` method of the `DoCmd` object in the following form:

```
DoCmd.OpenReport (reportname, view, filtername,  
wherecondition, windowmode, OpenArgs)
```

The `OpenArgs` argument is a string expression of the Variant data type. You can pass multiple values in the `OpenArgs` argument by concatenating your values.

The `OpenArgs` property can be used to set a report format or to determine what data the report should display. With the `OpenArgs` property, you can reuse the same report, instead of creating a new report for a similar requirement.

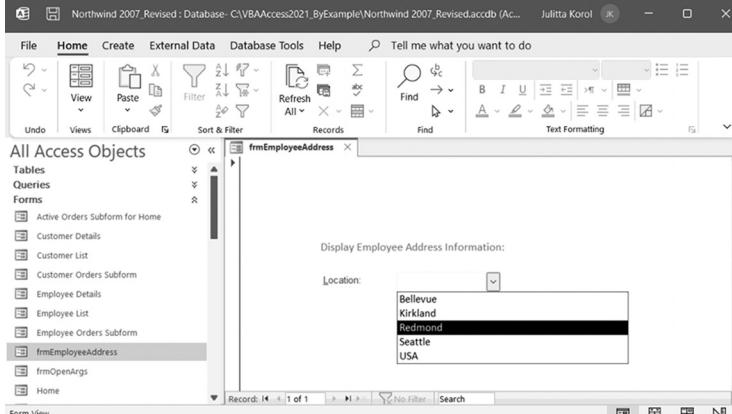
The Hands-On 22.10 demonstrates how to filter a report with the help of the `OpenArgs` property.



### Hands-On 22.10 Using the OpenArgs property to filter an Access report

1. Open the **Northwind 2007\_Revised.accdb** database in your **C:\VBAAccess 2021\_ByExample** folder. Cancel out of the Login dialog box.
2. In the Navigation pane on the left, locate and double-click the **frmEmployeeAddress**.

You should see the form as shown in Figure 22.10.



**FIGURE 22.10** The form used to filter the Employee Address Book report by City or Country/Region.

3. Choose **Redmond** from the combo box.

Access executes the following code in the `cboReports_AfterUpdate` event procedure:

```
Private Sub cboReports_AfterUpdate()
    Dim strFilterBy As Variant
    Dim strRpt As String

    strRpt = "Employee Address Book"

    If SysCmd(acSysCmdGetObjectState, acReport, _
        strRpt) <> 0 Then
        DoCmd.Close acReport, strRpt
    End If

    strFilterBy = Me.cboReports.Value
    DoCmd.OpenReport ReportName:=strRpt, _
        View:=acViewReport, _
        OpenArgs:=strFilterBy

End Sub
```

This event procedure closes the Employee Address Book report if it is open. The `SysCmd` method is used here to return the state of a specified database object. Use this method to find out whether the object is open, is a new object, or has been changed but not saved. For more information on using this method in your VBA procedures, see the online help.

Next, the procedure stores the selected value in the `strFilterBy` variable. This variable is then referenced in the `OpenArgs` property when the report is opened with the `OpenReport` method. The `Report_Load` event procedure of the Report\_Employee Address book (see the code below) then checks the `OpenArgs` property for the Null value. If the property is not Null, the `strFilter` variable is set to contain filtering criteria for the City or Country/Region fields. If you selected Redmond from the form's combo box, the `strFilter` will be set to `City = 'Redmond'`. The statement `Me.OpenArgs` returns the value stored in the `OpenArgs` property. With the filtering expression set, all you need to do is tell Access to turn the filter on by using the `FilterOn` property and set the `Filter` property to the `strFilter` variable.

```
Private Sub Report_Load()
    Dim strFilter As String

    If IsNull(Me.OpenArgs) Then
```

```
    Exit Sub
Else
    If Me.OpenArgs = "USA" Then
        strFilter = "[Country/Region] = '" &
                    Me.OpenArgs & "'"
    Else
        strFilter = "City = '" & Me.OpenArgs & "'"
    End If
    Me.FilterOn = True
    Me.Filter = strFilter

End If
End Sub
```

After the procedure finishes executing its code, you should see the Employee Address Book filtered by Redmond or whatever item you specified in the form's combo box (Figure 22.11).

The screenshot shows a Microsoft Access report titled "Employee Address Book". The report has a header section with the title and a timestamp "Monday, January 17, 2022 10:31:27 PM". Below the header is a table with columns: Employee Name, Address, City, State/Province, Zip/Postal Code, and Country/Region. The table has three rows of data. To the left of the table, there are vertical column headers "G", "K", and "N" corresponding to the first three letters of the employee names. The data is as follows:

Employee Name	Address	City	State/Province	Zip/Postal Code	Country/Region
Laura Giussani	123 8th Avenue	Redmond	WA	99998	USA
Jan Kotas	123 3rd Avenue	Redmond	WA	99998	USA
Michael Nepper	123 6th Avenue	Redmond	WA	99998	USA

FIGURE 22.11 This report was filtered by using the value passed in the OpenArgs property.

4. To filter the report again, make another selection from the form's combo box.

## RUNNING BUILT-IN MENU COMMANDS FROM VBA

Access has a special `RunCommand` method that allows you to run various built-in menu commands from VBA. This method requires that you pass to it an `acCommand` constant. Let's look at some of the commands related to reports that you can try out right from the Immediate window.

1. In the Access Navigation pane, highlight any table, for instance, Products.
2. Switch to the Visual Basic Editor Window and press **Ctrl+G** to activate the Immediate Window.

3. Type the following command and press Enter:

```
RunCommand acCmdNewObjectReport
```

Access displays the Report Wizard dialog box and displays the fields in the Products table that are available for your report. Notice that this command is equivalent to clicking the Report Wizard button in the Reports group of the Ribbon's Create tab.

4. Press Cancel to Exit the Report Wizard dialog and click Cancel to dismiss the message box that Access displays.
5. In the Immediate Window, type the following command and press Enter:

```
RunCommand acCmdNewObjectDesignReport
```

This time, Access displays a blank Report Design View. This is the same as clicking the Report Design button in the Reports group of the Ribbon's Create tab.

6. Close Report1.
7. Back in the Immediate Window, type the following command and press Enter:

```
RunCommand acCmdNewObjectBlankReport
```

Access displays a blank Report in the Layout View. This is the same as clicking the Blank Report button in the Reports group of the Ribbon's Create tab.

8. Close Report1.
- Make sure that some table or select query is highlighted in the Navigation pane of the Access Application window, and enter the following statement in the Immediate Window:

```
RunCommand acCmdNewObjectAutoReport
```

Access creates a report based on the fields in the selected table or query. This is the same as clicking the Report button in the Reports group of the Ribbon's Create tab.

9. Close the report that Access created without saving it.

Below is an example VBA procedure that creates a new report based on the table passed to it.

```
Sub Create_NewReport(strTable As String)
    With DoCmd
        .SelectObject acTable, strTable, True
        .RunCommand acCmdNewObjectAutoReport
    End With
End Sub
```

We start this procedure by using the `SelectObject` method of the `DoCmd` object to select the specified table in the Database window. After that, we execute the `RunCommand` method that creates a report. All table fields are used in the report layout. To call this procedure, create another one like this:

```
Sub ExecuteCmd()
    Create_NewReport "Shippers"
End Sub
```

10. Notice that there is still one more button left in the Reports group of the Ribbon's Create tab that you need to test. The Labels button can be invoked by the following `RunCommand`:

```
RunCommand acCmdNewObjectLabelsReport
```

Now that you know how to execute Access built-in Ribbon commands for reports, you can try running similar commands for forms and other Access objects. To find out the list of constants that need to be passed to the `RunCommand` method, use the Object Browser in the VBE screen (Press F2, or choose View | Object Browser). In the Object Browser's Find box, enter `acCommand` and click Search. You should now see the list of members in the `AcCommand` class, that is, the constants that can be used with the `RunCommand` method. Selecting any of the constants and pressing the F1 key will open the Microsoft online documentation.

## CREATING A REPORT WITH VBA

---

As mentioned in the beginning of this chapter, you can create an Access report programmatically by using the `CreateReport` method of the `Application` object. This method creates an empty report with a page header, page footer, and a detail section. This report is assigned a default name, `Report1`. To add controls to the report, use the `CreateControlReport` method of the `Application` object. Because an empty report is not attached to any database table or query, when creating a report programmatically, you need to specify an existing data source for the report records or create a new data source. In the following project, you will create a custom report that displays data based on the parameter crosstab query. Crosstab queries are a powerful tool for data analysis. In this project, you will create a cross tab query first in the Query Design View and then by writing a VBA function. Next, you will write another VBA function that creates a report and populates it with the controls to display relevant data. In this function, you will also write VBA code to create report's event procedures. The report function will call the query function. Let's get started.



## Custom Project 22.1 Programming a Query and a Report from scratch

### Part I-Creating a Crosstab Query in the Query Design View

Crosstab queries calculate sums, averages, counts and other types of aggregate totals on records and are often used for reporting purposes. Cross tab query groups the data down the left side of the datasheet and across the top. This format makes the data easier to read, analyze, and compare.

When creating a crosstab query, you can only have one Column Heading and one Value heading to perform calculations on. You can have multiple Row Headings. Access built-in Crosstab Query Wizard can be helpful in creating simple crosstab queries. However, if you need to use more than one table or query in the Crosstab query, you will first need to create a separate query with the table you want to use. In this Part, you will create a Crosstab query based on three tables: Products, Orders, and Order Details. The data will be grouped by Product Name (Row Heading) and will display sales for each quarter of the year passed as a parameter to this query. Figure 22.12 shows the Design view of the query. Figure 22.12 displays the result after running this query.

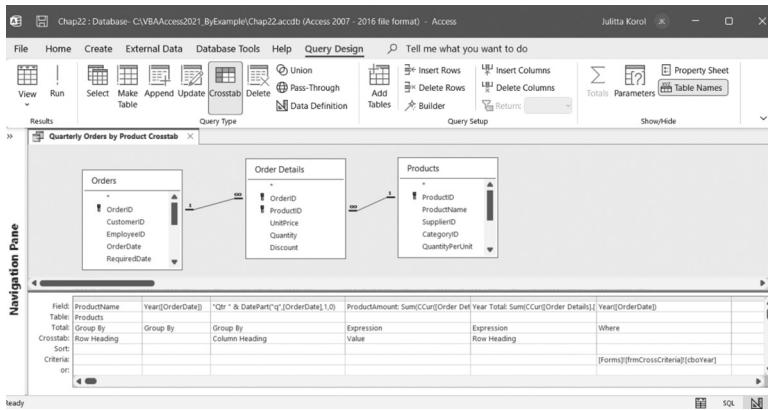


FIGURE 22.12 Crosstab Query Design View.

Product Name	Year Total	Qtr 1	Qtr 2	Qtr 3	Qtr 4
Alice Mutton	\$17,604.60	\$2,667.60	\$4,013.10	\$4,836.00	\$6,087.90
Aniseed Syrup	\$1,724.00	\$544.00	\$600.00	\$140.00	\$440.00
Boston Crab Meat	\$9,814.73	\$1,768.41	\$1,978.00	\$4,412.32	\$1,656.00
Camembert Pierrot	\$19,995.40	\$3,182.40	\$4,683.50	\$9,069.50	\$3,060.00
Carnarvon Tigers	\$15,950.00	\$1,500.00	\$2,362.50	\$7,100.00	\$4,987.50
Chai	\$4,779.00	\$705.60	\$878.40	\$1,066.50	\$2,128.50
Chang	\$6,658.55	\$2,435.80	\$228.00	\$1,871.50	\$2,123.25
Chartreuse verte	\$4,475.70	\$590.40	\$360.00	\$1,100.70	\$2,424.60
Chef Anton's Cajun Seasoning	\$5,214.88	\$225.28	\$2,970.00	\$1,337.60	\$682.00
Chef Anton's Gumbo Mix	\$373.62			\$288.22	\$85.40
Chocolade	\$1,282.01	\$744.60	\$162.56	\$68.85	\$306.00
Côte de Blaye	\$49,198.08	\$25,127.36	\$12,806.10	\$7,312.12	\$3,952.50
Escargots de Bourgogne	\$2,076.28		\$265.00	\$1,393.90	\$417.38
Filo Mix	\$2,124.15	\$187.60	\$742.00	\$289.80	\$904.75
Fløtemysost	\$8,438.74	\$2,906.80	\$174.15	\$2,541.29	\$2,816.50

FIGURE 22.13 Crosstab Query in Datasheet View.

1. In the main Access database window, choose **Create | Query Design**.  
At this point, you are in the empty window of the Select query (notice the **Select** button is highlighted on the Query Design Ribbon's tab).
2. From the list of tables, select **Products**, **Orders**, and **Order Details**, and add them to the Query Design View.
3. Close the Add Tables side pane.
4. Click the **Crosstab** button on the Ribbon to switch the Query type to Crosstab query.
5. Drag the **ProductName** from the **Products** table to the first column in the Query Design grid.
6. In the Crosstab row, select **Row Heading**.
7. In the Field row of the second column enter: **Year([OrderDate])** and set the Total row to **Group by**. When Access adds **Expr1** to the field name, leave it as is.
8. In the Field row of the third column, enter the following: “**Qtr “ & DatePart(“q”,[OrderDate],1,0)**”
9. Set the value of the Total row for this column to **Group By** and the Crosstab row to **Column Heading**.
10. In the Field row of the fourth column, enter the following: **ProductAmount: Sum(CCur([Order Details].UnitPrice\*[Quantity]\*(1-[Discount])/100)\*100)**
11. Set the value of the Total row for this column to **Expression** and the Crosstab row to **Value**.
12. In the Field row of the fifth column, enter: **Year Total: Sum(CCur([Order Details].[UnitPrice]\*[Quantity]\*(1-[Discount])/100)\*100)**

13. Set the value of the Total row for this column to **Expression** and the Crosstab row to **Row Heading**.
14. In the Field row of the sixth column, enter the following: **Year([OrderDate])**
15. Set the value of the Total row for this column to **Where** and in the Criteria row enter the following: **[Forms]![frmCrossCriteria]![cboYear]**  
This expression will filter the data by year passed from the combo box control on a form that will be created further in this project.

To prompt the user for the year when the query is running, we will add a parameter to the query. To use parameters in a crosstab query you must define them in the Query Parameters dialog box. When your query has parameters, you can easily get different sets of data without having to change the structure of your query for different criteria.

16. In the Show/Hide group of the Ribbon's Query Design, click the **Parameters** button.
17. Enter the parameter as shown in Figure 22.14. Notice that the parameter value is the same as the expression in the Where clause of the last column in the Query Design grid.

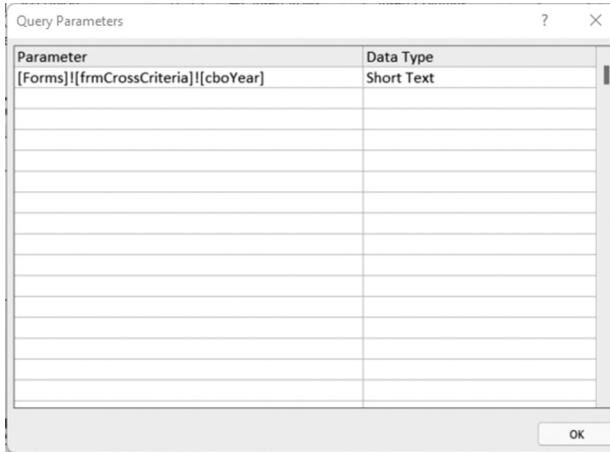


FIGURE 22.14 Use the Query Parameter dialog box to define parameters for the Crosstab Query.

18. Click **OK** to close the Parameters dialog box.
19. Save the completed query as **qryTestCrosstab**. Enter **1996** when prompted for the parameter.
20. Double-click the **qryTestCrosstab** in the Navigation pane to run it. Enter **1997** when prompted. The output of this query should match that shown in Figure 22.13.

21. Right-click the **qryTestCrosstab** tab and choose SQL view.

Access displays the SQL statement for the crosstab query you configured in the Query Design view, as shown in Figure 22.15.

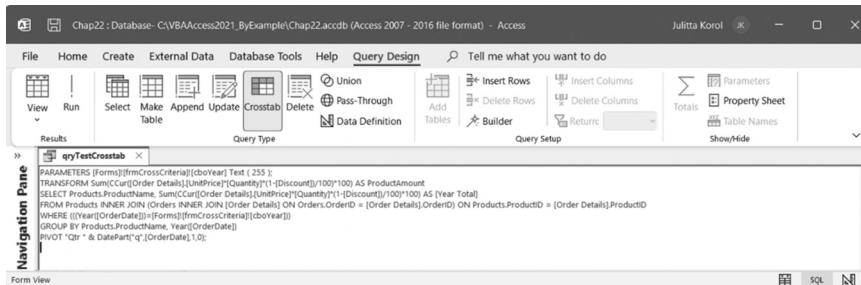


FIGURE 22.15 SQL statement for the completed Crosstab Query.

22. With a cursor placed anywhere in this statement, press **Shift+F2** to display the Zoom dialog box.  
You will need this statement when we write the VBA function to create the Crosstab query.
23. In the Zoom box, press **Ctrl+A** to select the entire statement. Press **Ctrl+C** to copy it to the Clipboard.
24. Close the Zoom dialog box and exit the query.

## ***Part II-Creating a Query with VBA***

When you must create a query programmatically, it helps to first create it manually in the Query Design View so that you can use the ready-made SQL statement for your programming code.

Let's use the SQL statement that is on your Clipboard to write the VBA function that creates the same query.

1. Switch to the Visual Basic Editor window and choose Insert | Module.
2. In the Module code window, enter the following function procedure.

```

Function CreateCrossTab_Qry(strQryName As String) As Boolean
Dim db As DAO.Database
Dim qryDef As QueryDef
Dim strSQL As String

On Error Resume Next
Set db = CurrentDb()
' test if the query with the specified name already exists
' in the QueryDefs collection of the current database
Set qryDef = db.QueryDefs(strQryName)
  
```

```

If Err.Number <> 0 Then
    Debug.Print Err.Number & " " & Err.Description
    Set qryDef = db.CreateQueryDef(strQryName)
    Err.Clear
End If

' prepare the SQL statement for the query
strSQL = strSQL & "PARAMETERS [Forms]![frmCrossCriteria]![cboYear] Text (255);"
strSQL = strSQL & "TRANSFORM Sum(CCur([Order Details].UnitPrice*[Quantity]*"
strSQL = strSQL & "(1-[Discount])/100)*100) AS ProductAmount "
strSQL = strSQL & "SELECT Products.ProductName, "
strSQL = strSQL & "      "Sum(CCur([Order Details].[UnitPrice]*[Quantity]*(1-[Discount])/100)*100) "
strSQL = strSQL & "AS [Year Total] FROM Orders INNER JOIN "
strSQL = strSQL & "(Products INNER JOIN [Order Details] "
strSQL = strSQL & "ON Products.ProductID = "
strSQL = strSQL & "[Order Details].ProductID) "
strSQL = strSQL & "ON Orders.OrderID = [Order Details].OrderID "
strSQL = strSQL & "WHERE ((Year([OrderDate])) = "
strSQL = strSQL & "[Forms]![frmCrossCriteria]![cboYear])) "
strSQL = strSQL & "GROUP BY Products.ProductName, Year([OrderDate]) "
"
strSQL = strSQL & "PIVOT ""Qtr "" & DatePart(""q"", [OrderDate],1,0)
"
strSQL = strSQL & "In (""Qtr 1"" , ""Qtr 2"" , ""Qtr 3"" , ""Qtr 4"" );"

Debug.Print strSQL
qryDef.SQL = strSQL

' check for error
If Err.Number <> 0 Then
    Debug.Print Err.Number & " " & Err.Description
    Exit Function
End If
' make the query persistant by appending it
' to the QueryDefs collection
db.QueryDefs.Append qryDef

' Close and refresh the Query object
qryDef.Close
End Function

```

The above function takes one argument, which is the name of the query we are going to create. This name will be passed by the function procedure that you

will write in the next section. The function will return a Boolean value of True/False, so we know whether the query was created successfully. We start by pointing the object variable db to the current database. In the current database, we will then set the object variable qryDef to point to the Query. Recall from earlier chapters that we use the `QueryDef` object to define a query. Next, we check if the query with the specified name already exists in the database. If we receive an error when trying to pass the query name to the `QueryDefs` collection, we know that the query does not exist, so we print the error number to the Immediate window and call the DAO `CreateQueryDef` method to create the query and we clear the error. If the query exists, we continue. Next, we prepare the SQL statement for the query. Notice how you may format the statement so it is easier to understand. The statement you copied from the query that you created manually in the previous section should be used here, but it must be formatted as shown. Notice that your statement differs a bit in the `PIVOT` clause from the one displayed in the above code. This change will allow us to display all four quarters regardless of whether the data is available and will make your report based on this query look more consistent. When preparing the SQL statement, pay attention to the extra spaces and double quotes. It helps to print the contents of the prepared SQL string variable to the Immediate window so you can spot any syntax errors. Once you have the correct SQL string, you must use it to set the `QueryDef.SQL` property. The SQL property contains the SQL statement that determines how records are selected, grouped, and ordered when the query is executed. The next error check point will let you know if there was a problem with the SQL statement. We will exit the function when the error occurs. You will need to fix the problem before attempting to run the query again. If all is fine, and no error was received, you must add the query to the `QueryDefs` collection to make it permanent in your database. After that, you can close the `QueryDef` object. You are done creating this query. If you need to create another query programmatically, all you need to do is change the SQL statement and pass a different name for the query.

3. Test this function by entering the following statement in the Immediate Window:

```
CreateCrossTab_Qry("Quarterly Orders by Product Crosstab")
```

4. In the Navigation pane of the Access database window, locate and run the newly created query. Enter **1996** when prompted.
5. Close the query.

Now, let's move on to creating our report.

### ***Part III-Creating a Report with VBA***

---

1. Return to the VBE screen and choose **Insert | Module**.
2. In the Module code window, enter the function procedure located in the companion files named **functionCreateReport\_Crosstab.txt**.

**Note:** You can also choose **File | Import File** in the VBE window and select **functionCreateReport\_Crosstab.bas** from the companion files.

3. Set several breakpoints in various parts of the function procedure and run it from the Immediate Window using this statement:

```
CreateReport_Crosstab 1996
```

4. Walk through the procedure by pressing the **F8** key. Refer to Chapter 9 if you need a refresher on the process of debugging the VBA code.

Notice that the function begins by calling the procedure that creates a crosstab query. Next, a reference is created to the crosstab query and a parameter is set. We also open a recordset based on the query. These tasks are performed in the following code snippet:

```
' call function to create query
CreateCrossTab_Qry (strQryName)

' get the parameter query
Set qdf = db.QueryDefs(strQryName)

' provide the parameter value
qdf.Parameters(0) = yr

' Open a Recordset based on the parameter query
Set rst = qdf.OpenRecordset()
```

Now we are ready to create a blank report and set some of its properties. Here is the code snippet:

```
'Create the Report object
Set rpt = CreateReport

'Add ReportHeader/Footer to the empty report
RunCommand acCmdReportHdrFtr
rpt.Caption = strQryName
```

Now it's time to create report controls. Each control is created using the **CreateReportControl** method, which requires the following parameters: **ReportName** and **ControlType**. There are also a few optional parameters that specify

the section that will contain the new control, the name of the parent control, the name of the field to which control will be bound if it is a data-bound control, the coordinates for the upper and left corner of the control in twips, and finally the width and height of the control in twips. The following code snippet demonstrates how we can add an image control to the ReportHeader section:

```
'Create report controls
Set rptCtl = CreateReportControl(rpt.Name, acImage, acHeader)
With rptCtl
    .Height = 400
    .Width = 400
    .Top = 0
    .Name = "Auto_Logo0"
    .Picture = CurrentProject.path & "\Images\redstar.gif"
End With
```

Our report has text boxes and labels, and here is how we can add them:

```
Set rptCtl = CreateReportControl(rpt.Name, acLabel, acPageHeader)
With rptCtl
    .Height = 280
    .Width = 1200
    .Top = 0
    .Caption = "Order Year:"
End With

Set rptCtl = CreateReportControl(rpt.Name, acTextBox,
                                 acPageHeader)
With rptCtl
    .Height = 280
    .Width = 1080
    .Top = 0
    .Left = 1250
    .ControlSource = "=[Forms]![frmCrossCriteria]![cboYear]"
End With
```

Note that all screen measurements are given in twips. A twip is equal to 1/20 point. One inch is 1440 twips and 1 cm is 567 twips. Twips are used to ensure that the placement and proportion of screen elements in your application will appear the same on all displays.

Next, we need to lay out the label text box controls for each field returned by the query. We do this with the help of the For ...Next loop and the Recordset object. See the code snippet below:

```

' Create corresponding label and text box controls for each field
For i = 0 To rst.Fields.Count - 1
    ' Create new text box control and size to fit data.
    Set txtNew = CreateReportControl(rpt.Name, acTextBox, _
        acDetail, , rst.Fields(i).Name, lngLeft, 0)
    If i = 0 Then
        txtNew.Width = 3500
    Else
        'txtNew.SizeToFit
    End If

    ' Create new label control and size to fit data.
    Set lblNew = CreateReportControl(rpt.Name, acLabel, _
        acPageHeader, , rst.Fields(i).Name, _
        lngLeft, 400, 1400, txtNew.Height)

    With lblNew
        .FontSize = 12
        .FontName = "Haettenschweiler"
        .TextAlign = 2
    End With

    ' Increment top value for next control
    If i = 0 Then
        lngLeft = lngLeft + txtNew.Width '+ 25
        Debug.Print lngLeft
    Else
        lngLeft = lngLeft + txtNew.Width '+ 25
        Debug.Print lngLeft
    End If
Next

```

In the following code, we make some visual adjustments to the report and put a timestamp in the footer section. We also specify the query as the report's RecordSource property.

```

With rpt
    ' Create timestamp on footer
    With CreateReportControl(.Name, acLabel, _
        acPageFooter, , Now(), 0, 0)
    End With

    .Section(acDetail).Height = 420
    .Section("detail").Height = 0
    .RecordSource = strQryName
    'make PageHeaderSection visible

```

```
.Section(3).Visible = True  
.DefaultView = 0 'show report in Print Preview  
End With
```

Before finishing out report creation tasks, we need to add some event procedures. We start by setting a reference to the Report Module. Next, we use the CreateEventProc method to create the first event procedure that will run when the report is opened. We also prepare the variable that will hold the procedure code.

```
' return reference to report module  
Set mdl = rpt.Module  
  
' add event procedure  
lngReturn = mdl.CreateEventProc("Open", "Report")  
strCode = strCode & vbCrLf & "Dim strDocName As String"  
strCode = strCode & vbCrLf & "strDocName = ""frmCrossCriteria"""  
strCode = strCode & vbCrLf & "' Open form."  
strCode = strCode & vbCrLf & "DoCmd.OpenForm strDocName"  
strCode = strCode & ", , , , , acDialog"  
strCode = strCode & vbCrLf & "If IsLoaded(strDocName) "  
strCode = strCode & "= False Then Cancel = True"
```

Notice that formatting the code of the event procedure in this manner can be very tedious and frustrating. To avoid errors, you must pay close attention to the single and double quotes, spaces, and commas. The good news is that there is a better way. Simply write your event procedures in a text file and add the text file to the module like this:

```
' Insert other event procedures from a text file  
mdl.AddFromFile "C:\VBAAccess2021_ByExample\ReportEvents.txt"
```

Before we are done with the report creation, we need to save the report and perhaps display it in the Print Preview. The final statements in the function are as follows:

```
'save the report with specified name  
DoCmd.Save , strRptName  
  
'open the report in Print Preview  
DoCmd.OpenReport strRptName, acViewPreview  
  
Set rpt = Nothing  
rst.Close
```

Finally, Figure 22.16 displays our completed report.

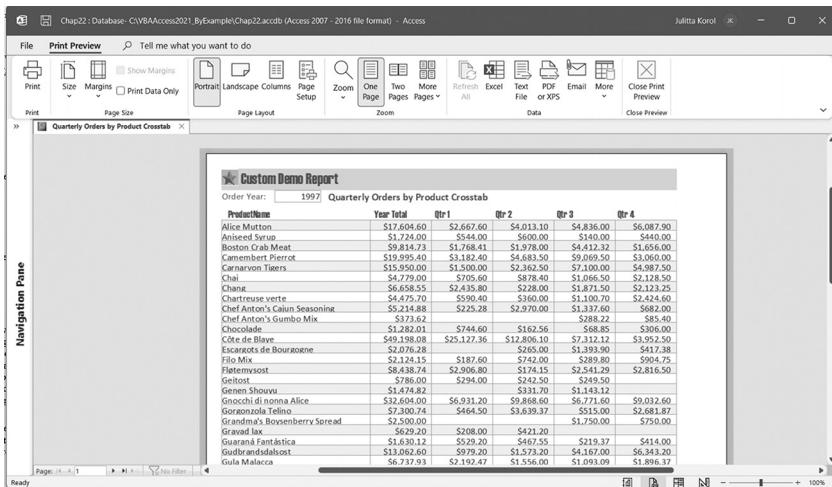


FIGURE 22.16 The Custom Demo Report was generated using the VBA code.

When you close the Print Preview, Access displays the Report in the Design View (Figure 22.17).

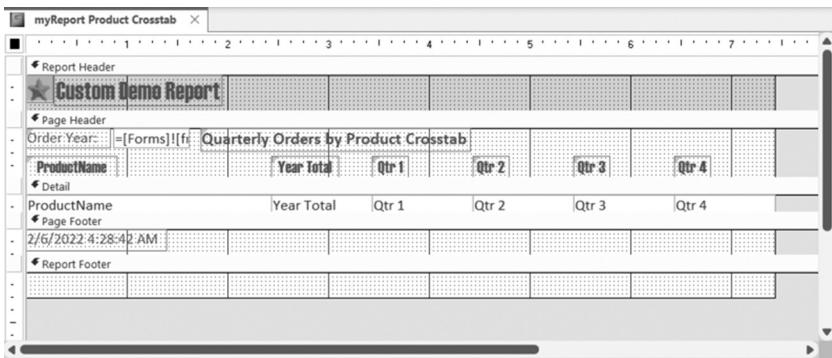


FIGURE 22.17 The Custom Demo Report in the Design View.

Now that we have the report ready, let's see how you can provide a user-friendly interface for running it.

#### ***Part IV-Creating a Custom Form for the Query's Parameters***

Prompting the user for parameters should be a pleasant experience. The user may not know what type of parameter is expected or what range of data is available for a specific report. When a report is based on a parameter query, creating a customized form for the query's criteria will make it less cumbersome.

some for the user to use your custom report. In this section, you will create the form shown in Figure 22.18. This form will include a combo box listing the years for which the report can be requested. The command button will run the report only when the selection is made from the combo box. Let's set this up.

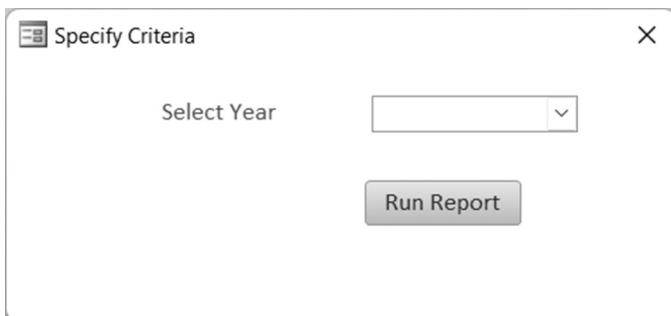


FIGURE 22.18 This custom form is used to provide the parameter for the report.

1. In the Access database window, choose Create | Form Design to create a blank form.
2. In the Form Design Tools group, click the Property Sheet button.
3. Click a combo box control in the Controls section of the Ribbon and click in the Form Detail section. The control should appear in its default size. Drag the combo control to position it, as shown in Figure 22.18. Click on the combo control's label to select it and in the Property Sheet, click the All tab, and change the **Caption** property to **Select Year**. Click the Unbound combo box control and in the Property Sheet, change its **Name** property to **cboYear**. In the Property Sheet, click the Data tab. In the Row Source property, enter the following statement:

```
SELECT DISTINCT Year([OrderDate]) FROM Orders GROUP BY [OrderDate];
```

This will provide the combo box with the list of available years for the report. To get more room for data entry, you can press Shift+F2 when the cursor is in the Row Source property field.

Set other property values for the cboYear control as follows:

Limit To List	Yes
Allow Value List Edits	No
Show Only Row Source Values	Yes

4. Click the command button control in the Controls section of the Ribbon and click in the Form Detail section. The control should appear in its default size. Drag the command button to the position shown in Figure 22.18. While the command button is selected, use the Property Sheet to change the **Name** property to **cmdRunProductsRpt** and the **Caption** property to **Run Report**. In the Property Sheet, click the **Event** tab and set the On Click event to [Event Procedure], then click the builder button (...). Access will open the report module with the stub of the Click procedure. Complete this procedure as follows.

```
Private Sub cmdRunProductsRpt_Click()
    Dim strRptName As String
    Dim strMsg As String
    Dim strTitle As String

    strRptName = "myReport Product Crosstab"
    strMsg = "Please Open the report: " & strRptName
    strTitle = "Open the Report"

    'check if report is open
    If Application.CurrentProject.AllReports(strRptName) .
        IsLoaded = True Then
        If IsNull(Me.cboYear) Or Me.cboYear = "" Then
            MsgBox "Please select value for the year.", _
                vbInformation, "Year selection?"
            With Me.cboYear
                .SetFocus
                .Dropdown
            End With
        Else
            ' Hide form.
            Me.Visible = False
        End If
    Else
        MsgBox strMsg, vbInformation, strTitle
        'close the form
        DoCmd.Close
    End If
End Sub
```

This procedure will ensure that criteria was selected from the dropdown and the user runs the report not by launching the form but clicking the report name in the Navigation pane.

Press the Save button on the toolbar to make sure your changes are saved. When prompted for the form name, enter **frmCrossCriteria** for its name.

Let's continue by setting the form properties.

5. Back in the Form Design view, use the Property Sheet's combo box to select **Form**. Click the Format tab and set the form's **Caption** property to **Specify Criteria**. Set other property values for the form as follows:

Border Style	Dialog
Record Selectors	No
Navigation Buttons	No
Scroll Bars	Neither
Width	3.9583"

Set the Form's height to 1.5" by dragging the sizing handle of the form (Figure 22.19).



FIGURE 22.19 Resizing the form.

The criteria form is now set up. Close the form and save your changes before proceeding to the next section.

#### ***Part V-Running the Form and Report***

1. In the Navigation pane double click the frmCrossCriteria form you created in the previous section.
2. When Access opens the form, click the Run Report button.

The click event attached to this button checks whether the specified report is open. The Else clause of the event procedure will run now because you have not clicked the report name in the Navigation pane. Click OK to the message that tells you that you should open the report. Access closes the form.

3. In the Navigation pane of Access database window, double-click the **myReport Product Crosstab**.

Access opens the **frmCrossCriteria** where you are prompted to specify year. Note that if you are working with multiple monitors, the form may pop up on a different screen.

4. Select any year from the drop-down box and click **Run Report**.

Access opens the report in the Print Preview where you have options to print it, email it, or save it in other formats.

5. Close the Report Print Preview.

#### Action Item 22.1

On your own add another command button to the **frmCrossCriteria** form. Set its name property to **cmdCancel** and its **Caption** property to **Cancel**. Write an event procedure for the **cmdCancel** button's **Click** event. The code of this procedure should close the form when the **Cancel** button is clicked.

---

## SUMMARY

In this chapter, you discovered various ways of creating reports in Access and learned how you can extend your reports by incorporating some VBA code. You worked with several events that fire when the report is run. By writing your own event procedures you can specify what happens when the report is opened, activated, deactivated, or closed. You can also display a custom message when an error occurs or the report does not contain any data, or you can make last-minute changes to the report format before it is printed or previewed. In the last section of this chapter, you were introduced to the `OpenArgs` property of the `Report` object and learned how to use it to filter a report.

This chapter barely scratched the surface of what is possible and doable with reports. There are numerous templates in Access 2021 that you can study to gain more insight into designing very appealing, informative, and interactive reports.

In the next chapter, you will be working with the menu interface in Access, commonly referred to as `RibbonX`.

Part

# VI

## *ENHANCING THE USER EXPERIENCE*

Since its 2007 release, Access, like other Microsoft 365 applications, uses the Ribbon interface for its menu system. Knowing how the Ribbon works and how you can modify it to customize your Access databases will enhance the experience of your database users. We will cover this topic in one big chapter with numerous illustrated hands-on examples and programming code written in VBA and XML.

### Chapter 23 Customizing the Menu System in Access



# Chapter 23

## CUSTOMIZING THE MENU SYSTEM IN ACCESS

**B**eginning with its 2007 version, Access, like other Microsoft 365 applications, uses the RibbonX interface for its menu system. The newest update to the Office UI (User Interface) was implemented at the end of 2021 for all Microsoft 365 and Access 2021 users. It includes a more rounded look to the Office ribbon bar and small changes in the design of various buttons. The new Office UI is designed to match the visual changes in Windows 11.

This chapter provides an overview of the programming elements available in the Ribbon and shows how you can customize the menu system in your Access database applications. In the following sections, we will examine many examples that demonstrate the use of different menu features. Because RibbonX is based on XML, if you are new to XML, I recommend you start with Chapter 27 (XML Features in Access 2021) before working through the examples in this chapter.

### THE INITIAL ACCESS 2021 WINDOW

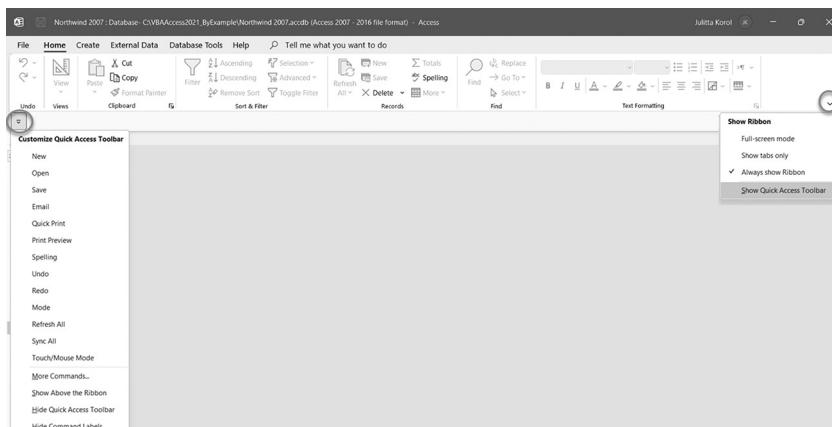
---

When you launch Access, you are presented with a button for creating a blank Access database plus a search bar where you can search online for prebuilt database templates. The templates come with ready-to-use tables, forms, reports, queries, relationships, and macros that can be modified as needed.

Once you open an existing Access database or create a new one, you see a rounded Ribbon bar with various options. The Quick menu Access toolbar that was positioned above the File tab in the previous version of Access is no longer there by default. Instead, to see the Quick Access Toolbar, you need to click the drop-down arrow at the end of the Ribbon to reveal other ribbon options, the last of which will allow you to Show Quick Access Toolbar. Once selected, Access adds an extra bar under the Ribbon, as shown in Figure 23.1. There, you can quickly access the most frequently used commands. The Quick Access toolbar allows you to add as many commands as you wish. You can access built-in commands in the Quick Access toolbar by using any of the following methods:

- Click on the drop-down arrow in the Quick Access toolbar and select a command you want to add or click More Commands. Access will add the chosen command to the empty bar below the Ribbon.
- Click File | Options and choose Quick Access Toolbar.

Note that the Quick Access Toolbar can be also positioned in the same area as in the previous versions of Access, in the title bar. Together with the title bar and the tabs, the Quick Access toolbar belongs to a large rectangular area called the Ribbon. This area is positioned at the top of the UI window. The Quick Access toolbar was designed for the convenience of end users. Developers should not alter this toolbar. However, if you have a valid reason to hide the contents of this toolbar or add other buttons to it, you can apply your own customizations.



**FIGURE 23.1** The circles in this screenshot indicate the areas that turn on and reveal the Quick Access Toolbar.

## CUSTOMIZING THE NAVIGATION PANE

When an Access database is open, you can easily access all of your objects via the Navigation pane on the left side of the window (see Figure 23.2). If you need more screen real estate, you can hide the Navigation pane by clicking on the Shutter Bar Open/Close button (the double arrow at the top of the pane) or by pressing F11.

Use the Navigation pane to organize your objects by type, date created or modified, or related table, or create your own custom groups of objects. By clicking on the down arrow button at the top of the Navigation pane, you can define how you view and manage database objects (see Figure 23.3). To sort and filter your database objects, activate the Search Bar, or access the navigation options, right-click on the top bar of the Navigation pane (Figure 23.4).

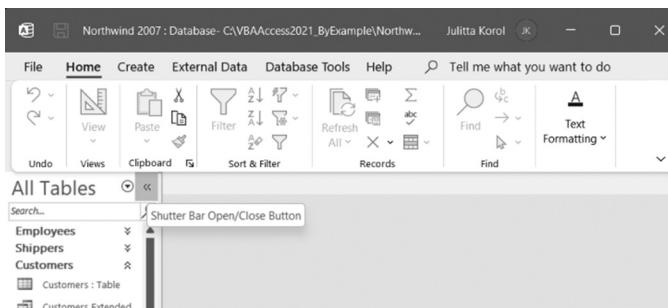


FIGURE 23.2 The Navigation pane in Access 2021.

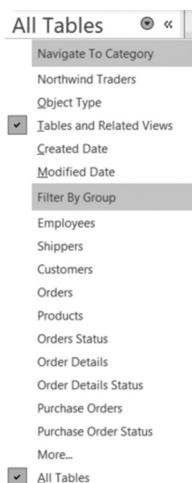


FIGURE 23.3 Grouping options in the Navigation pane.

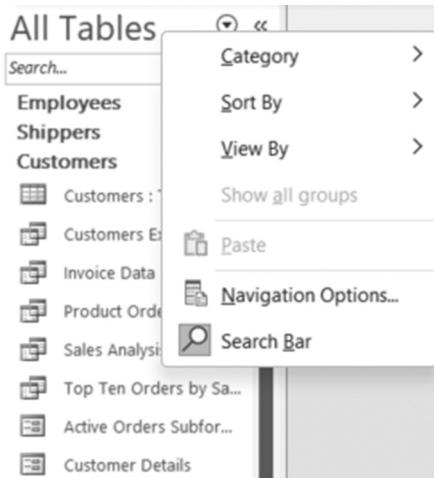


FIGURE 23.4 Objects in the Navigation pane can be easily categorized, sorted, and filtered. Use the Search Bar to locate a hard-to-find object. Use the Navigation Options tool to create custom groups of objects.

The Navigation Options dialog box (see Figure 23.5) allows you to create any number of custom groups for organizing your objects according to specific database needs.

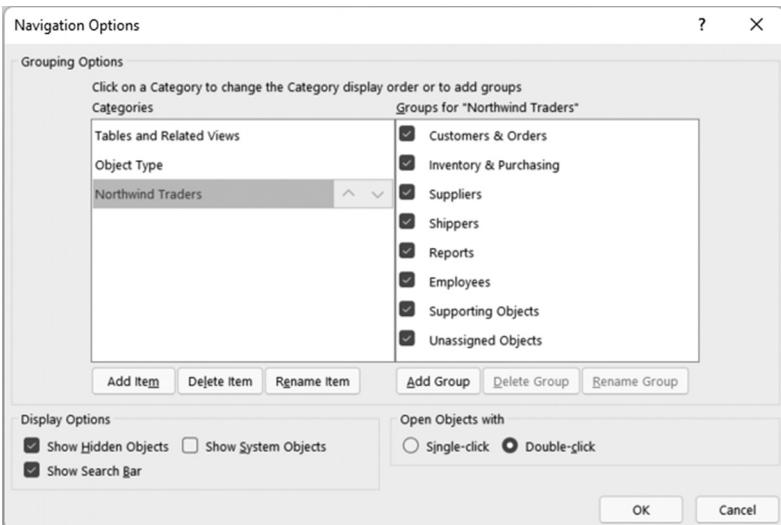


FIGURE 23.5 The Navigation Options dialog box.

The exercise in Hands-On 23.1 will walk you through the process of creating a custom group in the Navigation pane to track your development efforts.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



### Hands-On 23.1 Adding a Custom Group to the Navigation Pane

1. Open the C:\VBAAccess2021\_ByExample\Northwind 2007.accdb database. Cancel the login box.
2. Right-click the top bar of the Navigation pane and choose **Navigation Options** (see Figure 23.4).
3. Click the **Add Item** button and type a new name for the category: **Objects in Development**.
4. While the new Objects in Development category is selected, click the **Add Group** button and enter **Dev Tables** for the new group name.
5. Click the **Add Group** button again to add another group named **Dev Queries**.
6. Add two more groups under Objects in Development named **Dev Forms** and **Dev Reports**. See Figure 23.6 for the final output.
7. Click **OK** to close the Navigation Options dialog box.
8. Click on the down arrow button at the top of the Navigation pane and choose **Objects in Development** (Figure 23.7).

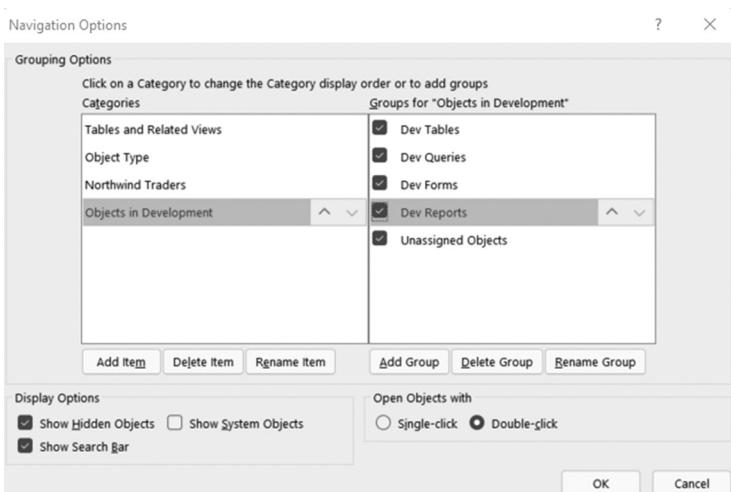


FIGURE 23.6 Creating custom groups in the Navigation Options.

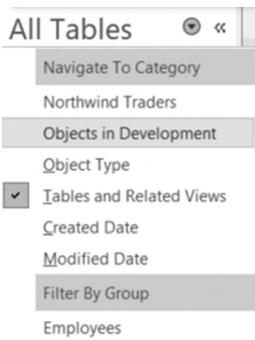


FIGURE 23.7 Displaying a custom group in the Navigation pane.

The next logical step is placing some database objects into your custom groups. Hands-On 23.2 requires prior completion of Hands-On 23.1.



## Hands-On 23.2 Assigning Objects to Custom Groups in the Navigation Pane

1. In the Northwind 2007.accdb database you opened in Hands-On 23.1, right-click the **Customers** table and choose **Copy**.
2. Right-click anywhere in the Navigation pane and choose **Paste**. In the Paste Table As dialog box, enter **Companies** for the new name of the table and select the **Structure Only** option button. Click **OK** to exit the dialog box.
3. Choose **Objects in Development** from the drop-down list at the top of the Navigation pane.
4. Drag the **Companies** table from the Unassigned Objects group to the **Dev Tables** group.

When you place a database object into a custom group in the Navigation pane, Access creates a shortcut to this object (see Figure 23.8). You can rename your shortcut by right-clicking its name and choosing **Rename Shortcut**. You can also hide the shortcut in the group or remove it from the group, provided the Navigation pane has not been locked (this is discussed in the next section). In addition to the **Hidden** attribute, each shortcut has a **Disable Design View** shortcuts attribute you can set to prevent users from switching to Design view when using the shortcut. You must restart your Access database for this property change to take effect. To display the shortcut properties as shown in Figure 23.8, right-click on the **Companies** shortcut under the **Dev Tables** group and choose **Table Properties**. Remember that you can drag any object listed in the Unassigned Objects group into your custom groups.

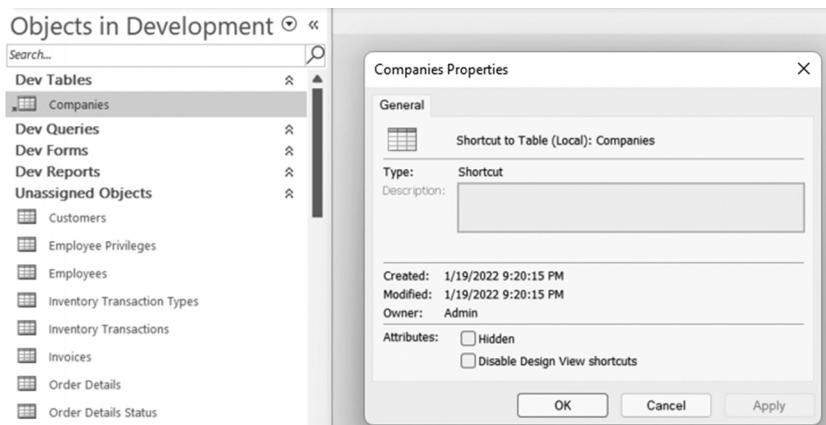


FIGURE 23.8 Navigation pane with custom groupings.

## USING VBA TO CUSTOMIZE THE NAVIGATION PANE

---

You can lock down and customize the Navigation pane programmatically by using the following methods of the DoCmd object: `LockNavigationPane`, `NavigateTo`, and `SetDisplayedCategories`. There are also two methods of the Application object (`ExportNavigationPane` and `ImportNavigationPane`) that enable you to quickly apply the same Navigation pane customizations to any other Access database.

### **Locking the Navigation Pane**

---

To prevent users from deleting database objects that are displayed in the Navigation pane, use the following statement:

```
DoCmd.LockNavigationPane True
```

The `LockNavigationPane` method of the DoCmd object requires a `Lock` argument. Use the Boolean value of `True` to lock the Navigation pane and `False` to unlock it.

### **Controlling the Display of Database Objects**

---

To automatically navigate to a specific category in the Navigation pane upon startup of your Access database or to display only certain objects in the category, use the `NavigateTo` method of the DoCmd object. This method takes two arguments: `Category` (required) and `Group` (optional). The `Category` argument specifies the category you want to navigate to. This argument can be the name

of your custom category, such as the Objects in Development category you created in Hands-On 23.1, or a constant representation of the Object Type, Tables and Views, Created Date, and Modified Date categories. The `Group` argument is optional. If you omit it, the Navigation pane will display all database objects arranged by the criteria specified in the `Category` argument. See Table 23.1 for valid `Group` arguments for the various `Category` arguments.

**TABLE 23.1** Category and Group arguments used in the `NavigateTo` method

Category Argument	Category Argument Constant	Group Argument	Group Argument Constant
Object Type	acNavigationCategoryObjectType	Tables Forms Reports Queries Pages Macros Modules	acNavigationGroupTables acNavigationGroupForms acNavigationGroupReports acNavigationGroupQueries acNavigationGroupPages acNavigationGroupMacros acNavigationGroupModules
Tables and Views	acNavigationCategoryTablesAndViews	Name of a specific table or view in your database	
Modified Date	acNavigationCategoryModifiedDate	Today Yesterday Last Month Older	acNavigationGroupToday acNavigationGroupYesterday acNavigationGroupLastMonth acNavigationGroupOlder
Created Date	acNavigationCategoryCreatedDate	Today Yesterday Last Month Older	acNavigationGroupToday acNavigationGroupYesterday acNavigationGroupLastMonth acNavigationGroupOlder
Custom	Name of your custom category	Name of one of the custom groups you have created for the specified custom category	

Let's get some practice with the `NavigateTo` method.

### **Hands-On 23.3 Using the `NavigateTo` Method to Control the Display of Database Objects in the Navigation Pane**

1. In the Northwind 2007.accdb database, press **Alt+F11** to switch to the Visual Basic Editor window. Press **Ctrl+G** (or choose **View | Immediate Window**) to open the Immediate window.
2. In the Immediate window, type each of the following `DoCmd.NavigateTo` statement examples on one line, pressing **Enter** to execute each statement. After the execution of each statement, check the resulting display in the

Navigation pane of the Access main window. Two monitors create the perfect environment for this exercise.

```
DoCmd.NavigateTo "acNavigationCategoryCreatedDate"
```

This statement will navigate to the Created Date category and display all database objects.

```
DoCmd.NavigateTo "acNavigationCategoryObjectType",  
"acNavigationGroupForms"
```

This statement will navigate to the Object Type category and select the Forms group.

```
DoCmd.NavigateTo "acNavigationCategoryTablesAndViews",  
"Invoices"
```

This statement will navigate to the Invoices table in the Tables and Views category.

```
DoCmd.NavigateTo "Objects in Development", "Dev Tables"
```

This statement will navigate to the Dev Forms group objects in the Objects in Development category created in Hands-On 23.1.

```
DoCmd.NavigateTo "acNavigationCategoryModifiedDate",  
"acNavigationGroupOlder"
```

This statement will navigate to the Modified Date category and display all the database objects beginning with a date earlier than the previous month.

Figure 23.9 displays all the statements entered in the Immediate window.

3. In the VBE window, press **Ctrl+G**, **Ctrl+A**, then the **Delete** key to remove the contents of the Immediate window.



FIGURE 23.9 Testing Navigation statements in the Immediate Window.

### Setting Displayed Categories

The `SetDisplayedCategories` method of the `DoCmd` object is used to specify which categories should be displayed under `Navigate To Category` in the title bar of the Navigation pane. Use this method to show and hide groups from

the top bar of the Navigation pane. For example, the following statement will remove the custom category Objects in Development from the Navigation pane titlebar's drop-down list:

```
DoCmd.SetDisplayedCategories False, "Objects in Development"
```

Notice that the `SetDisplayedCategories` method uses two arguments. The first argument specifies whether to show or hide the category. Use the Boolean value of `False` to hide the category specified in the second argument of this method or `True` to show the category. The second argument denotes the name of the category you want to show or hide. Do not specify this argument if you want to show or hide all categories.

### **Saving and Loading the Configuration of the Navigation Pane**

---

The configuration of the Navigation pane can be saved at any time with the `ExportNavigationPane` method of the Application object. This method requires one argument—the path and the name of the XML file where you want to save the configuration of the Navigation pane. For example, the following statement entered on a single line in the Immediate Window will save the current configuration of the Navigation pane to North2007NavConfig.xml in the C:\VBAAccess2021\_ByExample folder:

```
Application.ExportNavigationPane "C:\VBAAccess2021_ByExample\North2007NavConfig.xml"
```

To load a saved Navigation pane configuration from the XML file, use the `ImportNavigationPane` method of the Application object:

```
Application.ImportNavigationPane "C:\VBAAccess2021_ByExample\North2007NavConfig.xml", False
```

Notice that the `ImportNavigationPane` method used in the previous statement has two arguments. The first one specifies the path and name of the XML file that contains the Navigation pane configuration to load. The second argument is optional. When set to `True`, the imported categories will be appended to the existing categories. The default value is `False`.

Hands-On 23.4 demonstrates how to save the current configuration of the Navigation pane and then load it into another Access database.



### Hands-On 23.4 Saving and Loading the Configuration of the Navigation Pane

1. In the VBE window of the Northwind 2007.accdb database, type the following statement on one line in the Immediate window and press **Enter** to execute:

```
Application.ExportNavigationPane "C:\VBAAccess2021_ByExample\North2007NavConfig.xml"
```

2. Switch to File Explorer and check that the North2007NavConfig.xml file is in the VBAAccess2021\_ByExample folder.
3. Double-click the filename to open it in the browser. Figure 23.10 displays the partial content of the configuration file.

The XML file contains the objects and structure of the Access Navigation pane. This file includes information about the contents of the Navigation pane system tables: MSysNavPaneGroupCategories, MSysNavPaneGroups, MSysNavPaneGroupToObjects, and MSysNavPaneObjectIDs.

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema" xmlns:od="urn:schemas-microsoft-com:officedata">
  <xsd:schema>
    <xsd:element name="dataroot">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="MSysNavPaneGroupCategories" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="MSysNavPaneGroupToObjects" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="MSysNavPaneObjectIDs" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="generated" type="xsd:dateTime"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="MSysNavPaneGroupCategories">
      <xsd:annotation>
        <xsd:appinfo>
          <xsd:index index-name="Id" index-key="Id" primary="yes" unique="yes" clustered="no" order="asc"/>
        </xsd:appinfo>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="Filter" minOccurs="0" od:jetType="text" od:sqlType="nvarchar">
            <xsd:simpleType>
              <xsd:restriction base="string">
                <xsd:minLength value="255"/>
                <xsd:maxLength value="255"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="Flags" minOccurs="0" od:jetType="longinteger" od:sqlType="int" type="xsd:int"/>
          <xsd:element name="Id" minOccurs="1" od:jetType="autonumber" od:sqlType="int" od:autoUnique="yes" od:nonNullable="yes" type="xsd:int"/>
        </xsd:sequence>
        <xsd:element name="Name" minOccurs="0" od:jetType="text" od:sqlType="nvarchar">
          <xsd:simpleType>
            <xsd:restriction base="oddstring">
              <xsd:minLength value="255"/>
              <xsd:maxLength value="255"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Position" minOccurs="0" od:jetType="longinteger" od:sqlType="int" type="xsd:int"/>
        <xsd:element name="SelectedObjectID" minOccurs="0" od:jetType="longinteger" od:sqlType="int" type="xsd:int"/>
        <xsd:element name="Type" minOccurs="0" od:jetType="longinteger" od:sqlType="int" type="xsd:int"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</root>
```

FIGURE 23.10 The current configuration of the Navigation pane is saved in this XML file.

4. Close the Browser window.
5. Close the Northwind 2007.accdb database.
6. Create a new Access database named **Load\_North2007NavConfig.accdb** in your C:\VBAAccess2021\_ByExample folder.
7. Click the top bar of the Navigation pane and view the Navigation pane title bar's drop-down list before proceeding to import the saved configuration file.

8. Press **Alt+F11**, then press **Ctrl+G** to activate the Immediate window. Type the following statement on one line and press **Enter** to execute:

```
Application.ImportNavigationPane "C:\VBAAccess2021_ByExample\North2007NavConfig.xml", False
```

9. Press **Alt+F11** to switch back to the Access application window.
10. Click the top bar of the Navigation pane and display the Navigation pane title bar's drop-down list again (see Figure 23.11). Notice the additional entries in the drop-down list: Northwind Traders and Objects in Development.

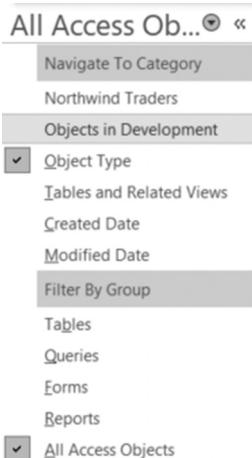


FIGURE 23.11 The Navigation pane can be easily modified using the external XML file containing the custom configuration settings.

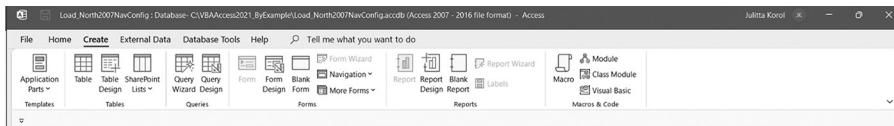
Now that you know how to manually and programmatically control the Navigation pane, you should find it easy to provide users with the needed customization of the Access database navigation system. The next section will expand your knowledge of the Access user interface by giving you a quick overview of the Ribbon.

## A QUICK OVERVIEW OF THE ACCESS 2021 RIBBON INTERFACE

---

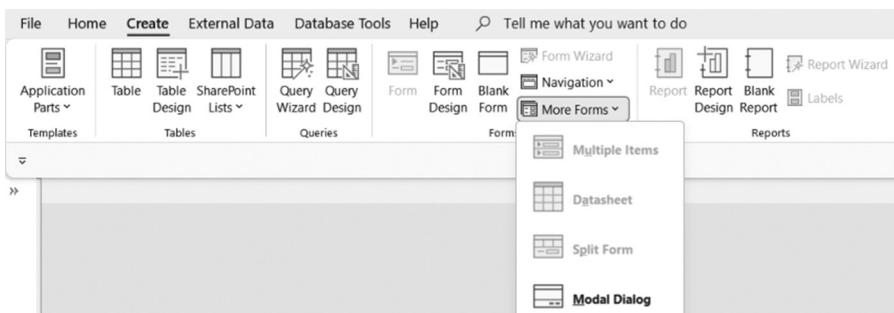
All Access program commands can be accessed from the Ribbon. Each tab on the Ribbon provides access to features and commands related to a particular database task. For example, you can use the Create tab to quickly create new

tables, forms, reports, queries, macros, modules, and Microsoft Windows SharePoint Services lists (see Figure 23.12). Related commands within a tab are organized into groups. For example, the Create tab divides its commands into six groups: Templates, Tables, Queries, Forms, Reports, and Macros & Code. This type of organization makes it easy to locate a particular command.



**FIGURE 23.12** All Access commands related to creating various database objects are grouped on the Create tab. The bottom of the Ribbon displays an empty Quick Access Toolbar.

Various program commands are displayed as large or small buttons. Large buttons denote frequently used commands, while small buttons show specific features of the main commands. For example, in the Forms group there is a large Form button and a small Form Wizard button. Some large and small command buttons include drop-down lists of other specialized commands. For example, the small More Forms button drop-down contains additional methods for creating a form: Multiple Items, Datasheet, Split Form, and Modal Dialog (Figure 23.13).



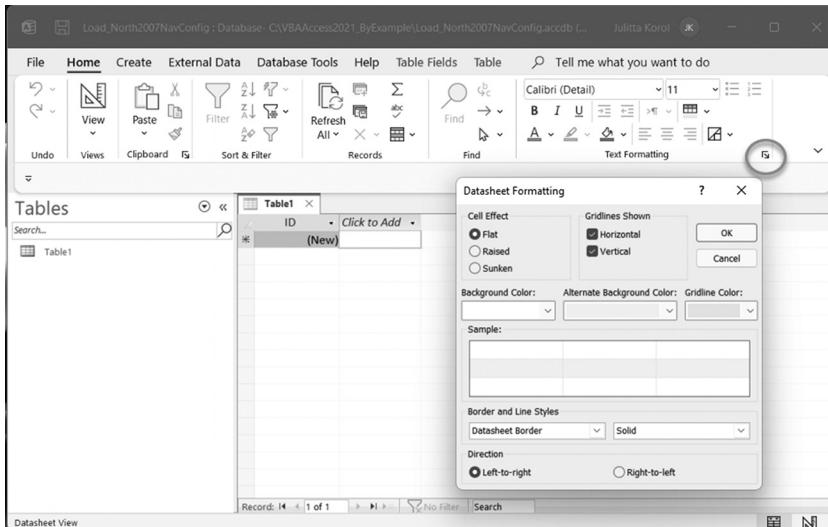
**FIGURE 23.13** Additional commands can be accessed by clicking on the down arrow to the right of the button control.

Some controls that you find on the Ribbon may be disabled until certain other options are chosen because these selected options do not display commands. Instead, they provide a visual clue of the output you might expect when a specific option is selected. These types of controls are known as *galleries*. Gallery controls are often used to present various formatting options, such as the margin settings shown in Figure 23.14.



**FIGURE 23.14** Clicking on the Application Parts button in the Template group of the Create tab displays a gallery of different types of blank form layouts.

Some tab groups have dialog box launchers in the bottom-right corner (see Figure 23.15) that display a dialog box in which you can set several advanced options at once.



**FIGURE 23.15** The dialog box launcher button in the bottom-right corner of the Text Formatting group on the Home tab will display the Datasheet Formatting dialog box.

In addition to the main Ribbon tabs, there are also contextual tabs that contain commands that apply to what you are doing. When a particular object is selected, the Ribbon displays a contextual tab that provides commands for working with that object. For example, when a table is open in Datasheet view, the Ribbon displays a contextual tab called Table Fields. Clicking on the Table Fields tab shows Ribbon options pertaining to this choice (see Figure 23.16). The contextual tab disappears when you cancel the selection of the object. In other words, close the datasheet and the Table Fields tab will be gone.

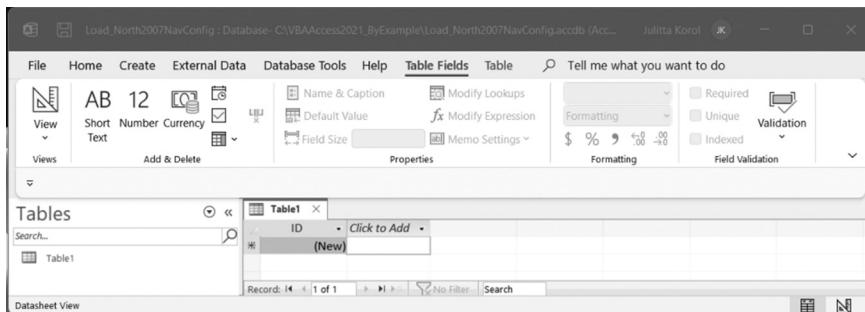


FIGURE 23.16 A contextual tab (Table Tools) in the Ribbon.

Now that you've reviewed the main features of the Ribbon interface, let's look at how you can extend it with your own tabs and controls. The next section introduces you to Ribbon programming.

## RIBBON PROGRAMMING WITH XML, VBA, AND MACROS

---

The components of the Ribbon user interface can be manipulated programmatically using Extensible Markup Language (XML) or other programming languages. Refer to Chapter 27 ("XML Features in Access 2021") for an introduction to using XML with Access.

All Microsoft 365 and Access 2021 standalone products use the Ribbon and rely on the programming model known as Ribbon extensibility, or *RibbonX*.

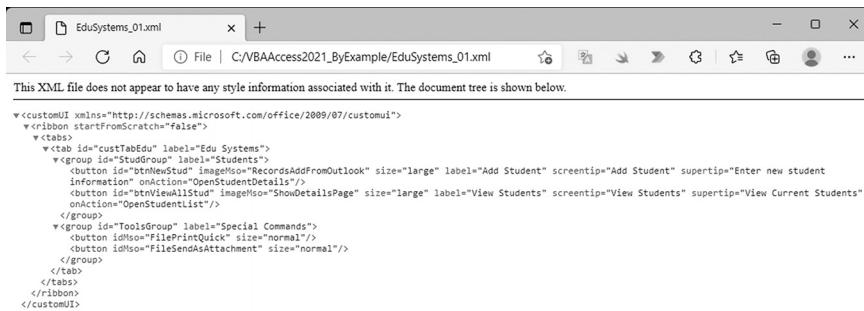
This section introduces you to customizing Access 2021 Ribbons by using XML markup. No special tools are required to perform these customizations. XML is plain text; therefore, you can use any text editor to create your customization files. In the examples that follow, we'll be using the simple Windows Notepad.

Your customizations can be stored in a special Access table, in a VBA procedure, or in another Access database, or they can be linked to an Excel spreadsheet. When storing your customizations in a location other than the Access table, you must call the `LoadCustomUI` method of the `Application` object to load your XML markup manually and then set the Ribbon name in your program at runtime. Ribbon customizations can be applied to the entire application or to specific forms and reports.

<b>NOTE</b>	<p><i>Simple text editors such as Notepad do not provide tools for validating your XML markup. You must be extra careful to write well-formed XML or your code will fail (see Chapter 27 for an introduction to XML terms and markup). If you are planning on performing extensive Ribbon customizations, I recommend that you consider acquiring a dedicated XML editor.</i></p> <p><i>For example, you can try the XML editor provided with the free Community Edition of Microsoft Visual Studio. You can read more about this editor in the following link:</i></p> <p><i><a href="https://docs.microsoft.com/en-us/visualstudio/xml-tools/xml-editor?view=vs-2022">https://docs.microsoft.com/en-us/visualstudio/xml-tools/xml-editor?view=vs-2022</a></i></p> <p><i>Another text editor that can come in handy is the free Notepad++. You can download it from this link:</i></p> <p><i><a href="https://notepad-plus-plus.org">https://notepad-plus-plus.org</a></i></p> <p><i>Once installed, use the Plugins menu to access the Plugin Admin, where you can search for and install XML Tools. The XML Tools plugin will provide you with extensive menu options for working with and validating XML documents.</i></p> <p><i>None of these special tools are necessary for the completion of this chapter's Ribbon customizations. Each new tool, especially an advanced one, requires that you first familiarize yourself with its interface, which can delay your progress if you are in a hurry. To get started with XML programming without further delays, the built-in Windows Notepad will do.</i></p>
-------------	--

### **Creating the Ribbon Customization XML Markup**

To make custom changes to the Ribbon user interface in Access 2021, you need to prepare an XML markup file that specifies all your customizations. The XML markup file that we will use in the Hands-On exercise in this section is shown in Figure 23.17. You can see the resulting output in Figure 23.18.

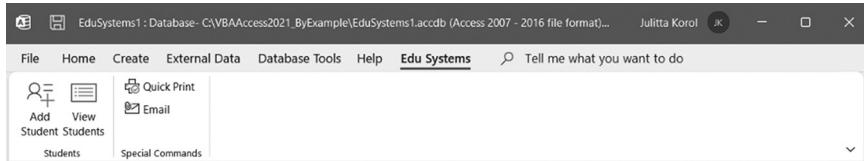


```

<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="false">
    <tabs>
      <tab id="custTabEdu" label="Edu Systems">
        <group id="StudGroup" label="Students">
          <button id="btnNewStud" imageMso="RecordsAddFromOutlook" size="large" label="Add Student" screentip="Add Student" supertip="Enter new student information" onAction="OpenStudentDetails" />
          <button id="btnViewAllStud" imageMso="ShowDetailsPage" size="large" label="View Students" screentip="View Students" supertip="View Current Students" onAction="OpenStudentList" />
        </group>
        <group id="ToolsGroup" label="Special Commands">
          <button idMso="FilePrintQuick" size="normal" />
          <button idMso="FileSendAsAttachment" size="normal" />
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>

```

**FIGURE 23.17** This XML file defines a new tab with two groups for the existing Access 2021 Ribbon. See the output this file produces in Figure 23.18.



**FIGURE 23.18** The custom Edu Systems tab is based on the XML markup file shown in Figure 23.17.



## Hands-On 23.5 Creating an XML Document with Ribbon Customizations

1. Open **Windows Notepad** and type the following XML markup, or copy the code from the **EduSystems\_01.txt** in the companion files.

```

<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon startFromScratch="false">
    <tabs>
      <tab id="custTabEdu" label="Edu Systems">
        <group id="StudGroup" label="Students">
          <button id="btnNewStud" imageMso="RecordsAddFromOutlook" size="large" label="Add Student" screentip="Add Student" supertip="Enter new student information" onAction="OpenStudentDetails" />
          <button id="btnViewAllStud" imageMso="ShowDetailsPage" size="large" label="View Students" screentip="View Students" supertip="View Current Students" onAction="OpenStudentList" />
        </group>
        <group id="ToolsGroup" label="Special Commands">
          <button idMso="FilePrintQuick" size="normal" />
          <button idMso="FileSendAsAttachment" size="normal" />
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>

```

```
</group>
</tab>
</tabs>
</ribbon>
</customUI>
```

Let's go over the contents of this file. In Chapter 27, you will learn that every XML document consists of several elements, called *nodes*. In any XML document there must be a root node, or a top-level element. In the Ribbon customization file, the root tag is `<customUI>`. The root's purpose is to specify the current Office RibbonX XML namespace (`xmlns`):

```
<customUI    xmlns="http://schemas.microsoft.com/office/2009/07/
customui">
```

Namespaces are used to uniquely identify elements in the XML documents and avoid name collisions when elements with the same name are combined in the same document. If you were to customize the Ribbon in the Office 2007 instead, you would use the following namespace instead:

```
<customUI    xmlns="http://schemas.microsoft.com/office/2006/01/
customui">
```

The `xmlns` attribute of the `<customUI>` tag holds the name of the default namespace to be used in the Ribbon customization. Notice that the root element encloses all other elements of this XML document: ribbon, tabs, tab, group, and button. Each element consists of a beginning and ending tag. For example, `<customUI>` is the name of the beginning tag and `</customUI>` is the ending tag.

The actual Ribbon definition is contained within the `<ribbon>` tag:

```
<ribbon startFromScratch="false">
[Include xml tags to specify the required ribbon customization]
</ribbon>
```

The `startFromScratch` attribute of the `<ribbon>` tag defines whether you want to replace the built-in Ribbon with your own (true) or add a new tab to the existing Ribbon (false).

---

**SIDE BAR** *Hiding the Elements of the Access User Interface*

Setting `startFromScratch="true"` in the `<ribbon>` tag will hide the default Ribbon as well as the contents of the Quick Access toolbar.

The File tab will be left with only four commands: New, Open, SaveAs, and Close Database.

---

To create a new tab set in the Ribbon, use the `<tabs>` tag. Each tab element is defined with the `<tab>` tag. The label attribute of the tab element specifies the name of your custom tab. The name in the id attribute is used to identify your custom tab:

```
<tabs>
<tab id="custTabEdu" label="Edu Systems">
```

Ribbon tabs contain controls organized in groups. You can define a group for the controls on your tab with the `<group>` tag. The example XML markup file defines the following two groups for the Edu Systems tab:

```
<group id="StudGroup" label="Students">
<group id="ToolsGroup" label="Special Commands">
```

Like the tab node, the group nodes of the XML document contain the id and label attributes. Placing controls in groups is easy. The group labeled Students has two custom button controls, identified by the `<button>` elements. The group labeled Special Commands also contains two buttons; however, unlike the Students group, the buttons placed here are built-in Office system controls rather than custom controls. You can quickly determine this by looking at the id attribute for the control. Any attribute that ends with “Mso” refers to a built-in Office item:

```
<button idMso="FilePrintQuick" size="normal" />
```

You can download control IDs for built-in controls in all applications that use the Office Fluent user interface (another name for the Ribbon) from the Github repository:

<https://github.com/OfficeDev/office-control-ids>

As mentioned earlier in this chapter, buttons placed on the Ribbon can be large or small. You can define the size of the button with the size attribute set to “large” or “normal.” Buttons can have additional attributes:

```
<button id="btnNewStud" imageMso="RecordsAddFromOutlook"
size="large" label="Add Student"
screentip="Add Student" supertip="Enter new student information"
onAction="OpenStudentDetails" />
```

The `imageMso` attribute denotes the name of the existing Office icon. You can use images provided by any Microsoft 365 or standalone Office application. To provide your own image, you must use the `getImage` attribute in the XML markup (see more information in the section “Using Images in Ribbon Customizations” later in this chapter).

The screentip and supertip attributes allow you to specify the short and longer text that should appear when the mouse pointer is positioned over the button.

The controls that you specify in the XML markup perform their designated actions via *callback procedures*. For example, the `onAction` attribute of a button control contains the name of the callback procedure that is executed when the button is clicked. When that procedure completes, it calls back the Ribbon to provide the status or modify the Ribbon. You will write the callback procedures for the `onAction` attribute in the next section (see Custom Project 23.1).

Buttons borrowed from the Office system do not require the `onAction` attribute. When clicked, these buttons will perform their default built-in action.

Before finishing off the XML Ribbon customization document, always make sure that you have included all the required ending tags:

```
</tab>
</tabs>
</ribbon>
</customUI>
```

2. Save the text file you created in Step 1 as **C:\VBAAccess2021\_ByExample\EduSystems\_01.xml**.

By entering the XML extension, the text file is saved as an XML document.

3. To ensure that this XML document is well formed (it follows the formatting rules for XML), open the created xml file in a browser.

If the browser can read the document, then its output should match Figure 23.17. If the browser finds problems with the document, it will show you the incorrect statement. It is up to you to figure out what correction is required. Open the file in Notepad, correct the erroneous code, save the file, and test it again by loading it in the browser. This is one of the situations where the dedicated XML tools would help you with debugging and validation issues.

4. Close the browser.

At this point, you should have a well-formed XML document with your first Ribbon customizations.

Now that you know how to structure an XML document for Ribbon customizations, you should find it straightforward to add other features to the Access Ribbon as they are discussed in this chapter.

### **Loading Ribbon Customizations from an External XML Document**

Since your first Ribbon customization is already in an external XML document, we will go ahead and load it into Access using the combination of VBA and macros. In a later section of this chapter, you will learn how to load the same

XML markup into a local Access table and have Access take care of the Ribbon modifications at startup.

Custom Project 23.1 walks you through the steps required to integrate Ribbon customizations into your database application.



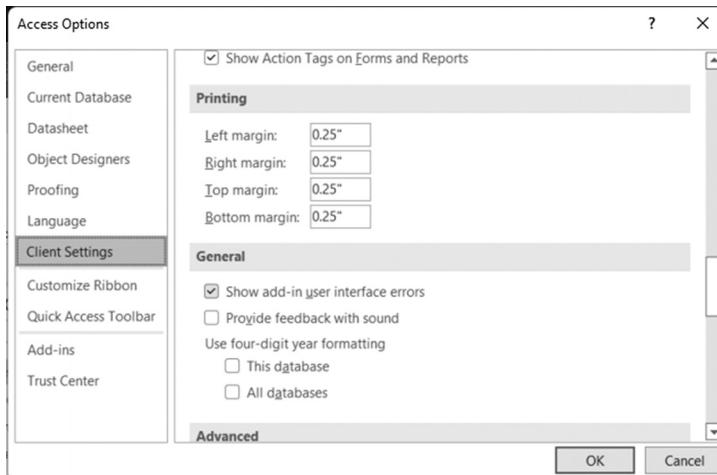
### Custom Project 23.1 Applying Ribbon Customizations from an External XML File

This custom project depends on the XML document prepared in Hands-On 23.5.

#### **Part 1: Setting Access Options**

1. Copy the EduSystems1.accdb database from the companion files to your C:\VBAAccess2021\_ByExample folder. This database is a copy of a database generated by the Students template provided by Microsoft Corporation. It can also be downloaded by opening Access and searching for templates. We will use this database to build Ribbon customizations in this chapter.
2. Open the EduSystems1 database and click **File | Options**, then **Client Settings**.
3. In the General section, make sure that the **Show add-in user interface errors** option is selected (Figure 23.19).

When you enable this option, you will be able to see error messages if errors are encountered when you load your Ribbon customizations.



**FIGURE 23.19** When you check Show add-in user interface errors, Access will notify you about any problems in the Ribbon XML.

4. Click **OK** to close the Access Options dialog box.

## Part 2: Setting Up the Programming Environment

5. Choose Database Tools | Visual Basic to switch to the Visual Basic Editor window.
6. Choose Tools | References. In the References dialog box, add references to the following two libraries: **Microsoft Office 16.0 Object Library** and **Microsoft Scripting Runtime** (Figure 23.20). Scroll down the list until you find and select these libraries.

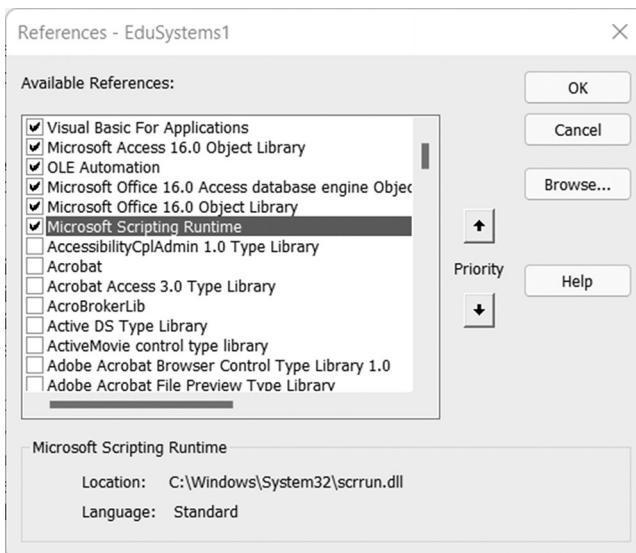


FIGURE 23.20 To avoid compile errors, you must set library references as shown here.

7. Click **OK** to close the **References** dialog box.
8. Choose **Insert | Module**.
9. In the Properties window, change the **Name** property of the module to **RibbonModification**.

## Part 3: Writing VBA Code

10. In the RibbonModification Code window, enter the following VBA procedures, or copy and paste the VBA code from **RibbonVBA.txt** in the companion files:

```
Sub OpenStudentDetails(ByVal control As IRibbonControl)
    DoCmd.OpenForm "Student Details", acNormal, , , acFormAdd
End Sub
```

```
Sub OpenStudentList(ByVal control As IRibbonControl)
    DoCmd.OpenForm "Student List", acNormal
End Sub
```

Notice that both preceding procedures open the specified Access forms. In addition, the OpenStudentDetails procedure opens the Student Details form in Add mode. You may recall that these procedures (OpenStudentDetails and OpenStudentList) are the names of the callback procedures that were specified in the onAction attribute of the button XML:

```
<button id="btnNewStud" imageMso="RecordsAddFromOutlook"
    size="large" label="Add Student"
    screentip="Add Student" supertip="Enter new student information"
    onAction="OpenStudentDetails" />
<button id="btnViewAllStud" imageMso="ShowDetailsPage"
    size="large" label="View Students"
    screentip="View Students"
    supertip="View Current Students"
    onAction="OpenStudentList" />
```

A callback procedure executes some action and then notifies the Ribbon that the task has been completed. The onAction callback can be handled by a VBA procedure, a macro, or an expression. When using VBA, the callback must include the IRibbonControl parameter and return type, as shown here:

```
Sub OpenStudentDetails(ByVal control As IRibbonControl)
Sub OpenStudentList(ByVal control As IRibbonControl)
```

The IRibbonControl parameter is the control that was clicked. This control is passed to your VBA code by the Ribbon. For VBA to recognize this parameter, we added a reference to the Microsoft Office 16.0 Object Library in Part 2 of this custom project.

---

**SIDE BAR** *The IRibbonControl Properties*

You can view the properties (Context, Id, and Tag) of the IRibbonControl object in the Object Browser. The Context property returns the active window that contains the Ribbon interface. The Id property contains the ID of the control that was clicked. The Tag property can be used to store additional information with the control. To use this property, you need to add the tag attribute to the XML markup. You can write a more generic procedure to handle the callbacks by using the Tag property. For example, instead of writing a separate procedure

to open the Student Details and Student List forms as we did in Custom Project 23.1, you could write a single procedure like this:

```
Sub OpenFrm(ByVal control AS IRibbonControl)
    Select Case control.Id
        Case "btnNewStud"
            DoCmd.OpenForm "Student Details", acNormal, , , acFormAdd
        Case "btnViewAllStud"
            DoCmd.OpenForm "Student List", acNormal
    End Select
End Sub
```

Next, you would need to add the tag attribute to the XML markup and change the onAction callback to the OpenFrm procedure name:

```
<button id="btnNewStud" imageMso="RecordsAddFromOutlook"
    size="large" label="Add Student"
    screentip="Add Student" supertip="Enter new student information"
    onAction="OpenFrm" tag="Student Details" />
<button id="btnViewAllStud" imageMso="ShowDetailsPage"
    size="large" label="View Students"
    screentip="View Students"
    supertip="View Current Students"
    onAction="OpenFrm" tag="Student List" />
```

You can see the implementation of the preceding technique in the EduSystems2.accdb database and EduSystems\_02.xml document located in the companion files.

- 
- 11.** In the RibbonModification Code module, enter the following **LoadRibbon** function procedure, or copy and paste the procedure code from the **RibbonVBA.txt**:

```
Public Function LoadRibbon()
    Dim strXML As String
    Dim oFso As New FileSystemObject
    Dim oTStream As TextStream

    ' Open the file containing the Ribbon customizations
    ' and return a TextStream object that will be used
    ' for reading from the file
    Set oTStream = oFso.OpenTextFile _
        ("C:\VBAAccess2021_ByExample" & _
        "\EduSystems_01.XML", ForReading)

    ' Read the entire stream into a string variable
```

```
strXML = oTStream.ReadAll

' Close the TextStream object
oTStream.Close

' Free up resources
Set oTStream = Nothing
Set oFso = Nothing

' load XML markup that represents a customized Ribbon
Application.LoadCustomUI "EduTabR", strXML

End Function
```

This procedure uses the `LoadCustomUI` method of the `Application` object to load into Access the XML markup that contains your Ribbon customizations. To use this method, you must pass the name of the Ribbon and the XML code that defines the customized Ribbon. In this example, `EduTabR` is the name of our customized Ribbon. You can name your Ribbon anything you want. The `strXML` variable contains the XML markup. Because the XML markup must be passed as a text string, the procedure begins by accessing the `FileSystemObject` from the Microsoft Scripting Runtime (see Part 2) and reading the contents of the XML file using the `ReadAll` method of the `TextStream` object.

In Part 4 of this custom project, you will call the `LoadRibbon` function from the `AutoExec` macro to make the custom Ribbon available to the database application on startup.

12. Click the **Save** button to save changes in the `RibbonModification` module.
13. Choose **Debug | Compile EduSystems1** to ensure that the VBA code does not contain spelling or other errors. If errors are found, correct them before proceeding to the next step.
14. Press **Alt+Q** to close the Visual Basic Editor window and return to Microsoft Access.

#### ***Part 4: Calling the LoadRibbon Function from an Autoexec Macro***

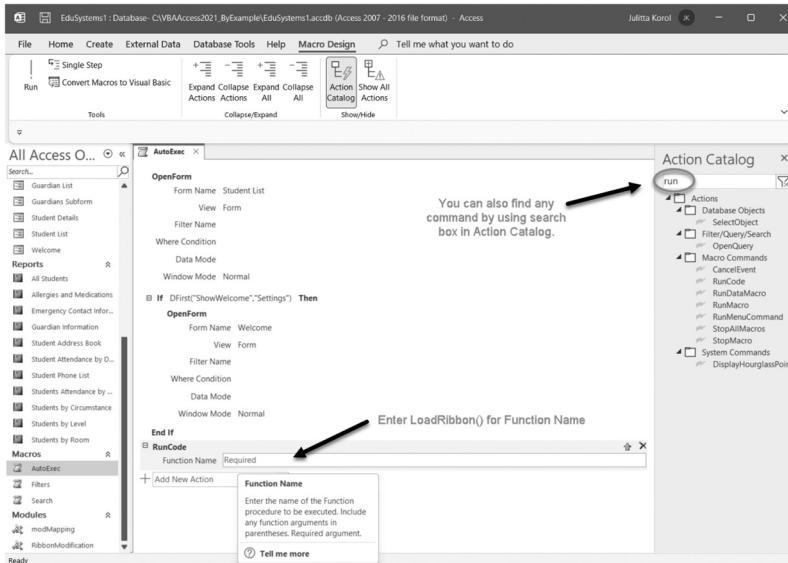
---

15. In the Access window's Navigation pane, right-click the `AutoExec` macro name and select **Design View**.
16. In the macro design window, select **RunCode** from the Add New Action drop-down list. Enter **LoadRibbon()** in the Function Name text box (see Figure 23.21). The function name should automatically appear when you start typing its name.

**17.** Press **Ctrl+S** or click the **Save** icon in the title bar.

An Access macro named AutoExec runs automatically each time you open the database. If you need to open a database without running this macro or to bypass other startup options, remember to hold down the Shift key when opening the database. For more information about using macros in Access 2021, refer to Chapter 26 (“Macros and Templates.”).

- 18.** To run the AutoExec macro right now, click the **Run** button on the Design tab. If you clicked the Run button and received no error, the macro has run successfully, and your Ribbon customization (EduTabR) has been loaded. For the changes in the Ribbon to become visible, you must complete the steps in Part 5 of this custom project.
- 19.** Close the AutoExec Macro Designer window.



**FIGURE 23.21** To load a Ribbon customization in your Access database, enter the custom LoadRibbon() function in the AutoExec macro.

### Part 5: Applying the Customized Ribbon

- 20.** Click the **File** tab, then click **Options**.
- 21.** Click the **Current Database** option. In the **Ribbon and Toolbar Options** section, choose **EduTabR** from the **Ribbon Name** list (Figure 23.22).

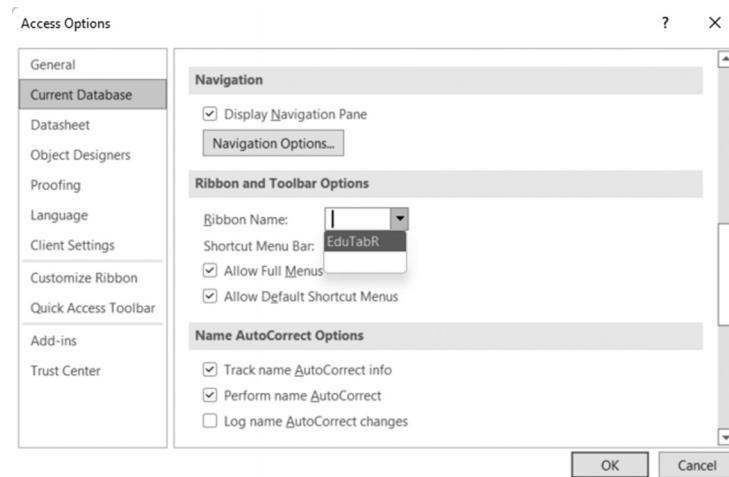


FIGURE 23.22 Enabling a customized Ribbon in the current database.

22. Click **OK** to close the Access Options window.

Microsoft Access displays a message informing you that you must close and reopen the current database for the specified option to take effect.

23. Click **OK** to the message. Then close and restart the **EduSystems1** database. When the database reopens, you should see in the default database Ribbon your custom tab named Edu Systems (see Figure 23.18 earlier in this chapter). Before going on to the next section, allow some time for testing the controls placed in this custom tab.
24. Close the **EduSystems1.accdb** database.

### **Embedding Ribbon XML Markup in a VBA Procedure**

In Custom Project 23.1, you saw how to load a Ribbon customization from an external XML document. Because the name and path of this document are hard-coded in the LoadRibbon function, prior to loading the database you must make sure that the XML markup file exists in the specified folder or Access will greet you with one or more error messages. If you don't want to worry about the location of the XML markup file, you can place the XML markup directly inside the VBA function procedure that loads the Ribbon, as shown in Figure 23.23. While the formatting of the XML string is more time-consuming than referencing the file directly, it will ensure that your Ribbon markup travels with the database. Placing Ribbon XML markup inside a VBA procedure is not recommended if you plan on using the same Ribbon customizations in more than one database. If the Ribbon needs to be modified, you would need to make changes in several places, which can become very confusing.

```

Microsoft Visual Basic for Applications - EduSystems3 - [RibbonModification (Code)]
File Edit View Insert Debug Run Tools Add-Ins Window Help
Project - Desktop Student
Project Explorer
  Desktop Student (EduSystems3)
    Microsoft Access Class Objects
      Form_Guardian List
      Form_Student List
    Modules
      modMapping
      RibbonModification
Properties - RibbonModification
  RibbonModification Module
  Alphabetic Categorized
  (Name) RibbonModification

Option Explicit

Sub OpenStudentDetails(ByVal control As IRibbonControl)
  DoCmd.OpenForm "Student Details", acNormal, , , acFormAdd
End Sub

Sub OpenStudentList(ByVal control As IRibbonControl)
  DoCmd.OpenForm "Student List", acNormal
End Sub

Public Function LoadRibbon()
  Dim strXML As String

  strXML = "<customUI xmlns='http://schemas.microsoft.com/office/2009/07/customui'>" & _
    "<ribbon startFromScratch='false'>" & _
    "<tabs>" & _
    "<tab id='custTabEdu' label='Edu Systems'>" & _
    "<group id='custGroup1' label='Students'>" & _
    "<button id='btnNewStud' imageMso='RecordsAddFromOutlook' size='large' label='Add Student'" & _
    "superTip='Enter new student information' onAction='OpenStudentDetails' />" & _
    "<button id='btnViewAllStud' imageMso='ShowDetailsPage' size='large' label='View Students'" & _
    "superTip='View Current Students' />" & _
    "<button id='btnOpenStudList' imageMso='OpenStudentList' size='large' label='Open Student List'" & _
    "superTip='Open Student List' />" & _
    "</group>" & _
    "<group id='ToolsGroup' label='Special Commands'>" & _
    "<button idMso='FilePrintQuick' size='normal' />" & _
    "<button idMso='FileSendAsAttachment' size='normal' />" & _
    "</group>" & _
  "</tab>" & _
  "</tabs>" & _
  "</ribbon>" & -
  "</customUI>"

  ' load XML markup that represents a customized Ribbon
  Application.LoadCustomUI "EduTabR3", strXML
End Function

```

FIGURE 23.23 Ribbon XML markup can be embedded inside the VBA procedure. To try this out, copy the EduSystems3.accdb database from the companion files to your VBAAccess2021\_ByExample folder and then open the database. Opening the database directly from CD or another folder will generate an error.

## Storing Ribbon Customization XML Markup in a Table

If you store your Ribbon customization XML markup in a local database table, your XML code will be loaded automatically at startup and you won't need to write a special VBA function to load your markup as you did in Custom Project 23.1. To store your XML in a table, you must create a system table named USysRibbons. This table must include two fields: a text field named RibbonName and a memo field named RibbonXML. Access expects these specific column names and data types to read your Ribbon customizations. Any additional fields in this table will be ignored. In the RibbonName field, enter a unique name that identifies your custom Ribbon. The RibbonXML field must contain the XML customization markup to be applied to the Ribbon. The USysRibbons table is a hidden system table. To show this table in the Navigation pane, you must tell Access to show system objects (see the next side bar). You can define multiple Ribbons in your database application by adding a new record to the USysRibbons table.

Let's now proceed to Hands-On 23.6 in which you create the USysRibbons table to store the Ribbon customization prepared earlier in this chapter.



### Hands-On 23.6 Creating a Local System Table to Store Ribbon Customization

This hands-on exercise requires the XML document prepared in Hands-On 23.5.

1. Copy the Access database named **EduSystems\_Local.accdb** from the companion files to your **C:\VBAAccess2021\_ByExample** folder.
2. Open the EduSystems\_Local.accdb database and click **Create | Table Design**.
3. In Table Design view, enter the table structure as shown in Figure 23.24. To make the RibbonName field the primary key, select this field and click the **Primary Key** button in the Tools group of the Table Design tab.
4. Save the table as **USysRibbons**. (Press **Ctrl+S** or click the **Save** button to open the Save As dialog box.)

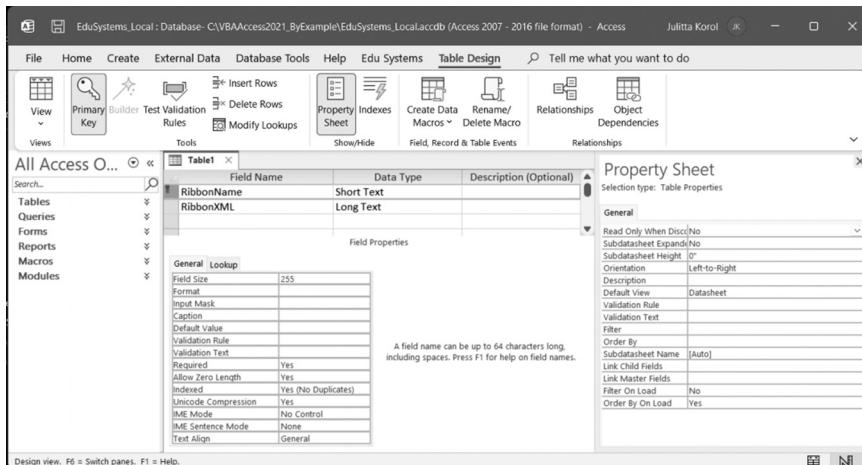


FIGURE 23.24 USysRibbons is a special system table used for storing Ribbon customizations.

5. Open the **C:\VBAAccess2021\_ByExample\EduSystems\_01.xml** in Windows Notepad. Press **Ctrl+A** to select all text and **Ctrl+C** to copy it to the clipboard, then close Notepad.
6. Back in main Access window, in the Table Design tab, click the **View** button and open the table in the Datasheet view. If you closed this table after you saved it, read the side-bar on how to enable system objects in Navigation pane.

7. In the USysRibbons datasheet, in the **RibbonName** field, enter **TestRibbonTab**. Press **Ctrl+V** to paste the entire contents of the **C:\VBAAccess2021\_ByExample\EduSystems\_01.xml** document into the **RibbonXML** field. Expand the row and column width so that the entire XML markup is visible. Make changes in the **onAction** attribute of the buttons as shown in Figure 23.25. In the **onAction** attribute for **btnNewStud**, enter **RibbonLib.OpenStudentDetails**. In the **onAction** attribute for **btnViewAllStud**, enter **RibbonLib.OpenStudentList**.

RibbonName	RibbonXML	Click to Add
TestRibbonTab	<pre>&lt;customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui"&gt; &lt;ribbon startFromScratch="false"&gt; &lt;tabs&gt; &lt;tab id="custTabEdu" label="Edu Systems"&gt; &lt;group id="StudGroup" label="Students"&gt; &lt;button id="btnNewStud" imageMso="RecordsAddFromOutlook" size="large" label="Add Student" screentip="Add Student" supertip="Enter new student information" onAction="RibbonLib.OpenStudentDetails" /&gt; &lt;button id="btnViewAllStud" imageMso="ShowDetailsPage" size="large" label="View Students" screentip="View Students" supertip="View Current Students" onAction="RibbonLib.OpenStudentList" /&gt; &lt;/group&gt; &lt;group id="ToolsGroup" label="Special Commands"&gt; &lt;button idMso="FilePrintQuick" size="normal" /&gt; &lt;button idMso="FileSendAsAttachment" size="normal" /&gt; &lt;/group&gt; &lt;/tab&gt; &lt;/tabs&gt; &lt;/ribbon&gt; &lt;/customUI&gt;</pre>	

FIGURE 23.25 The USysRibbons table with a record defining Ribbon customization. To define multiple Ribbons in your application, simply add a new record to this table.

8. Save and close the **USysRibbons** table.

#### SIDE BAR *Showing System Objects in the Navigation Pane*

By default, system tables do not show in the Navigation pane. If you need to open the USysRibbons table to correct any errors or add a new record, you must enable the system objects in the Navigation Options dialog box, as follows:

1. Click the **File** tab, then click **Options**.
2. Click **Current Database**, then in the Navigation section click the **Navigation Options** button.
3. Select **Show System Objects** and click **OK** to close the Navigation Options dialog box.
4. Click **OK** to exit the Access Options dialog box. You may be asked to restart the database.

---

The next step is to enter callbacks that are needed for the button actions. In Custom Project 23.1 you wrote VBA callback procedures for the btnNewStud and btnViewAllStud buttons. Instead of a VBA callback, the **onAction** attribute of the button control can invoke a macro. Macro callbacks do not require that you return a value to the Ribbon. Also, your Ribbon customization can be functional even in safe mode (when code is not enabled for the database). It is up to you to decide whether to write VBA callbacks or to create simple macro actions for your custom Ribbon controls.

Hands-On 23.7 demonstrates the implementation of macros in the **onAction** attribute for controls. This hands-on exercise also introduces you to submacros. *Submacros* are like subroutines. Instead of cluttering the Navigation pane with many small macros that perform a specific task, you can define a series of actions in one place as a submacro and then call that submacro whenever it's needed.



### Hands-On 23.7 Using Macros Instead of VBA Callbacks

1. In the database window, click the **Create** tab, then in the Macros & Code group, click the **Macro** button.
  2. In the macro design window, select **Submacro** from the Add New Action drop-down list. This option appears towards the top of the drop-down list.
  3. In the Submacro name text box, enter **OpenStudentDetails**.
  4. Specify the form settings as shown in Figure 23.26:
    - a. Select **OpenForm** from the Add New Action drop-down list.
    - b. Select **StudentDetails** from the Form Name drop-down list.
    - c. Select **Add** from the Data Mode drop-down list.
- You have now completed the first submacro.
5. Select **Submacro** from the Add New Action drop-down list located below **End Submacro**.
  6. In the Submacro name text box, enter **OpenStudentList**.

7. Specify the form settings as shown in Figure 23.27:
  - a. Select **OpenForm** from the Add New Action drop-down list.
  - b. Select **StudentList** from the Form Name drop-down list.

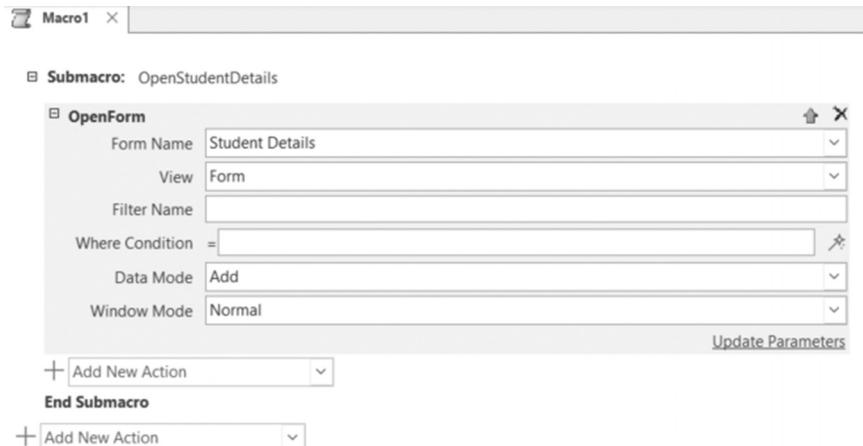


FIGURE 23.26 Creating the OpenStudentDetails submacro.

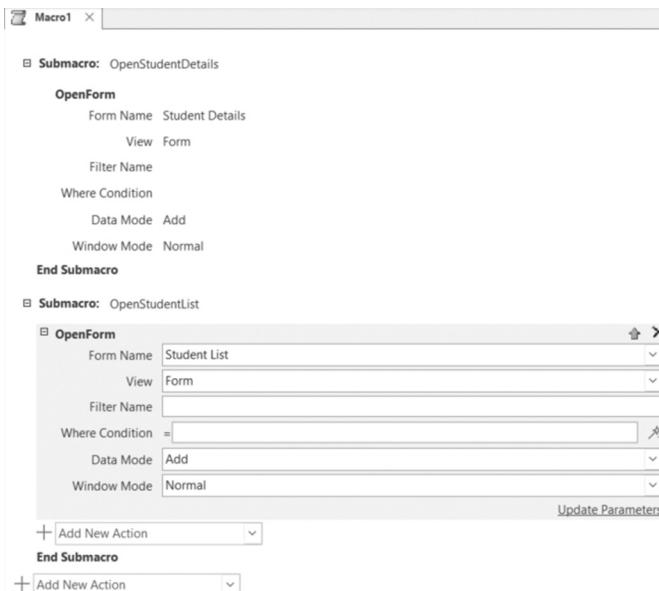


FIGURE 23.27 Creating the OpenStudentList submacro.

8. Press **Ctrl+S** to invoke a Save As dialog box. Enter **RibbonLib** for your macro name. This macro contains the two submacros created in earlier steps.

9. Close the RibbonLib Macro Designer window.

Now that your macro callbacks are ready, you must tell Access to read your Ribbon definition from the USysRibbons system table. To do this, you must close and restart the database.

10. Restart Access and then reload the **EduSystems\_Local.accdb** database.

When the application starts, Access looks for the USysRibbons system table. If the table exists, Access proceeds to read the data. If any errors are encountered in the Ribbon definition and you have set the option to Show add-in user interface errors (see Figure 23.19 earlier), you will see error messages like the one shown in Figure 23.28. You must correct all the errors before Access can display your customization in the Ribbon.

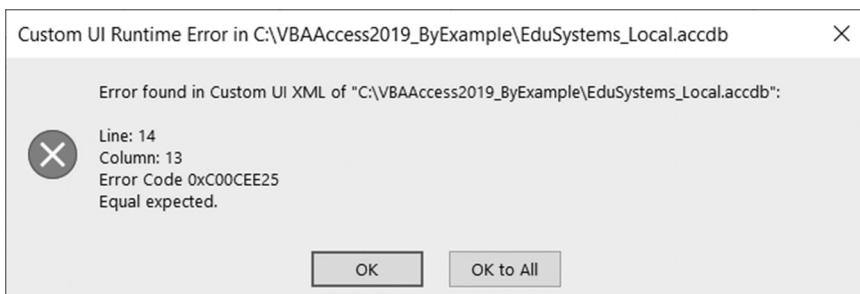


FIGURE 23.28 Upon loading the database, Access displays an error message if errors are found in the Ribbon customization markup.

If there are no errors, Access loads your customization; however, before you can see the Ribbon you need to tell Access to apply your Ribbon customization when the application is started.

11. Click the **File** tab, then click **Options**.

12. Click **Current Database**. In the Ribbon and Toolbar Options section, choose the name of your customized Ribbon from the Ribbon Name list: **Test.RibbonTab**.

13. Click **OK** to close the Access Options window.

Access will advise you that you must close and restart the application before the changes take effect.

14. Close and restart the database.

When the EduSystems\_Local database is reloaded, it should see your custom Ribbon tab named Edu Systems. Take the time to test the controls placed on this tab to make sure that the macro actions are invoked correctly.

**15.** Close the **EduSystems\_Local.accdb** database.

**NOTE**

*If you don't want Access to automatically load Ribbon customizations from the USysRibbons table, simply rename this table.*

### **Assigning Ribbon Customizations to Forms and Reports**

In addition to customizing the main database Ribbon, Access allows you to create Ribbons that are associated with a particular form or report. To display Ribbon content for forms and reports, you can use a contextual tabset called `AccessFormReportExtensibility`. This tabset is hidden by default; however, it will become visible when it has controls to display. You will insert some commands into this contextual tabset in Custom Project 23.2. Because the contextual tabset takes focus when the form or report is first opened, your users will be able to see right away the special controls you've made available for them. These controls can include built-in icons from other Access tabs or your own custom buttons and other types of controls as discussed later in this chapter.

Keep in mind that Ribbon customizations for forms and reports are only displayed when a form or report is loaded or activated, and they are removed when the object is closed or deactivated. While a specific form or report is in use, you may also hide other built-in Ribbon items. You can do this by setting the `visible` attribute of a Ribbon item to `False`. This will prevent users from using features of the program that you don't want to be available.

To assign a custom Ribbon to a form or report, you must open a form or report in Design or Layout view. On the Other tab of the property sheet, choose the Ribbon you want to apply from the Ribbon Name list.



### **Custom Project 23.2 Creating and Assigning Ribbon Customization to a Report**

This custom project requires access to the `EduSystems_Local.accdb` database and the `USysRibbons` table that was created in Hands-On 23.6.

#### ***Part 1: Creating Ribbon Customization for a Report Using a Local System Table***

1. Open the `C:\VBAAccess2021_ByExample\EduSystems_Local.accdb` database.

This database will display a custom tab named Edu Systems. Recall that the XML markup for this customization is stored in the local system table named `USysRibbons`. In this exercise, you will add another record to this table to specify a Ribbon customization for an Access report. Before you proceed to the

next step, make sure that the USysRibbons table is displayed in the Navigation pane. To unhide the table, follow the steps outlined in the previous sidebar, “Showing System Objects in the Navigation Pane.”

2. Open the **USysRibbons** table and enter a new record for the Ribbon named **AlergMedRpt** as shown in Figure 23.29. You can copy the XML markup file from EduSystems\_04.xml in the companion files.

RibbonName	RibbonXML	Click to Add
AlergMedRpt	<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui"> <ribbon startFromScratch="false"> <contextualTabs> <tabSet idMso="TabSetFormReportExtensibility"> <tab id=" rptTools" label="Report Tools"> <group idMso="GroupSortAndFilter" /> <group idMso="GroupFindAccess" /> </tab> </tabSet> </contextualTabs> <tabs> <tab idMso="TabExternalData" visible="true"> <group idMso="GroupCollectData" visible="false" /> <group idMso="GroupSharepointLists" visible="false" /> </tab> </tabs> </ribbon> </customUI>	
TestRibbonTab	<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui"> <ribbon startFromScratch="false"> <tabs>	

FIGURE 23.29 Entering Ribbon customization for a report into the new record of the USysRibbons table.

As mentioned earlier, the RibbonXML field contains the XML markup you want to apply to a report. The RibbonName can be any name you want to use to identify this customization. To have Access use the special contextual tabset available for forms and reports, you must use the `<contextualTabs>` XML tag. Within this tag, use the `<tabSet>` tag. Because this tabset is defined by Access, you must specify `TabSetFormReportExtensibility` in the `idMso` attribute:

```
<contextualTabs>
<tabSet idMso="TabSetFormReportExtensibility">
In the next statement, assign a custom ID and a name to the tab
that will contain your customization:
<tab id="rptTools" label="Report Tools">
The preceding XML statement tells Access to place the focus on
the Report Tools tab when the report is opened.
The next two XML statements define the controls you want to
display:
<group idMso="GroupSortAndFilter" />
<group idMso="GroupFindAccess" />
</tab>
```

In this example, you are telling Access to simply add the Sort and Filter and Find groups from its library of built-in controls. As mentioned earlier, you can download the list of control IDs from the GitHub repository. Since currently you are not defining other customizations to appear on this tab, you need to close this XML group by including the closing tags:

```
</tabSet>
</contextualTabs>
```

When the report is loaded, you also want to disable certain built-in features such as controls that collect data and use SharePoint lists. This can be done by setting the visible attribute of the named built-in control groups to `false`:

```
<tabs>
<tab idMso="TabExternalData" visible="true">
<group idMso="GroupCollectData" visible="false" />
<group idMso="GroupSharepointLists" visible="false" />
</tab>
</tabs>
```

To finish off the customization markup, you need to include the ending tags:

```
</ribbon>
</customUI>
```

3. Press **Ctrl+S** to save changes to the **USysRibbons** table.
4. Close the **USysRibbons** table.

## ***Part 2: Making Access Aware of the New Customization***

---

Remember that the Ribbon customization cannot be displayed until you close and reopen the database.

5. Exit Access and reopen the **EduSystems\_Local.accdb** database.

When Access loads, it will read the Ribbon customizations from the USysRibbons table. Now is the time to tell Access to load the customized Ribbon for a specific report.

**NOTE**

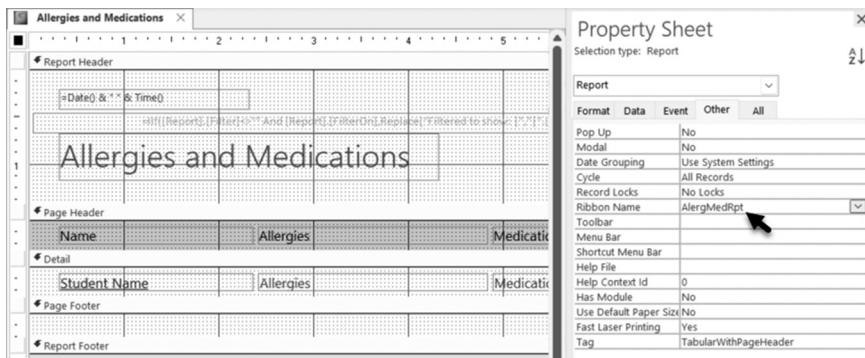
*You should follow the same steps for creating and assigning Ribbon customizations for a form. Of course, your XML markup for a form ought to include the features related to forms and not reports.*

---

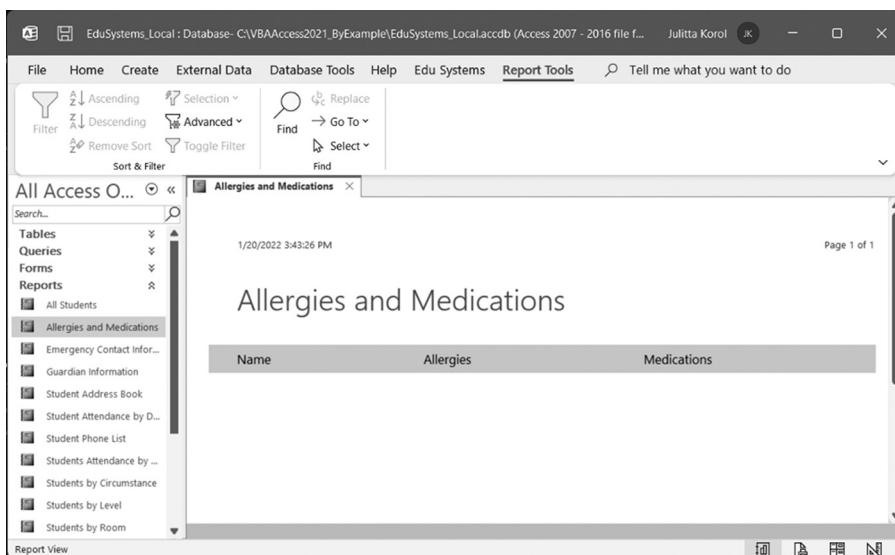
***Part 3: Assigning a Ribbon Customization to a Report***

---

6. In the Navigation pane, right-click the **Allergies and Medications** report and choose **Design View**.
7. If the property sheet is not displayed, press **Alt+Enter** to display it or click the **Property Sheet** button in the Ribbon. Make sure **Report** is selected in the selection list at the top of the property sheet.
8. In the property sheet, click the **Other** tab, click the down arrow next to the **Ribbon Name** property, and choose **AlergMedRpt** from the drop-down list (see Figure 23.30).
9. Press **Ctrl+S** to save the changes.
10. Close the **Allergies and Medications** report, then reopen it.  
Notice that when the report opens, the focus is on your custom Ribbon tab named **Report Tools** (Figure 23.31).
11. Click the **External Data** tab and notice that only two control groups are shown: Import & Link and Export. The Collect Data tab that normally appears for reports is removed from the Ribbon. This report group is made invisible when the Allergies and Medications report is active, and appears on the External Data tab when any other report is open.
12. Close the **Allergies and Medications** report when you are finished viewing Ribbon customizations.
13. Close the **EduSystems\_Local.accdb** database.



**FIGURE 23.30** Use the Ribbon Name property of the report to assign your Ribbon customization to the active report.



**FIGURE 23.31** The custom Report Tools tab appears in the Access Ribbon when the Allergies and Medications report is opened.

## USING IMAGES IN RIBBON CUSTOMIZATIONS

The images you have learned to use so far in your Ribbon customizations are images provided by any Microsoft 365 application that implements the Ribbon. You already know that to reuse an Office icon you must use the `imageMso` attribute of a control. However, instead of using built-in Office images you can also

use your own BMP, GIF, and JPEG image files. These images can be stored in a directory on your computer or a network drive, or in an Access table, then passed to your Ribbon controls via the loadImage callback for the Ribbon or the getImage callback for a control.

### **Requesting Images via the loadImage Callback**

---

You can specify the name of a custom image file to be loaded for a specific control on the Ribbon by using the image attribute. When you request the image via the image attribute, the loadImage callback is called. To load images dynamically with one procedure call, define the callback procedure name in the loadImage attribute of the customUI node. Here's a fragment of the XML markup file that we'll use in Custom Project 23.3 to implement this method of loading images:

1. In the first line of your Ribbon customization markup (inside the <customUI> tag), use the loadImage attribute and specify the name of the callback procedure:

```
<customUI    xmlns="http://schemas.microsoft.com/office/2009/07/
customui"
loadImage="OnLoadImage">
```

2. When defining your Ribbon controls, use the image attribute and specify the name of the image file:

```
<group id="ImagesGroup" label ="Special Features">
<button id="btnNotes" label="Open Notepad"
image="Note.gif" size="large" onAction="OpenNotepad" />
<button id="btnComputer" label="Computer folder"
image="MyFolder.gif" size="normal" />
</group>
```

3. Write the loadImage callback procedure (OnLoadImage) in a VBA module:

```
Public Sub OnLoadImage(imgName As String, ByRef image)
    Dim strImgFileName As String
    strImgFileName = "C:\VBAAccess2021_ByExample\images\" & imgName

    Set image = LoadPicture(strImgFileName)
End Sub
```

Notice that to load a picture from a file, you need to use the `LoadPicture` function. This function is a member of the `stdole.StdFunctions` library. The library file, which is called `stdole2.tlb`, is installed in the System or System32 folder on your computer and is available to your VBA procedures without setting additional references. The `LoadPicture` function returns an object of type `IPictureDisp` that represents the image. You can view objects, methods,

and properties available in the stdole library by activating the Object Browser in the Visual Basic Editor window.

4. Write the callback procedure for the button labeled OpenNotepad:

```
Public Sub OpenNotepad(ctl As IRibbonControl)
    Shell "Notepad.exe", vbNormalFocus
End Sub
```

The OpenNotepad procedure tells Access to use the `Shell` function to open Windows Notepad. Notice that the name of the program's executable file is in double quotes. The second argument of the `Shell` function is optional. This argument specifies the window style, that is, how the program will appear once it is launched. The `vbNormalFocus` constant will open Notepad in a normal size window with focus. If the window style is not specified, the program will be minimized with focus (`vbMinimizedFocus`).

Let's proceed to Custom Project 23.3, which adds two new buttons with custom images to the Ribbon.



### Custom Project 23.3 Loading Custom Images Using the loadImage Callback

This project requires access to the EduSystems\_Local.accdb database and the USysRibbons table that was created in Hands-On 23.6. To use custom images, copy the **Images** folder from the companion files to your **C:\VBAAccess2021\_ByExample** folder.

#### Part 1: Creating Ribbon Customization for Loading Custom Images

1. Open the **C:\VBAAccess2021\_ByExample\EduSystems\_Local.accdb** database.
2. In the Navigation pane, double-click the **USysRibbons** table to open it. If you cannot find this table, refer to Part 1 in Custom Project 23.2.
3. Enter a new record for the Ribbon named **CustomImage1** as shown in Figure 23.32. You can copy the XML markup from the **EduSystems\_05.txt** file in the companion files.

In the first line of the Ribbon customization markup (inside the `<customUI>` tag), notice that we've added the `loadImage` attribute. This attribute specifies the name of the callback procedure, `OnLoadImage`, that will handle loading the custom images included in the Special Features group. The Special Features group contains two images to be loaded from the **C:\VBAAccess2021\_ByExample\Images** folder. Notice that the names of these images are specified in the `image` attribute of each button control in this group. You do not

need to specify the file path; the OnLoadImage procedure will contain this information. For the button to perform some action, you need to include the onAction attribute with the name of the macro, VBA procedure, or expression to be executed. This example does not define the onAction callback for the button named Computer Folder. To test your skills, you can add your own action for this button when you have completed this project.

4. Press **Ctrl+S** to save changes to the USysRibbons table.
5. Close the **USysRibbons** table.

RibbonName	RibbonXML	Click to Add
CustomImage1	<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" loadImage="OnLoadImage"><ribbon startFromScratch="false"><tabs><tab id="custTabEdu" label="Edu Systems"><group id="StudGroup" label="Students"><button id="btnNewStud" imageMso="RecordsAddFromOutlook" size="large" label="Add Student" screentip="Add Student" supertip="Enter new student information" onAction="RibbonLib.OpenStudentDetails" /><button id="btnViewAllStud" imageMso="ShowDetailsPage" size="large" label="View Students" screentip="View Students" supertip="View Current Students" onAction="RibbonLib.OpenStudentList" /></group><group id="ToolsGroup" label="Special Commands"><button idMso="FilePrintQuick" size="normal" /><button idMso="FileSendAsAttachment" size="normal" /></group><group id="ImagesGroup" label="Special Features"><button id="btnNotes" label="Open Notepad" image="Note.gif" size="large" onAction="OpenNotepad" /><button id="btnComputer" label="Computer Folder" image="MyFolder.gif" size="normal" /></group></tab></tabs></ribbon></customUI>	

FIGURE 23.32 Entering Ribbon customization for loading custom images. Notice that this is the third record in the USysRibbons table.

## ***Part 2: Setting Up the Programming Environment***

---

6. Press **Alt+F11** to switch to the Visual Basic Editor window.
7. Choose **Tools | References**. In the References dialog box, add a reference to the following library: **Microsoft Office 16.0 Object Library**.
8. Click **OK** to close the References dialog box.

### ***Part 3: Writing the VBA Callback Procedures***

---

- 9.** Choose **Insert | Module**.
- 10.** In the module Code window, enter the following VBA procedures:

```
Public Sub OnLoadImage(imgName As String, ByRef image)
    Dim strImgFileName As String
    strImgFileName = "C:\VBAAccess2021_ByExample\images\" & imgName

    Set image = LoadPicture(strImgFileName)
End Sub

Public Sub OpenNotepad(ctl As IRibbonControl)
    Shell "Notepad.exe", vbNormalFocus
End Sub
```

<b>NOTE</b>	<i>For the explanations of these procedures, please refer to the beginning of this section.</i>
-------------	---

- 11.** Press **Ctrl+S** to save changes in the Code window. When asked to name your module, enter any name you want.
- 12.** Choose **File | Close and Return to Microsoft Access**.

### ***Part 4: Making Access Aware of the New Customization***

---

Remember that the Ribbon customization cannot be displayed until you close and reopen the database.

- 13.** Close and reopen the **EduSystems\_Local.accdb** database.  
When Access loads, it will read the Ribbon customizations from the USysRibbons table.
- 14.** Click the **File** tab, then click **Options**.
- 15.** Click the **Current Database** option. In the Ribbon and Toolbar Options section, choose **CustomImage1** from the Ribbon Name list.
- 16.** Click **OK** to close the Access Options window.  
Access displays a message informing you that you must close and reopen the current database for the specified option to take effect.
- 17.** Click **OK** to the message. Then close and restart the **EduSystems\_Local** database.  
When the database reopens, you should see the default database Ribbon with your custom tab named Edu Systems (Figure 23.33).

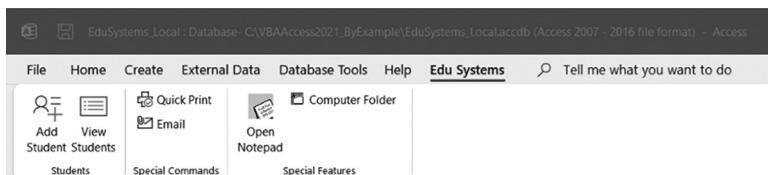


FIGURE 23.33 The Ribbon customization as defined in Custom Project 23.3 with two image buttons in the Special Features group of the Edu Systems tab.

18. Try out one of the buttons by clicking the **Open Notepad** button to open Windows Notepad. Then close Notepad.

Before going on to the next section, take time to modify the Ribbon XML to include the `onAction` callback for the button labeled Computer Folder and write your own custom VBA procedure to execute when this button is clicked. For example, you can make this button display a dialog box asking the user for the name of the folder to create, then use the VBA built-in function `MkDir` to create it. Use the Object Browser to locate this function. Remember that you will have to close and reopen the database for Access to recognize your Ribbon changes.

### Requesting Images via the `getImage` Callback

---

Custom images can also be loaded to the Ribbon using the `getImage` attribute of a control. The procedure you specify in this attribute will retrieve the correct image from the specified location using the same `LoadPicture` function you worked with in the previous section. The following XML markup adds two new controls with custom images to the Special Features group that was defined in Custom Project 23.3:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/
loadImage="OnLoadImage">
  <ribbon startFromScratch="false">
    <tabs>
      <tab id="custTabEdu" label="Edu Systems">
        <group id="StudGroup" label="Students">
          <button id="btnNewStud" imageMso="RecordsAddFromOutlook"
size="large" label="Add Student"
screentip="Add Student" supertip="Enter new student
information"
onAction="RibbonLib.OpenStudentDetails" />
          <button id="btnViewAllStud" imageMso="ShowDetailsPage"
size="large" label="View Students"
screentip="View Students"
supertip="View Current Students"
```

```
onAction="RibbonLib.OpenStudentList" />
</group>
<group id="ToolsGroup" label="Special Commands">
<button idMso="FilePrintQuick" size="normal" />
<button idMso="FileSendAsAttachment" size="normal" />
</group>
<group id="ImagesGroup" label="Special Features">
<button id="btnNotes" label="Open Notepad"
image="Note.gif" size="large"
onAction="OpenNotepad" />
<button id="btnComputer" label="Computer Folder"
image="MyFolder.gif" size="normal" />
<button id="btnRedStar" label="Honor Student"
getImage="OnGetImage" size="large" />
<gallery id="glHolidays" label="Holidays" columns="3" rows="4"
getImage="OnGetImage" getItemCount="OnGetItemCount"
getItemLabel="OnGetItemLabel" getItemImage="OnGetItemImage"
getItemID="onGetItemID" onAction="onSelectedItem" />
</group>
</tab>
</tabs>
</ribbon>
</customUI>
```

In the preceding Ribbon customization markup, we are using all the controls that have been added thus far in this chapter's hands-on exercises and projects. In addition, the Special Features group now includes a new button labeled Honor Student and a gallery control labeled Holidays:

```
<button id="btnRedStar" label="Honor Student"
getImage="OnGetImage" size="large" />
<gallery id="glHolidays" label="Holidays" columns="3" rows="4"
getImage="OnGetImage" getItemCount="OnGetItemCount"
getItemLabel="OnGetItemLabel" getItemImage="OnGetItemImage"
getItemID="onGetItemID" onAction="onSelectedItem" />
```

In this XML markup, the gallery control will perform the action specified in the onSelectedItem callback procedure. To specify your own callback procedure for the Honor Student button, you must add the onAction attribute to this button, then write the appropriate VBA code. Notice that the gallery control has many attributes that contain static text or define callbacks. We will discuss them later. Right now, let's focus on the image-loading process. Both the button and the gallery controls use the getImage attribute with the OnGetImage callback

procedure. This procedure will tell Access to load the appropriate image to the Ribbon for each of these controls:

```
Public Sub OnGetImage(ctl As IRibbonControl, ByRef image)
    Select Case ctl.id
        Case "btnRedStar"
            Set image = LoadPicture("C:\VBAAccess2021_ByExample\images\redstar.gif")
        Case "glHolidays"
            Set image =
LoadPicture("C:\VBAAccess2021_ByExample\images\Square0.gif")
    End Select
End Sub
```

Notice that the decision as to which image should be loaded is based on the ID of the control in the `Select Case` statement. The gallery control also uses the `OnGetItemImage` callback procedure (defined in the `getItemImage` attribute) to load custom images for its drop-down selection list (see Figure 23.34).

Use the `columns` and `rows` attributes to specify the number of columns and rows in the gallery when it is opened. If you need to define the height and width of images in the gallery, use the `itemHeight` and `itemWidth` attributes (not used in this example due to the simplicity of the utilized images). The `getCount` and `getItemLabel` attributes contain callback procedures that provide information to the Ribbon on how many items should appear in the drop-down list and the names of those items. The `getItemImage` attribute contains a callback procedure that specifies the images to be displayed next to each gallery item. The `getItemID` attribute specifies the `onGetItemID` callback procedure that will provide a unique ID for each of the gallery items.

Now that we've discussed the Ribbon customization markup, let's go over the VBA callbacks that are referenced in it. The following procedures need to be added to the VBA module for the preceding XML markup to work:

```
Public Sub OnGetItemCount(ctl As IRibbonControl, ByRef count)
    count = 12
End Sub
```

In this procedure, we use the `count` parameter to return to the Ribbon the number of items we want to have in the gallery control.

```
Public Sub OnGetItemLabel(ctl As IRibbonControl, _
    index As Integer, ByRef label)
    label = MonthName(index + 1)
End Sub
```

This procedure will label each of the gallery items. The VBA `MonthName` function is used to retrieve the name of the month based on the value of the index. The initial value of the index is zero (0). Therefore, `index + 1` will return February. To display the month's name abbreviated (Jan, Feb, etc.), specify `True` as the second parameter to this function:

```
label = MonthName(index + 1, True)
```

If you are using a localized version of Microsoft 365 or standalone Access (French, Spanish, etc.), the `MonthName` function will return the name of the month in the specified interface language.

The next callback procedure shows how to load images for each gallery item:

```
Public Sub OnGetItemImage(ctl As IRibbonControl, _
    index As Integer, ByRef image)

    Dim imgPath As String

    imgPath = "C:\VBAAccess2021_ByExample\images\square"
    Set image = LoadPicture(imgPath & index + 1 & ".gif")
End Sub
```

Each item in the gallery must have a unique ID, so the `onGetItemID` callback uses the `MonthName` function to specify the ID:

```
Public Sub onGetItemID(ctl As IRibbonControl, _
    index As Integer, ByRef id)

    id = MonthName(index + 1)

End Sub
```

The last procedure you need to write for the gallery control should define the actions to be performed when an item in the gallery is clicked. This is done via the following `onSelectedItem` callback that was specified in the `onAction` attribute of the XML markup:

```
Public Sub onSelectedItem(ctl As IRibbonControl, _
    selectedId As String, _
    selectedIndex As Integer)

    Select Case selectedIndex
        Case 6
            MsgBox "Holiday 1: Independence Day, July 4th", _
                vbInformation + vbOKOnly, _
                selectedId & " Holidays"
        Case 11
    End Select
End Sub
```

```

MsgBox "Holiday 2: Christmas Day, December 25th", _
vbInformation + vbOKOnly, _
selectedId & " Holidays"
Case Else
    MsgBox "Please program holidays for " & selectedId & ".",
    -
    vbInformation + vbOKOnly, _
    " Under Construction"
End Select
End Sub

```

In the preceding callback procedure, the `selectedId` parameter returns the name that was assigned to the label, while the `selectedIndex` parameter is the position of the item in the list. The first item in the list (January) is indexed with zero (0), the second with 1, and so forth. In this procedure we have just coded two holidays: one for the month of July (`selectedIndex=6`) and one for December (`selectedIndex=11`). The `Case Else` clause in the `Select Case` statement provides a message when other months are selected.

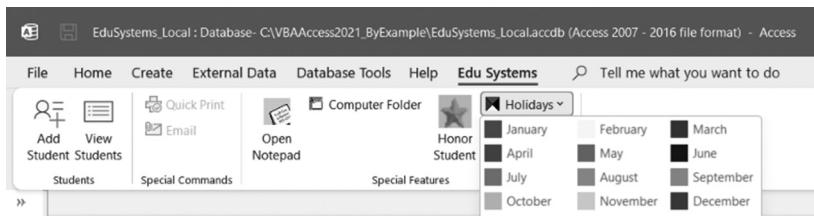


FIGURE 23.34 Customized Ribbon with the gallery control.

To implement the Ribbon customization shown in Figure 23.32, follow the steps outlined in Hands-On 23.8.

### Hands-On 23.8 Loading Custom Images Using the getImage Callback

This hands-on exercise requires access to the `EduSystems_Local.accdb` database and the `USysRibbons` table that was created in Hands-On 23.6. This exercise assumes that you have also completed Custom Project 23.3, which presented a method of loading images via the `loadImage` callback. By now you should be very familiar with the Ribbon customization process, and thus this exercise outlines only the main steps you need to take to complete it.

For a detailed explanation of the process, refer to the previous exercises and projects. The images used in this example are located in the `C:\VBAAccess2021_ByExample\Images` folder.

1. In the **USysRibbons** table of the **EduSystems\_Local** database, add a new record. In the RibbonName field, enter **CustomImage2** for the name of the new Ribbon customization. In the RibbonXML field, paste the XML markup from the **EduSystems\_06.txt** file in the companion files. Press **Ctrl+S** to save the changes, then close the **USysRibbons** table.
2. Press **Alt+F11** to switch to the Visual Basic Editor window. You should see one module with VBA procedures that were added in Custom Project 23.3. You do not need to create a new module for this customization. Simply enter in the existing module Code window the VBA procedures discussed earlier in this section (OnGetImage, OnGetItemCount, OnGetItemLabel, OnGetItemImage, onGetItemID, and onSelectedItem). Press **Ctrl+S** to save the changes in your module and exit Visual Basic Editor. All these procedures can be also copied from the **EduSystems\_HandsOn\_23\_8.txt** file located in the companion files.
3. Close and restart the **EduSystems\_Local** database. When the database is reloaded, click the **File** tab and select **Options**. In the Access Options window, click **Current Database**, and select your new Ribbon (**CustomImage2**) from the Ribbon Name list in the Ribbon and Toolbar Options section. Click **OK** to close the Access Options window. Access will display a message informing you that you must close and reopen the current database for the specified option to take effect. Click **OK** to the message. Then close and restart the **EduSystems\_Local** database.

The customized Ribbon should appear as shown earlier in Figure 23.34. Test the gallery control by clicking on some of the month items.

<b>NOTE</b>	<i>Instead of loading custom images from a computer folder, you can create an Access table to store your images and then use the Recordset object in the getImage callback to read the images from the table. This table should contain at least two fields: the ControlID field with the name of the control and the ImageFileName field specifying the name of the image file for the control. Custom images can also be stored and loaded from an Attachment field, which is available in Access databases created in the .accdb file format.</i>
-------------	--

### **Understanding Attributes and Callbacks**

---

Ribbon controls have properties defined by attributes, such as id, label, enabled, screentip, and so on. By using a specific attribute you can modify the appearance of a control either at design time or at runtime. To define a control attribute at runtime, simply set it to the allowable value right in the Ribbon customization

XML markup. For example, you can provide the name for your control in the label attribute. The control label can contain up to 1,024 characters.

If the attribute value is unknown at design time, add the prefix “get” to the design-time attribute name and specify the name of the callback procedure or macro as the attribute value. For example, if the control’s label needs to be defined at runtime, use the getLabel attribute and specify the name of the callback procedure:

```
<group id="Today's Events" getLabel="getEventDate">
```

When the Ribbon is loaded, the procedure in the getLabel attribute will run and provide the actual value of the attribute:

```
Public Sub getEventDate(ctl As IRibbonControl, _
ByRef ReturnValue As Variant)

ReturnValue = "Events for " & Format(Now(), "mm/dd/yyyy")
End Sub
```

This procedure will display the current date in the name of the group label. Although many times you will see the callback procedure name prefixed by “get” or “onGet,” keep in mind that you do not have to give the callback procedure the same name as the attribute it is used with. Use any name that makes sense to you. The only requirement is that the callback procedure matches a particular *signature*, which is the declaration of the procedure, the parameters, and return types. For example, the callback for the onAction attribute of a button control has the following signature:

```
Public Sub NameOfCallback(control As IRibbonControl)
```

IRibbonControl is the control that was clicked. This control is passed to your procedure by the Ribbon. You can specify your own name for the control parameter. For example:

```
Public Sub NameOfCallback(ctl As IRibbonControl)
```

Before using the IRibbonControl, you need to add a reference to the Microsoft Office 16.0 Object Library in your VBA project. The onAction attribute is a special type of attribute that does not need to be prefixed by the word “get” to point to a callback procedure.

## USING VARIOUS CONTROLS IN RIBBON CUSTOMIZATIONS

---

Now that you know how to go about creating the XML markup for your Ribbon customizations as well as loading and applying the custom Ribbon to a database, form, or report, let's look at other types of controls you can show in the Ribbon to give your database application a more polished and professional look. You can reuse the EduSystems\_Local database used in the earlier examples to create additional Ribbon customizations that utilize the controls discussed in this section.

### Creating Toggle Buttons

---

A *toggle button* is a button that alternates between two states. Many formatting features such as Bold, Italic, or Format Painter are implemented as toggle buttons. When you click a toggle button, the button stays down until you click it again. To create a toggle button, use the <toggleButton> XML tag as shown here:

```
<toggleButton id="tglNewStudent" label="New Student  
Questionnaire"  
size="normal" getPressed="OnGetPressed" onAction="ShowHideQ" />
```

You can add a built-in image to the toggle button with the imageMso attribute, or use a custom image as discussed earlier in this chapter. To find out whether the toggle button is pressed, include the getPressed attribute in your XML markup. The getPressed callback procedure provides two arguments: the control that was clicked and the pressed state of the toggle button:

```
Sub OnGetPressed(control As IRibbonControl, _  
ByRef pressed)  
  
If control.id="tglNewStudent" then  
    pressed = False  
End If  
End Sub
```

The preceding callback routine will ensure that the specified toggle button is not pressed when the Ribbon is loaded.

To perform an action when the toggle button is clicked, set the onAction attribute to the name of your custom callback procedure. This callback also provides two arguments: the control that was clicked and the state of the toggle button:

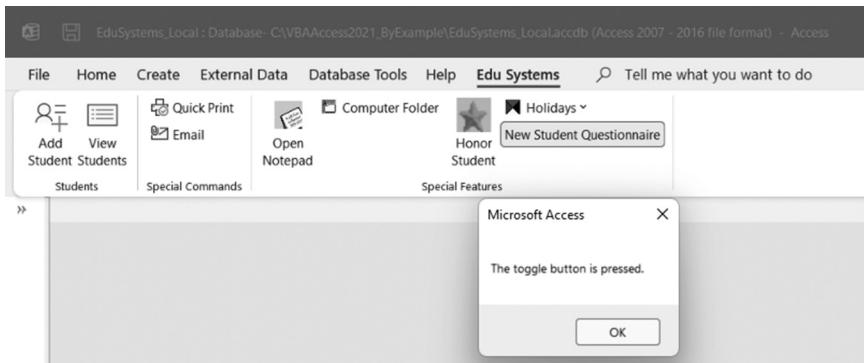
```
Sub ShowHideQ(control As IRibbonControl, pressed As Boolean)  
    If pressed Then
```

```

    MsgBox "The toggle button is pressed."
Else
    MsgBox "The toggle button is not pressed."
End If
End Sub

```

If the toggle button is pressed, the value of the pressed argument will be True; otherwise, it will be False. The toggle button named New Student Questionnaire is shown in Figure 23.35.



**FIGURE 23.35** The custom toggle button Student Questionnaire will become highlighted when pressed and will return to its normal state when clicked again.

#### **NOTE**

The XML for adding the Toggle button to the current Edu Systems tab can be found in **EduSystems\_07.txt** file in the companion files. The VBA code is in the **EduSystems\_07\_withToggle\_VBA.txt** file.

For each of the discussed controls, continue to make new records in the USysRibbons table and copy the XML customization code from the provided text file. See Figure 23.42 later in this chapter for the names of the created Ribbon tabs. If preferred, use your own names. If you find it difficult to do data entry in the RibbonXML field, press Shift+F2 for the Zoom dialog.

### **Creating Split Buttons, Menus, and Submenus**

A *split button* is a combination of a button or toggle button and a menu. Clicking the button performs one default action, and clicking the drop-down arrow opens a menu with a list of related options to select from. To create the split button, use the <splitButton> tag. Within this tag, you need to define a <button>

or a <toggleButton> control and the <menu> control, as shown in the following XML markup:

```
<group id="OtherControlsGroup" label="Other Controls" >
<splitButton id="btnSplit1" size="large" >
<button id="btnImport" label="Import More"
imageMso="ImportAccess" />
<menu id="mnuImport" label="More Import Formats"
itemSize="normal" >
<menuSeparator id="mnuDiv1" title="Other Databases" />
<button id="btnImportODBC" label="ODBC database"
imageMso="ImportOdbcDatabase" />
<button id="btnImportDbase" label="Dbase file"
imageMso="ImportDBase" />
<button id="btnImportParadox" label="Paradox file"
imageMso="ImportParadox" />
<menuSeparator id="mnuDiv2" title="Spreadsheet Files" />
<menu id="mnuExcel" label="Excel File Formats"
imageMso="ImportExcel" itemSize="normal" >
<checkBox id="xlsFormat" label="xls file" />
<checkBox id="xlsxFormat" label="xlsx file" />
</menu>
<button id="btnImportLotus" label="Lotus 1-2-3 file"
imageMso="ImportLotus" />
<menuSeparator id="mnuDiv3" title="Other Files" />
<button id="btnText" label="Text file"
imageMso="ImportTextFile" />
<button id="btnXML" label="XML file" imageMso="ImportXmlFile" />
<button id="btnHTML" label="HTML file"
imageMso="ImportHtmlDocument" />
<button id="btnOutlook" label="Outlook folder"
imageMso="ImportOutlook" />
<button id="btnSharepoint" label="SharePoint List"
imageMso="ImportSharePointList" />
</menu>
</splitButton>
</group>
```

**NOTE**

The <checkbox> tag used in the preceding example XML is discussed in detail in the next section.

You can specify the size of the items in the menu using the itemSize attribute. To display a description for each menu item below the item label, set the itemSize attribute to large (`itemSize="large"`) and use the description attribute to

specify the text. The <menuSeparator> tag can be used inside the menu node to break the menu into sections. Each menu segment can then be titled using the title attribute, as shown in the preceding example. You can add the onAction attribute to each menu button to specify the callback procedure or macro to execute when the menu item is clicked. In addition to button controls, menus can contain toggle buttons, checkboxes, gallery controls, split buttons, nested menus, and dynamic menus. Figure 23.36 displays the Ribbon with split buttons, menus, and submenus created in this section.

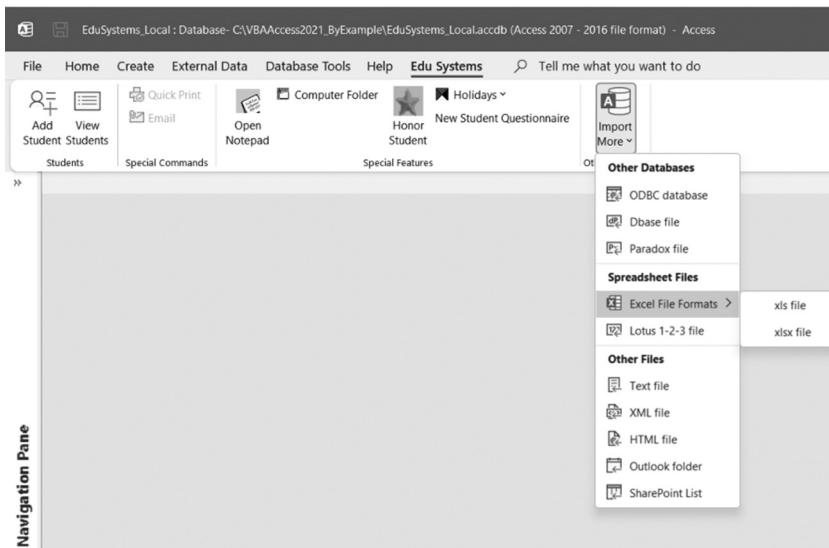


FIGURE 23.36 Custom split button controls can use the built-in Office images. They can also contain menus and submenus consisting of checkboxes.

See the EduSystems\_08\_withSplitPlusMenus.txt in the companion files for the XML code used to produce custom split button shown in Figure 23.36.

### Creating Checkboxes

The checkbox control is used to provide an option, such as true/false or on/off. It can be included inside a menu control as was demonstrated in the previous section or used as a separate control on the Ribbon. To create a checkbox, use the <checkbox> tag, as shown in the following XML:

```
<separator id="OtherControlsDiv1" />
<labelControl id="TitleForBox1" label="Areas of Interest (please
check below)" />
```

```
<box id="boxLayout1">
<checkBox id="chkSafety" label="School Safety"
    enabled="true" visible="true"
    onAction="DoSomething" />
<checkBox id="chkHealth" label="Health" enabled="false" />
<checkBox id="chkSportsMusic" getLabel="onGetLabel" />
</box>
```

In the preceding XML markup, the `<separator>` tag will produce the vertical bar that visually separates controls within the same Ribbon group (see Figure 23.37). The `<labelControl>` tag can be used to display static text anywhere in the Ribbon. In this example, we use it to place a header over a set of controls. To control the layout of various controls (to display them horizontally instead of vertically), use the `<box>` tag. You can define whether a checkbox should be visible or hidden by setting the `visible` attribute to true or false. To disable a checkbox, set the `enabled` attribute to false; this will cause the checkbox to appear grayed out. Notice that the checkbox labeled Health is not active (it is grayed out).

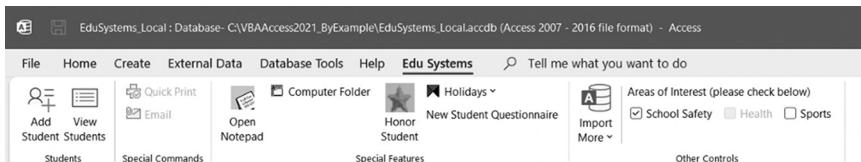


FIGURE 23.37 These checkbox controls are laid out horizontally.

See the **EduSystems\_09\_withCheckboxes.txt** in the companion files for the XML code used to produce the checkbox controls shown in Figure 23.37. The required VBA code can be found in **EduSystems\_09\_withCheckboxes\_VBA.txt**.

Similar to other controls, labels for checkboxes can contain static text in the `label` attribute, or they can be assigned dynamically using the callback procedure in the `getLabel` attribute:

```
<checkBox id="chkSportsMusic" getLabel="onGetLabel" />
```

The `getLabel` attribute points to the `onGetLabel` callback procedure, which needs to be added to your VBA module:

```
Public Sub onGetLabel(ctl As IRibbonControl, ByRef label)
If ctl.id = "chkSportsMusic" And _
Weekday(Now(), vbWednesday) Then
    label = "Sports"
```

```
    Else
        label = "Music"
    End If
End Sub
```

This procedure will run automatically when the Ribbon loads. If today happens to be Wednesday, you will see a checkbox for Sports; otherwise, it will be Music.

The action of the checkbox control is handled by the callback procedure in the `onAction` attribute:

```
<checkBox id="chkSafety" label="School Safety"
    enabled="true" visible="true"
    onAction="DoSomething" />
```

The `DoSomething` procedure needs to be added to the VBA module for the School Safety checkbox to respond to a user's click:

```
Public Sub DoSomething(ctl As IRibbonControl, _
    pressed As Boolean)
    If ctl.id = "chkSafety" And pressed Then
        MsgBox "Safety is our number one concern."
    Else
        MsgBox "Sorry to hear that safety is not your concern."
    End If
End Sub
```

To get the checked state for a checkbox, point to your callback procedure in the `getPressed` attribute, similar to what we've done earlier with the toggle button. The default VBA syntax for this callback is as follows:

```
Sub GetPressed(control As IRibbonControl, ByRef return)
```

**NOTE**

*As mentioned earlier, callback procedures don't need to be named the same as the attribute they are used with. Also, you may change the callback's argument names as desired.*

## Creating Edit Boxes

Use the `<editBox>` tag to provide an area on the Ribbon where users can type text or numbers:

```
<editBox id="txtFullName" label="First and Last Name:"
    sizeString="AAAAAAAAAAAAAAA" maxLength="25"
    onChange="onFullNameChange" />
```

Figure 23.38 shows the result of the preceding XML markup.

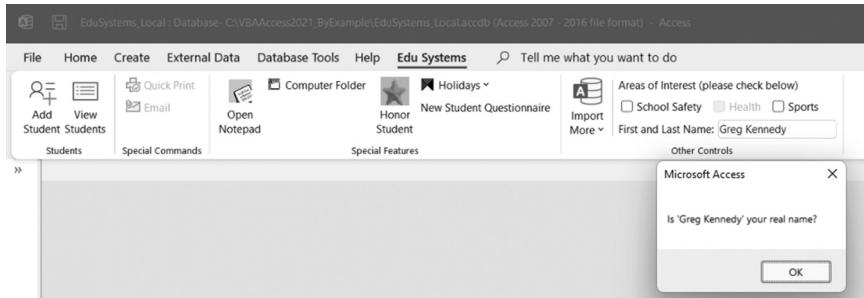


FIGURE 23.38 An edit box control allows data entry directly on the Ribbon.

See the **EduSystems\_10\_withEditBoxes.txt** file in the companion files for the XML code used to produce the edit box control shown in Figure 23.38. The required VBA code can be found in **EduSystems\_10\_withEditBoxes\_VBA.txt**.

The `sizeString` attribute specifies the width of the edit box. Set it to a string that will give you the width you want. The `maxLength` attribute allows you to limit the number of characters and/or digits that can be typed in the edit box. If the text entered exceeds the specified number of characters (25 in this case), Access automatically displays a balloon message on the Ribbon: “The entry may contain no more than 25 characters.”

When the entry is updated in an edit box control, the callback procedure specified in the `onChange` attribute is called:

```
Public Sub onFullNameChange(ctl As IRibbonControl, _
    text As String)
    If text <> "" Then
        MsgBox "Is '" & text & _
            "' your real name?"
    End If
End Sub
```

When the user enters text in the edit box, the procedure displays a message box.

### **Creating Combo Boxes and Drop Downs**

There are three types of drop-down controls that can be placed on the Ribbon: combo box, drop down, and gallery.

These controls can be dynamically populated at runtime by writing callbacks for their `getItemCount`, `getItemID`, `getItemLabel`, `getItemImage`, `getItemScreenTip`, or `getItemSupertip` attributes. The combo box and drop-down controls can

also be made static by defining their drop-down content using the <item> tag, as shown here:

```
<separator id="OtherControlsDiv2" />
<comboBox id="cmbLang" label="Languages"
supertip="Select Language Guide"
onChange="OnChangeLang" >
<item id="English" label="English" />
<item id="Spanish" label="Spanish" />
<item id="French" label="French" />
<item id="German" label="German" />
<item id="Russian" label="Russian" />
</comboBox>
```

To separate the combo box control from other controls in the same Ribbon group, this example uses the <separator> tag. Notice that each <item> tag specifies a new drop-down row.

**NOTE**

A combo box is a combination of a drop-down list and a single-line edit box, allowing the user to either type a value directly into the control or choose from the list of predefined options. Use the sizeString attribute to define the width of the edit box.

The combo box control does not have the onAction attribute. It uses the onChange attribute that specifies the callback to execute when the item selection changes:

```
Public Sub OnChangeLang(ctl As IRibbonControl,
text As String)

    MsgBox "You selected the " & text & " language guide."
End Sub
```

Notice that the onChange callback provides only the text of the selected item; it does not give you access to the selected index. If you need the index of the selection, use the dropdown control instead, as shown here:

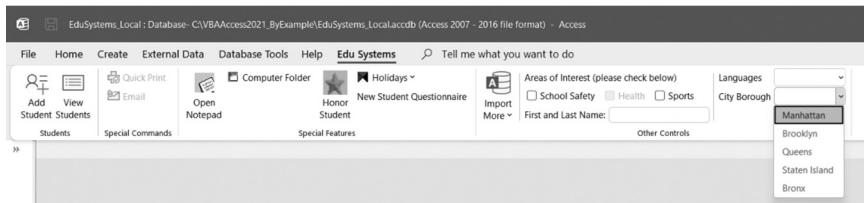
```
<dropDown id="drpBoro" label="City Borough"
supertip="Select School Borough"
onAction="OnActionBoro" >
<item id="M" label="Manhattan" />
<item id="B" label="Brooklyn" />
<item id="Q" label="Queens" />
<item id="I" label="Staten Island" />
<item id="X" label="Bronx" />
</dropDown>
```

The `onAction` callback of the drop-down control will give you both the selected item's ID and its index:

```
Public Sub OnActionBoro(ctl As IRibbonControl, _
ByRef SelectedID As String, _
ByRef SelectedIndex As Integer)

    MsgBox "Index=" & SelectedIndex & " ID=" & SelectedID
End Sub
```

Figure 23.39 shows the combo box and drop-down controls created in this section.



**FIGURE 23.39** The Languages combo box and City Borough drop-down controls look the same on the Ribbon.

See the **EduSystems\_11\_withComboAndDropDowns.txt** file in the companion files for the XML code used to produce the combo and drop-down controls shown in Figure 23.39. The required VBA code can be found in **EduSystems\_11\_withComboAndDropDowns\_VBA.txt**.

#### NOTE

*The gallery control was introduced earlier in this chapter in the section titled “Requesting Images via the getImage Callback.” This control cannot be static; it must be dynamically populated at runtime.*

### Creating a Dialog Box Launcher

Some Ribbon tabs have a small dialog launcher button at the bottom-right corner of a group (see Figure 23.15 earlier). You can use this button to open a special form that allows the user to set up many options at once, or you can display a form that contains specific information. To add a custom dialog launcher button to the Ribbon, use the `<dialogBoxLauncher>` tag, as shown here:

```
<dialogBoxLauncher>
<button id="Launch1">
```

```
screentip="Show Product Key"
onAction="OnActionLaunch" />
</dialogBoxLauncher>
```

The dialog box launcher control must contain a button. The OnAction attribute for the button contains the callback procedure that will execute when the button is clicked:

```
Public Sub OnActionLaunch(ctl As IRibbonControl)
    ' open the About Microsoft Office Access box
    DoCmd.RunCommand acCmdAboutMicrosoftAccess
End Sub
```

The dialog box launcher control must appear as the last element within the containing group element in the XML markup. The entire definition of the custom Edu Systems Ribbon tab created in this chapter and depicted in Figure 23.40 is available in the **EduSystems\_12\_withDialogLauncher.txt** file in the companion files. The required VBA procedure is in **EduSystems\_12\_withDialogLauncher\_VBA.txt**.

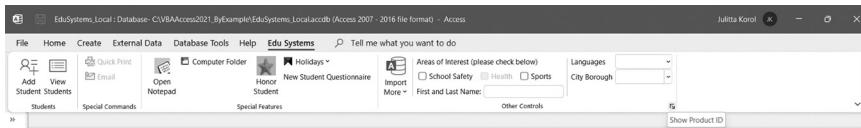


FIGURE 23.40 A dialog box launcher control on the Ribbon.

## Disabling a Control

You can disable a built-in or custom Ribbon control by using the enabled or getEnabled attribute. Here's how we disabled our custom checkbox control earlier by using the enabled attribute:

```
<checkBox id="chkHealth" label="Health" enabled="false" />
```

Use the getEnabled attribute to disable a control based on some conditions or simply display a “not authorized” message. The following XML code shows how to disable the built-in Relationships button on the Ribbon’s Database Tools tab:

```
<!-- Built-in commands section -->
<commands>
    <command idMso="DatabaseRelationships"
    onAction="DisableRelations" />
</commands>
```

To make your XML code more readable, you can include comments between the <!-- and --> characters. The <command> tag can be used to refer to any built-in command. This tag must appear in the <commands> section of the XML code.

To see the exact position of the above XML markup in the Ribbon customization, open the **EduSystems\_13\_DisableAndRepurpose.txt** file in the companion files. Notice the built-in command section just before the line:

```
<ribbon startFromScratch="false">
```

The onAction attribute contains the following callback procedure that will display a message when the Relationships button is clicked:

```
Sub DisableRelations(ctl As IRibbonControl, _
ByRef cancelDefault)

    MsgBox "You are not authorized to use this function."
    cancelDefault = True
End Sub
```

You can add more code to this procedure if you need to cancel the control's default behavior only when certain conditions have been satisfied.

### **Repurposing a Built-in Control**

---

It is possible to change the purpose of a built-in Ribbon button. For example, when the user clicks the DatabaseDocumentor button (Database Tools | Analyze Group) while the Student List form is open, you could display a Database Properties dialog box instead of the default Documentor dialog box:

```
<command idMso="DatabaseDocumentor" onAction="ShowDbProperties" />

Public Sub ShowDbProperties(ctl As IRibbonControl, _
ByRef cancelDefault)

    If CurrentProject.AllForms("Student List").IsLoaded Then
        ' display Database Properties dialog box instead
        DoCmd.RunCommand acCmdDatabaseProperties
    Else
        cancelDefault = False
    End If
End Sub
```

Only simple buttons that perform an action when clicked can be repurposed. You cannot repurpose advanced controls such as combo boxes, drop downs, or galleries.

See the **EduSystems\_13\_DisableAndRepurpose.txt** for the XML code.

The matching VBA file can be found in the companion files (**EduSystems\_13\_DisableAndRepurpose\_VBA.txt**).

## Refreshing the Ribbon

---

So far in this chapter you've seen how to use callback procedures to specify the values of control attributes at runtime. But what if you need to update your custom Ribbon or the controls placed in the Ribbon based on what the user is doing in your application? The good news is that you can change the attribute values at any time by using the `InvalidateControl` method of the `IRibbonUI` object. To use this object, start by adding the `onLoad` attribute to the `customUI` element in your Ribbon customization XML:

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/
customui"
loadImage="OnLoadImage" onLoad="RefreshMe" >
```

The `onLoad` attribute points to the callback procedure that will give you a copy of the Ribbon that you can use to refresh anytime you want. In this example, the `onLoad` callback procedure name is `RefreshMe`.

Let's say you have a checkbox that is disabled when the Ribbon is first loaded and you would like to enable it when the user enters text in an edit box. Also, upon entry you want the text of the edit box to appear in uppercase. To implement the `onLoad` callback, start by declaring a Public module-level variable of type `IRibbonUI`:

```
Public objRibbon As IRibbonUI
```

The preceding statement should appear in the declaration section at the top of the VBA module. To keep track of the state of the two Ribbon controls we are interested in, declare two Private module-level variables:

```
Private strUserTxt As String
Private isCtlEnabled As Boolean
```

Next, enter the callback procedure that will store a copy of the Ribbon in the `objRibbon` variable and assign an initial value to the `isCtlEnable` variable:

```
' callback for the onLoad attribute of customUI
Public Sub RefreshMe(ribbon As IRibbonUI)
    Set objRibbon = ribbon
    isCtlEnabled = False
End Sub
```

When the Ribbon loads, the checkbox control will be disabled. You will also have a copy of the IRibbonUI object saved for later use. Now, let's take a look at the XML markup used in this scenario:

```
<checkBox id="chkHealth" label="Health"  
getEnabled="onGetEnabled_Health" />  
<editBox id="txtFullName" label="First and Last Name:"  
sizeString="AAAAAAAAAAAAAAA" maxLength="25"  
getText="getEditBoxText" onChange="onFullNameChangeToUcase" />
```

These checkbox and edit box controls were introduced earlier in this chapter (see Figures 23.38 and 23.39). In order to change the enabled state of the checkbox control based on the user action, the getEnabled attribute must be used. The callback procedure for this attribute is as follows:

```
Public Sub onGetEnabled_Health(control As IRibbonControl, _  
    ByRef enabled)  
    enabled = isCtlEnabled  
End Sub
```

When the Ribbon is loaded, the onGetEnabled\_Health procedure will provide the value for the getEnabled attribute. The Health checkbox will be displayed in the Ribbon in its disabled mode because we have set the value of the isCtlEnabled variable to `False` in the RefreshMe procedure.

The edit box control contains two attributes that require callback procedures. The getText attribute points to the following callback:

```
Public Sub getEditBoxText(control As IRibbonControl, _  
    ByRef text)  
    text = UCASE(strUserTxt)  
End Sub
```

The preceding callback uses the VBA built-in `UCASE` function to change the text that the user entered in the edit box to uppercase letters. When text is updated in the edit box, the procedure in the onChange attribute is called:

```
Public Sub onFullNameChangeToUcase(ByVal control As  
    IRibbonControl, _text As String)  
  
    If text <> "" Then  
        strUserTxt = text  
        objRibbon.InvalidateControl "txtFullName"  
        isCtlEnabled = True  
    Else  
        isCtlEnabled = False  
    End If
```

```

    objRibbon.InvalidateControl "chkHealth"
End Sub

```

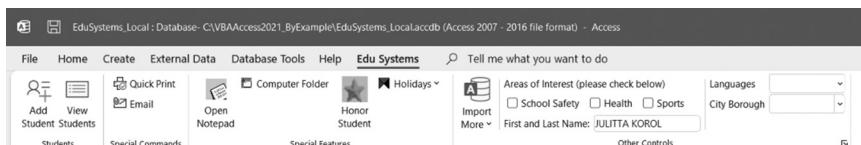
The preceding callback begins by checking the value of the `text` parameter provided by the Ribbon. If this parameter contains a value other than an empty string (" "), the text the user entered is stored in the `strUserTxt` variable. Before a change can occur in the Ribbon control, you need to mark the control as invalid. This is done by calling the `InvalidateControl` method of the `IRibbonUI` object that we have stored in the `objRibbon` variable:

```
objRibbon.InvalidateControl "txtFullName"
```

This statement will tell the `txtFullName` control to refresh itself the next time it is displayed. When the control is invalidated, it will automatically call its callback functions. The `getEditBoxText` callback procedure in the `onChange` attribute will execute, causing the text entered in the `txtFullName` edit box control to appear in uppercase letters.

The second action that we want to perform is to enable the `chkHealth` checkbox control when the user enters text in the edit box control and keep this button disabled when the edit box control is empty. This is done by setting the `isCtlEnabled` Boolean variable to `True` or `False` and invalidating the `chkHealth` checkbox control. When the `chkHealth` control is marked as invalid, it will call its callback functions. The `onGetEnabled_Health` callback procedure in the `getEnabled` attribute will execute, causing the control to appear in the enabled state if the `txtFullName` edit box control contains any text.

Figure 23.41 shows the Ribbon after it has been refreshed.



**FIGURE 23.41** The Ribbon controls are shown here after the Ribbon refresh. The Health checkbox is enabled upon entry of text in the First and Last Name edit box, and disabled when the entry is deleted.

The XML markup for the final Ribbon customization is contained in the **EduSystems\_14\_WithRefresh.txt** file in the companion files and the VBA code is found in **EduSystems\_14\_WithRefresh\_VBA.txt**.

You will find the completed customizations demonstrated in the preceding sections of this chapter in the **EduSystems\_Local.accdb** file in the **VBAAccess2021\_ByExample** folder in companion files.

Figure 23.42 shows the names of all custom Ribbons we created in that database. At this point, your USysRibbons table should contain 12 records, each one representing a different Ribbon.

<b>NOTE</b>	<i>The IRibbonUI object has only two methods: InvalidateControl and Invalidate. Use the InvalidateControl method to refresh an individual control. Use the Invalidate method to refresh all controls in the Ribbon.</i>
-------------	---

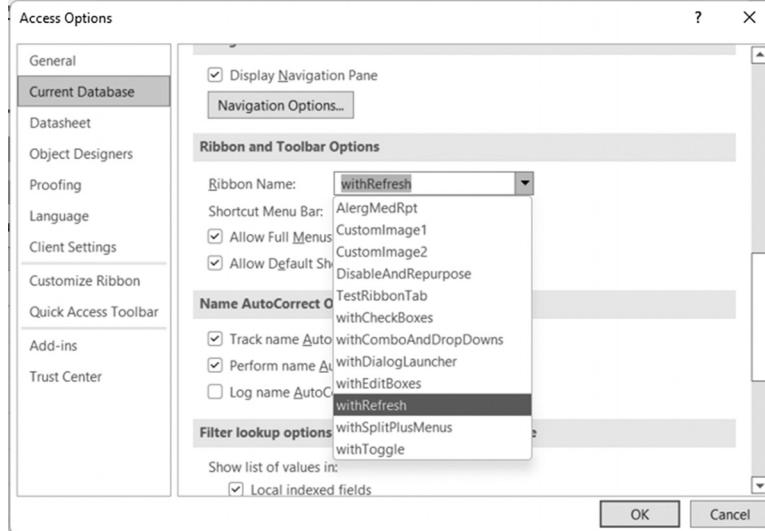


FIGURE 23.42 Each time you apply a different Ribbon customization you need to close and reopen the Access database.

## THE COMMANDBARS OBJECT AND THE RIBBON

You can make your custom Ribbon button match any built-in button by using the CommandBars object. This object has been extended with several get methods that expose the state information for the built-in controls: GetEnabledMso, GetImageMso, GetLabelMso, GetPressedMso, GetScreentipMso, GetSupertipMso, and GetVisibleMso. Use these methods in your callbacks to check the built-in control's properties. For example, the following statement will return False if the Ribbon's built-in Cut button is currently disabled (grayed out), and True if it is enabled (ready to use):

```
MsgBox Application.CommandBars.GetEnabledMso("Cut")
```

Notice that the `GetEnabledMso` method requires that you provide the name of the built-in control. To see the result of the preceding statement, simply type it in the Immediate window and press Enter.

The `GetImageMso` method is very useful if you'd like to reuse any of the built-in button images in your own controls. This method allows you to get the bitmap for any `imageMso` tag. For example, to retrieve the bitmap associated with the Cut button on the Ribbon, enter the following statement in the Immediate window:

```
MsgBox Application.CommandBars.GetImageMso("Cut", 16, 16)
```

The preceding `GetImageMso` method uses three arguments: the name of the built-in control, and the width and height of the bitmap image in pixels. Because this method returns the `IPictureDisp` object, it is very easy to place the retrieved bitmap onto your own custom Ribbon control by writing a simple VBA callback for your control's `getImage` attribute.

In addition to the methods that provide information about the properties of the built-in controls, the `CommandBars` object also includes a handy `ExecuteMso` method that can be used to trigger the built-in control's default action. This method is quite useful when you want to perform a click operation for the user from within a VBA procedure or want to conditionally run a built-in feature.

Let's take a look at the example implementation of the `GetImageMso` and `ExecuteMso` methods. Here's the XML definition for a custom Ribbon button (see Figure 23.43):

```
<button id="btnRptWizard" label="Use Report Wizard" size="normal"
       getImage="onGetBitmap" onAction="DoDefaultPlus" />
```

The preceding XML code can be added to any of the custom Ribbon definitions you've already defined in the `USysRibbons` table. Now let's look at the VBA part.

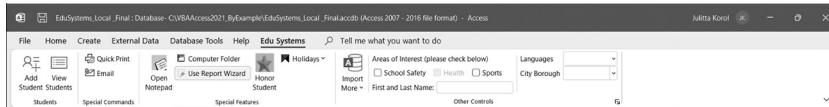
Suppose you want the button to use the same image as the built-in button labeled Report Wizard. When the button is clicked, you'd like to display the built-in Report Wizard dialog box only when a certain condition is true. Here is the code you need to add to your VBA module:

```
Sub onGetBitmap(ctl As IRibbonControl, ByRef image)
    Set image = Application.CommandBars.-
        GetImageMso("CreateReportFromWizard", 16, 16)
End Sub
```

When the Ribbon is loaded, the `onGetBitmap` callback automatically retrieves the image bitmap from the Report Wizard button's `imageMso` attribute and

assigns it to the getImage attribute of your button. When your button is clicked and the Student List form is open, the Report Wizard dialog box will pop up; if the specified object is not open, the user will see a message box:

```
Sub DoDefaultPlus(ctl As IRibbonControl)
    If Application.CurrentObjectName = "Student List" Then
        Application.CommandBars.ExecuteMso "CreateReportFromWizard"
    Else
        MsgBox "To run this Wizard you need to open " & _
            " the Student List Form", _
            vbOKOnly + vbInformation, "Action Required"
    End If
End Sub
```



**FIGURE 23.43** The custom button (Use Report Wizard) in the Special Features group of the Edu Systems tab uses a built-in image and runs a built-in Access feature based on the condition specified in the callback assigned to its onAction attribute. Notice the new database name is EduSystems\_Local\_Final, which is a copy of the EduSystems\_Local database and contains all the controls created so far in this chapter.

You will find the XML markup discussed in this section in the **EduSystems\_15\_withCommandBars.txt** file in the companion files. The sample VBA code is included in **EduSystems\_15\_withCommandBars\_VBA.txt**.

## TAB ACTIVATION AND GROUP AUTO-SCALING

Tab activation makes it possible to activate a specific tab in response to some event. To activate a custom tab on the Access 2021 Ribbon, use the `ActivateTab` method of the `IRibbonUI` object by passing to it the ID of the custom string. For example, to activate the Edu Systems tab you created in this chapter, try the following statement in the Immediate window while any of the default Access tabs is active:

```
objRibbon.ActivateTab "custTabEdu"
```

Recall that `objRibbon` is the module-level Public variable we declared earlier for accessing the `IRibbonUI` object. To activate a built-in tab, use the `ActivateTabMso` method. For example, the following statement activates the Create tab:

```
objRibbon.ActivateTabMso "TabCreate"
```

Finally, there is also a special `ActivateTabQ` method used to activate a tab shared between multiple add-ins. In addition to the tabID, this method requires that you specify the namespace of the add-in. The syntax is shown here:

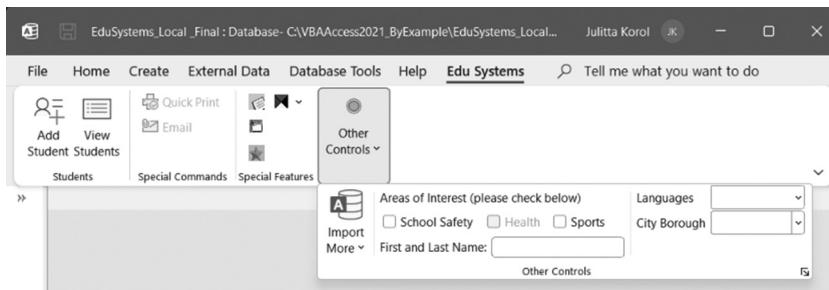
```
expression.ActivateTabQ(tabID As String, namespace as String)
```

where `expression` returns an `IRibbonUI` object. Keep in mind that tab activation applies only to tabs that are visible.

Group auto-scaling enables custom Ribbon groups to change their layout when the user resizes the window (see Figure 23.44). You can enable auto-scaling by setting the `autoScale` attribute of the `<group>` tab to true as in the following:

```
<group id="ImagesGroup" label="Special Features" autoScale="true">
```

Notice that the value of the `autoScale` attribute is entered in lowercase. Auto-scaling is set on a per-group basis.



**FIGURE 23.44** The commands in the Other Controls group of the Ribbon are automatically compressed to a single button when the Access application window is made smaller. To change the icon that appears when the group is compressed, assign an image to the group itself. When you set the `autoScale` attribute to true, the group of controls in Special Features will change its layout to best fit the resized window.

You will find the Ribbon customizations discussed in this section in the **EduSystems\_16\_WithAutoSize.txt** file in the companion files.

## CUSTOMIZING THE BACKSTAGE VIEW

---

The Access File tab provides an entry point to a part of the Office UI known as Backstage View. This view is specifically designed for working with a database as a whole. It contains commands known as *Fast commands* that provide quick access to common functionality such as saving, opening, or closing a database.

Here you also find the Exit command for exiting Access and the Options command for customizing numerous Access features. In addition to Fast commands, the navigation bar on the left-hand side of the Backstage View includes several tabs that group related tasks. For example, clicking the Print tab in the navigation bar displays all the information related to the installed printers and allows you to easily access and change many of the print settings. The Info tab organizes tasks related to compacting and repairing a database and encrypting it with a password. As an Access developer already familiar with Ribbon UI customization, you should feel very comfortable customizing the Backstage View. Like the Ribbon, the Backstage View uses XML markup. The Backstage View is a perfect place to include custom solutions that present summaries of business processes or workflows. In this section, you'll perform some simple operations in the Backstage View to get started with the customization of this interface.

The Backstage View XML markup should be entered between <backstage> </backstage> elements within the <customui> </customui> tags and below any Ribbon customization markup.

The **EduSystems\_17\_WithBackstageView.txt** file in the companion files contains the XML markup that adds a custom button named Synchronize and a custom tab named Endless Possibilities to the Backstage View. When you add a new Ribbon record to the USysRibbons table using the XML code in that file, the Backstage customization should match Figure 23.45.

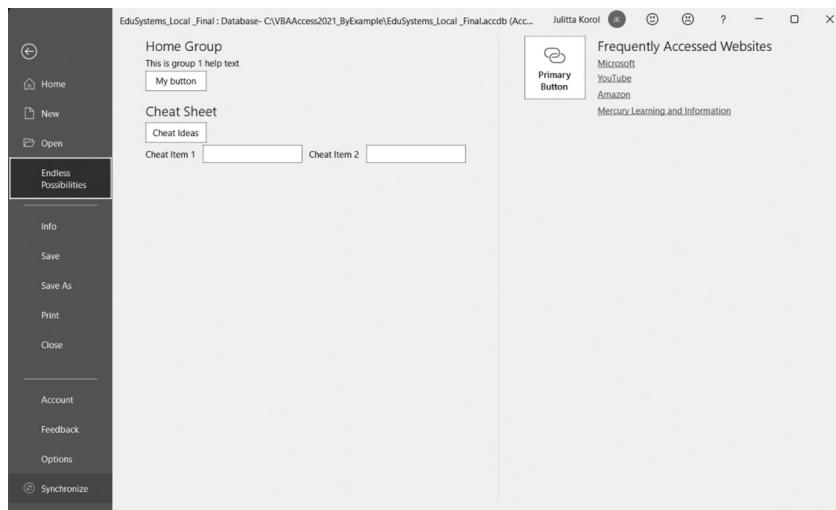


FIGURE 23.45 The Backstage View is highly customizable. The Synchronize button and the Endless Possibilities tab were created by adding custom XML markup to the USysRibbons system table.

Open the **EduSystems\_17\_WithBackstageView.txt** file to analyze its XML markup. Note that the `<button>` element is used to incorporate into the navigation bar of the Backstage View a custom command labeled Synchronize:

```
<button id="btnSync" label="Synchronize" imageMso="SyncNow"
isDefinitive="true" insertBeforeMso="FileClose" onAction="onActi
onCopyToArchive" />
```

The `<button>` element contains the `isDefinitive` attribute. When this attribute is set to `true`, clicking the button will trigger the callback procedure defined in the `onAction` attribute and then automatically close the Backstage View. The `onAction` callback for the custom Synchronize button is shown here. The callback calls the `CreateDbCopy` procedure that allows you to make a copy of the specified database.

```
Sub onActionCopyToArchive(ctl As IRibbonControl)
    CreateDbCopy
End Sub

Sub CreateDbCopy()
    Dim fso As Object
    Dim dbName As String
    Dim dbNewName As String

    On Error GoTo ErrorHandler

    Set fso = CreateObject("Scripting.FileSystemObject")

    dbName = InputBox("Enter the name of the database " & _
        "you want to copy: " & _
        "(C:\VBAAccess2021_ByExample\Chap20.accdb)", _
        "Create a copy of")

    If dbName = "" Then Exit Sub
    If Dir(dbName) = "" Then
        MsgBox dbName & " was not found. " & Chr(13) & _
            "Check the database name or path."
        Exit Sub
    End If

    dbNewName = InputBox("Enter the name for the " & _
        "copied database:" & Chr(13) & _
        "(C:\VBAAccess2021_ByExample\Chap20Ver2.accdb)", _
        "Save As")
    If dbNewName = "" Then Exit Sub
```

```

If Dir(dbNewName) <> "" Then
    Kill dbNewName
End If

fso.CopyFile dbName, dbNewName
Set fso = Nothing

Exit Sub
ErrorHandler:
    MsgBox Err.Number & ":" & Err.Description
End Sub

```

The Backstage View XML markup also adds to the navigation bar of the Backstage View a custom tab labeled Endless Possibilities. Each `<tab>` element can have one or more columns. Our example contains two columns. Each tab can contain multiple `<group>` elements. Here we have two groups in the first column and one group in the second column. The Backstage group can contain different types of controls. You can group the controls into the following three types of sections:

<code>&lt;primary item&gt;</code>	This element is used to specify the most important item in the group. The primary item control can be a button or a menu with buttons, toggle buttons, checkboxes, or another menu.
<code>&lt;topItems&gt;</code>	This element defines controls that will appear at the top of the group.
<code>&lt;bottomItems&gt;</code>	This element defines the controls that will appear at the bottom of the group.

The layout of controls in the Backstage View is defined using the `<layoutContainer>` element. This element's `layoutChildren` attribute can define the layout of controls as horizontal or vertical. The second column of our example XML markup uses the `onActionExecHyperlink` callback procedure for the hyperlinks shown in Figure 23.45.

```

Sub onActionExecHyperlink(ctl As IRibbonControl, _
    ByRef target)
    Select Case ctl.ID
        Case "YouTube"
            target = "http://www.YouTube.com"
        Case "amazon"
            target = "http://www.amazon.com"
        Case "merc"
            target = "http://www.merclearning.com"
        Case "msft"
            target = "http://www.Microsoft.com"
        Case Else
            MsgBox "You clicked control id " & ctl.ID & _
                " that has not been programmed!"
    End Select
End Sub

```

```
End Select  
End Sub
```

**SIDE BAR** *Hiding Backstage Buttons and Tabs*

The following XML will hide the Options button in the Backstage View navigation bar:

```
<button idMso="ApplicationOptionsDialog" visible="false" />
```

The Backstage View uses the following button IDs: FileSave, FileSaveAs, FileOpen, FileClose, ApplicationOptionsDialog, and FileExit.

To hide the Info tab in the Backstage View, use this markup:

```
<tab idMso="TabInfo" visible="false" />
```

The Backstage View tab IDs are as follows: TabInfo, TabRecent, TabNew, TabPrint, TabShare, and TabHelp.

**SIDE BAR** *Things to Remember when Customizing the Backstage View*

- The maximum number of allowed tabs is 255.
- You cannot reorder built-in tabs.
- You can add your custom tab before or after the built-in tab.
- You cannot modify the column layout of any built-in tab.
- You cannot reorder built-in groups; however, you can specify the order of groups you create.

---

## CUSTOMIZING THE QUICK ACCESS TOOLBAR (QAT)

---

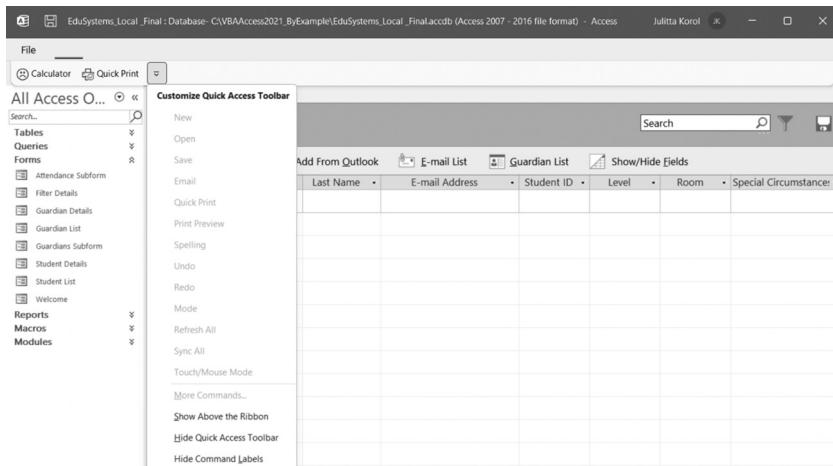
As mentioned in the beginning of this chapter, the Quick Access toolbar has a changed location and look in the current version of Access. The Quick Access toolbar can only be customized at the start from scratch mode by setting the startFromScratch attribute to true in the Ribbon XML customization file:

```
<ribbon startFromScratch="true">
```

The preceding XML markup will hide all built-in tabs. You must add your own custom tabs as demonstrated earlier in this chapter. Quick Access toolbar modifications are specified using the <qat> element. Within this element you should use the <documentControls> element to specify the controls that you want to appear in the Quick Access toolbar. The following XML markup creates the

custom Quick Access toolbar shown in Figure 23.46. You will find this code in the **CustomUI\_withQAT.txt** file located in the companion files. The example VBA procedure that opens the Calculator is included in the **CustomUI\_withQUAT\_VBA.txt** file. Please note that to see the controls placed on the QAT, you need to make sure that the Quick Access Toolbar is enabled (see the first figure in this chapter).

```
<customUI
  xmlns="http://schemas.microsoft.com/office/2009/07/customui" >
<ribbon startFromScratch="true">
  <qat>
    <documentControls>
      <button id="btnCalc2" label="Calculator"
        imageMso="SadFace" onAction="OpenCalculator" />
      <button idMso="FilePrintQuick" />
    </documentControls>
  </qat>
</ribbon>
</customUI>
```



**FIGURE 23.46** Customized Quick Access toolbar.

The button labeled Calculator that is represented by the SadFace image calls the OpenCalculator procedure shown here:

```
Public Sub OpenCalculator(ctl As IRibbonControl)
  Shell "Calc.exe", vbNormalFocus
End Sub
```

The **EduSystems\_Local\_Final.accdb** database in the **VBAAccess2021\_ByExample** folder in the companion files contains all the Ribbon customizations introduced in this chapter.

## SUMMARY

---

This chapter introduced you to using and customizing the user interface in Access 2021. After a short overview of the initial Access screen and the Quick Access toolbar, we looked at numerous features of the Access Navigation pane. You learned how to use the Navigation pane to access and organize your database objects by using both manual techniques and VBA code. Next, we briefly covered the Ribbon interface to get you warmed up and ready for the Ribbon customization exercises. You learned how to create XML Ribbon customization markup and load it in your database by using the `LoadCustomUI` method of the Application object and via a special Access system table called USysRibbons. You also learned how Ribbon customizations can be assigned to forms or reports. You spent quite a bit of time in this chapter familiarizing yourself with various controls that can be added to the Ribbon and writing callback procedures in order to set your controls' attributes at runtime. In addition to Ribbon customizations, you learned how to modify the Quick Access Toolbar (QAT).

While this chapter introduced many controls and features of the Ribbon, it did not attempt to cover all there is to know about this interface. After all, this book is about VBA programming in Access in general, not just the Ribbon. The knowledge and experience you gained in this chapter can be applied to customizing the Ribbon in all of the Microsoft 365 applications.

In the next chapter, we will examine more advanced concepts in Access VBA programming.



Part

# VII

## *ADVANCED CONCEPTS IN ACCESS VBA*

**M**icrosoft Access offers numerous built-in objects that you can access from your VBA procedures to automate many aspects of your databases. You are not limited to using these built-in objects, however. VBA allows you to create your own objects and collections of objects, complete with their own methods and properties. In this part of the book, you learn how thinking in terms of objects can help you write reusable code that's easy to maintain. In the next two chapters, you'll be working in a new type of module, known as a class module, as well as creating and using classes, and responding to class events.

Chapter 24 Creating Classes in VBA

Chapter 25 Advanced Event Programming



# Chapter 24 CREATING CLASSES IN VBA

**S**o far in this book, you learned that there are many ways to perform the same tasks in Access. You've written many procedures and functions that contained programming code that you might want to reuse in other Access projects. As your Access applications become more complex, you may find that your code is scattered all over the place and is difficult to maintain. Copying code from place to place and modifying procedures to include more arguments and enhancements because of the changing requirements will result in creating a coding mess that is difficult to manage. Access, however, has a special feature known as a *class module* that allows you to create code that is self-contained and reusable. The class helps you organize your code into manageable objects that you can easily reuse and adjust when necessary. Classes also make it easier to share your programming code with others. The class hides its inner workings from the rest of the program. Any programmer can use the class without knowing how that class was put together. It's like driving a car. A driver does not need to know the intricate details of how a car is manufactured.

By getting acquainted with classes, you'll be able to move from the procedural programming that you're already familiar with into object-oriented programming (OOP), where you program by creating objects that may contain data defined as attributes or properties, and code, defined as procedures, or methods. Objects are the basic units of OOP and are created by adding class

modules to your Access applications. A class is like a cookie cutter. Once you create it, you can make any number of cookies. In Access, your cookies will be the custom objects you create from a class defined in a class module.

## **IMPORTANT TERMINOLOGY**

---

In this chapter, you will work with advanced VBA concepts: VBA classes, class objects, and collection classes. Before examining the theory and this chapter's hands-on examples, let's review the following terms:

**Object**—An object is a logical representation of a thing. This thing can be a tangible, physical entity such as person, a car, a customer or an employee, or a logical entity such as an order, a transaction, or a report. It can also be something related to your specific Access applications, such as a process that simplifies data validation or the manipulation of data. In other words, an object is basically anything you need to define.

**Collection**—An object that contains a set of related objects.

**Class**—A definition of an object that includes its name, properties, methods, and events. The class acts as a sort of object template from which an instance of an object is created at runtime.

**Instance**—A specific object that belongs to a class is referred to as an *instance of the class*. When you create an instance, you create a new object that has the properties and methods defined by the class.

**Class module**—A module that contains the definition of a class, including its property and method definitions.

**Form module**—A module that contains the VBA code for all event procedures triggered by events occurring in a user form or its controls. A form module is a type of class module.

**Report module**—A module that contains the VBA code for all event procedures triggered by events occurring in a report or its controls. A report module is a type of class module.

**Module**—A structure containing subroutine and function procedures that are available to other VBA procedures and are not related to any specific object. You perform the procedural programming in modules, often referred to as *standard modules*.

**Event**—An action recognized by an object, such as a mouse click or a key-press, for which you can define a response. Events can be triggered by a user action, a VBA statement, or the system. In addition to form and report events and events for their controls, class modules also have events, Class\_Initialize and Class\_Terminate events, that provide control over how various variables and resources used by the class module are initialized and cleaned up.

**Event procedure**—A procedure that is automatically executed in response to an event triggered by the user, program code, or the system.

## **CREATING CUSTOM OBJECTS IN CLASS MODULES**

---

There are two module commands available in the Visual Basic Editor's Insert menu: Module and Class Module. So far, you've used a module to create sub procedures and function procedures. You'll use the class module for the first time in this chapter to create a custom object and define its properties and methods.

Creating a new VBA object involves inserting a class module into your project and adding code to that module. However, before you do so you need a basic understanding of what a class is.

If you refer to the list of terms at the beginning of this chapter, you will find out that the *class* is a sort of object template. A frequently used analogy is comparing an object class to a cookie cutter. Just like a cookie cutter defines what a cookie will look like; the definition of the class determines how a particular object should look and how it should behave. Before you can use an object class, you must first create a new *instance* of that class. Object instances are the cookies. Each object instance has the characteristics (properties and methods) defined by its class. Just as you can cut out many cookies using the same cookie cutter, you can create multiple instances of a class. You can change the properties of each instance of a class independently of any other instance of the same class.

A *class module* lets you define your own custom classes, complete with custom properties and methods.

A *property* is an attribute of an object that defines one of its characteristics, such as shape, position, color, title, and so forth. For example, a person's attributes include their name, address, and date of birth. You can create the properties for your custom objects by writing property procedures in a class module.

A *method* is an action that the object can perform. For example, a car object can be steered, accelerated, and stopped. The object methods are created in a class module by writing sub procedures or function procedures.

After building your object in the class module, you can use it in the same way you use other Access built-in objects. You can also export the object class outside the VBA project to other VBA-capable applications.

### Creating a Class

Working with classes can seem a daunting task when you've already gotten used to procedural programming. Again, working by example can make it much easier for you to understand how classes can help you break your bigger programming problems into reusable and organized components.

The following sections of this chapter walk you through the process of creating and working with a custom object called `CEmployee`. This object will represent an employee. It will have properties such as ID, FirstName, LastName, and Salary. It will also have a method to modify the current salary.

**NOTE**

*All code files and figures for the hands-on projects may be found in the companion files.*



#### Custom Project 24.1 (Part 1) Creating a Class Module

1. Start Access and create a new database named **Chap24.accdb** in your C:\VBAAccess2021\_ByExample folder.
2. Press **ALT+F11** to switch to the Visual Basic Editor window, and choose **Insert | Class Module**.
3. In the Project Explorer window, highlight the **Class1** module and use the Properties window to rename the class module **CEmployee** (see Figure 24.1).

**SIDE BAR**

#### Naming a Class Module

Every time you create a new class module, Access gives the class a default name Class1, Class2, and so forth. Before working with the class, give it your own meaningful name. Set the name of the class module to the name you want to use in your VBA procedures using the class. The name you choose for your class should be easily understood and should identify the “thing” the object class represents. As a rule, the object class name is prefaced with an uppercase “C.”

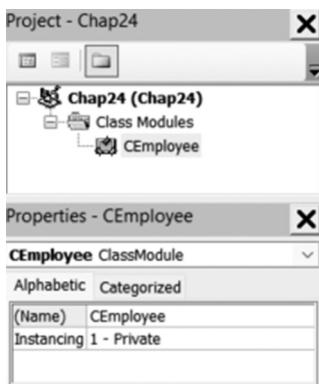


FIGURE 24.1 Use the Name property in the Properties window to rename the Class module.

## Variable Declarations

After adding and renaming the class module, the next step is to declare the variables that will hold the data you want to store in your custom CEmployee object. As mentioned earlier, the CEmployee object will have ID, FirstName, LastName, and Salary properties. Each item of data you want to store in an object should be assigned a variable. Class variables are called *data members* and are declared with the `Private` keyword. Using the `Private` keyword in a class module hides the data members and prevents other parts of the application from referencing them. Only the procedures within the class module in which the private variables were defined can modify the value of these variables.

Because the name of a variable also serves as a property name, use meaningful names for your object's data members. It's traditional to preface the class variable names with "m\_" to indicate that they are data members of a class.



### Custom Project 24.1. (Part 2) Declaring Class Members

1. Type the following declaration lines at the top of the CEmployee class module's code window:

```
Option Explicit
```

```
' declarations
Private m_LastName As String
Private m_FirstName As String
Private m_Salary As Currency
Private m_ID As String
```

Notice that the name of each data member variable begins with the prefix “m\_.” You can list your variables in any order you want.

When you declare the class member variables using the `Private` keyword, these variables can only be accessed through the Property code as detailed in the next Part of this custom project. By using the Property code, you can ensure that the values stored in these variables are valid and adhere to the specific rules of your application. For example, the `LastName` and `FirstName` cannot be blank and must begin with an uppercase letter.

### Defining the Properties for the Class

---

Declaring the variables with the `Private` keyword ensures that they cannot be directly accessed from outside the object. This means that the VBA procedures outside the class module will not be able to set or read data stored in those variables. To enable other parts of your VBA application to set or retrieve the employee data, you must add special property procedures to the `CEmployee` class module. There are three types of property procedures:

- **Property Get**—This type of procedure allows other parts of the application to get or read the value of a property. This property procedure works like a VBA function that returns some value.
- **Property Let**—This type of procedure allows other parts of the application to set the value of a property. This property procedure is used with simple data types such as numeric, string, and date properties.
- **Property Set**—This type of procedure is used instead of Property Let when setting the reference to an object. For example, it can be used when setting a property for a Recordset object or another object data type.

Property procedures are executed when an object property needs to be set or retrieved. The Property Get procedure can have the same name as the Property Let procedure. You should create property procedures for each property of the object that can be accessed by another part of your VBA application.

If you only create a Property Get procedure, the property becomes read-only and its value can't be changed by the program using the object. If you create only the Let or Set property procedure, the property becomes write-only, and its value can't be viewed by the calling program. Write-only properties are often used for sensitive information, such as passwords and login credentials.

The easiest of the three types of property statements to understand is the Property Get procedure. Let's examine the syntax of the property procedures by taking a close look at the `Property Get LastName` procedure.

Property procedures contain the following parts:

- A procedure declaration line that includes the `Property` keyword
- An assignment statement
- The `End Property` keywordsA procedure declaration line specifies the name of the property and the data type:

```
Property Get LastName() As String
```

`LastName` is the name of the property and `As String` determines the data type of the property's return value. The procedure declaration line must include the `Property` keyword.

Property Get procedures are always public by default so they can be exposed to the other parts of your application. You can use the `Public` keyword in front of the `Property` keyword to explicitly specify the property procedure's scope:

```
Public Property Get LastName() As String
```

An assignment statement is similar to the one used in a function procedure:

```
LastName = m_LastName
```

`LastName` is the name of the property and `m_LastName` is the data member variable that holds the value of the property you want to retrieve or set. The `m_LastName` variable should be defined with the `Private` keyword at the top of the class module. Here's the complete Property Get procedure:

```
Property Get LastName() As String
    LastName = m_LastName
End Property
```

The above property procedure retrieves the last name of the employee.

The Property Get procedures can also return a result from a calculation, like this:

```
Property Get Royalty()
    Royalty = (Sales * Percent) - Advance
End Property
```

The `End Property` keywords specify the end of the property procedure.

**SIDE BAR** *Immediate Exit from Property Procedures*

Just as the `Exit Sub` and `Exit Function` keywords allow you to exit early from a subroutine or a function procedure, the `Exit Property` keywords give you a way to immediately exit from a property procedure. Program execution will continue with the statements following the statement that called the `Property Get`, `Property Let`, or `Property Set` procedure.

*Creating the Property Get Procedures*

The `CEmployee` class object has four properties that need to be exposed to VBA procedures that we will write later in a standard module named `EmpOperations`. When working with the `CEmployee` object, you will need to get information about the employee ID, first and last name, and current salary. Let's see how this is done by writing `Property Get` procedures.

**Custom Project 24.1 (Part 3) Writing Property Get Procedures**

1. Type the following `Property Get` procedures in the `CEmployee` class module, just below the declaration section that you entered in Part 2 of this custom project:

```
' Property Get Procedures will retrieve
' Employee ID, first and last name and Salary
Property Get ID() As String
    ID = m_ID
End Property

Property Get FirstName() As String
    FirstName = m_FirstName
End Property

Property Get LastName() As String
    LastName = m_LastName
End Property

Property Get Salary() As Currency
    Salary = m_Salary
End Property
```

Notice that each employee information type requires a separate `Property Get` procedure. Each of the preceding `Property Get` procedures returns the current value of the property. Notice also how a `Property Get` procedure is like

a function procedure. Like function procedures, the Property Get procedures contain an assignment statement. As you recall from Chapter 4, to return a value from a function procedure, you must assign it to the function's name.

### ***Creating the Property Let Procedures***

---

In addition to retrieving values stored in data members (private variables) with Property Get procedures, you must prepare corresponding Property Let procedures to allow other procedures to change the values of these variables as needed. The only time you don't define a Property Let procedure is when the value stored in a private variable is meant to be *read-only*.

Let's continue with our project and write the required Property Let procedures for our custom CEmployee object.



### **Custom Project 24.1 (Part 4) Writing Property Let Procedures**

1. Type the following Property Let procedures in the CEmployee class module below the Property Get procedures:

```
' Property Let procedures assign values
' to employee first and last name and salary
Property Let FirstName(F As String)
    m_FirstName = F
End Property

Property Let LastName(L As String)
    m_LastName = L
End Property

Property Let Salary(ByVal dollar As Currency)
    m_Salary = dollar
End Property
```

The Property Let procedures require at least one parameter that specifies the value you want to assign to the property. This parameter can be passed by *value* (note the `ByVal` keyword in the preceding Property Let Salary procedure) or by *reference* (`ByRef` is the default, so we have skipped it here). If you need a refresher on the meaning of these keywords, see the section titled “Passing Arguments by Reference and by Value” in Chapter 4.

The data type of the parameter passed to the Property Let procedure must be the same data type as the value returned from the Property Get or Set procedure with the same name. Notice that the Property Let procedures have the same

names as the Property Get procedures prepared in the preceding section. By skipping the Property Let procedure for the ID property, you created a read-only ID property that can be retrieved but not set. We will assign a value to the member variable m\_ID in the Class\_Initialize procedure as described in Part 7 of this project.

#### SIDE BAR *Defining the Scope of Property Procedures*

You can place the `Public`, `Private`, or `Static` keyword before the name of a property procedure to define its scope. To indicate that the Property Get procedure is accessible to procedures in all modules, use the following statement format:

```
Public Property Get FirstName() As String
```

To make the Property Get procedure accessible only to other procedures in the module where it is declared, use the following statement format:

```
Private Property Get FirstName() As String
```

To preserve the Property Get procedure's local variables between procedure calls, use the following statement format:

```
Static Property Get FirstName() As String
```

If not explicitly specified using either `Public` or `Private`, property procedures are public by default. Also, if the `Static` keyword is not used, the values of local variables are not preserved between procedure calls.

---

#### **Creating the Class Methods**

Apart from properties, objects usually have one or more methods. A *method* is an action that the object can perform. Methods allow you to manipulate the data stored in a class object. Methods are created with subroutines or function procedures. To make a method available outside the class module, use the `Public` keyword in front of the sub or function definition. The CEmployee object that you create in this chapter has one method that allows you to calculate the new salary. Assume that the employee salary can be increased or decreased by a specific percentage or amount.

Let's continue with our project by writing a class method that calculates the employee salary.



### Custom Project 24.1 (Part 5) Writing Class Methods

- Type the following **CalcNewSalary** function procedure in the **CEmployee** class module, just below the property procedures:

```
' A class method will calculate Employee Salary  
' and return the result  
Public Function CalcNewSalary(choice As Integer,  
    curSalary As Currency, amount As Long) As Currency  
    Select Case choice  
        Case 1 ' by percent  
            CalcNewSalary = curSalary + ((curSalary * amount) / 100)  
        Case 2 ' by amount  
            CalcNewSalary = curSalary + amount  
    End Select  
End Function
```

The **CalcNewSalary** function defined with the **Public** keyword in a class module serves as a method for the **CEmployee** class. To calculate a new salary, a VBA procedure from outside the class module must pass three arguments: **choice**, **CurSalary**, and **amount**. The **choice** argument specifies the type of the calculation to be performed. Suppose you want to increase the employee salary by 5% or by \$5.00. The first option (Choice 1) will increase the salary by the specified percentage, and the second option (Choice 2) will add the specified amount to the current salary. The **curSalary** argument is the current salary figure for an employee, and **amount** determines the value by which the salary should be changed.

**SIDE BAR**

#### About Class Methods

- Only those methods that will be accessed from outside of the class should be declared as **Public**. All others should be declared as **Private**.
- Methods perform some operation on the data contained within the class.
- If a method needs to return a value, write a function procedure. Otherwise, create a subprocedure.

---

#### Creating an Instance of a Class

After creating all the necessary Property Get, Property Let, sub, or function procedures for your VBA database application in the class module, you are ready to create an object based on the new class you created. Creating an object is often called *creating an instance of a class* or *instantiating the object*. It is important to

understand the difference between the class and an object. A class is like a data type; it defines a type of object. It is not itself an object, but a template that you can use to create as many objects as you want. In other words, multiple objects can be created from a single class. Each object is considered a completely different entity with distinct properties. For example, you may have two procedures that handle employees. They both create an object called newEmployee based on the CEmployee class. The first procedure defines the properties of a new employee, such as Full Name, Department, and Date Of Hire, and if the employee is hired as a manager, it calls the second procedure that defines an additional set of managerial properties. Using classes in this way requires a different way of thinking, but once you become familiar with the process, you will be able to create extremely reusable and organized components for yourself and others.

Before an object can be created, an object variable must be declared in a standard module to store the reference to the object. If the name of the class module is CEmployee, then a new instance of this class can be created with the following statement:

```
Dim emp As New CEmployee
```

The `emp` variable will represent a reference to an object of the CEmployee class. When you declare the object variable with the `New` keyword, VBA creates the object and allocates memory for it. However, the object isn't instanced until you refer to it in your procedure code by assigning a value to its property or by running one of its methods.

You can also create an instance of the object by declaring an object variable with the data type defined to be the class of the object, as in the following:

```
Dim emp As CEmployee  
Set emp = New CEmployee
```

If you don't use the `New` keyword with the `Dim` statement, VBA does not allocate memory for your custom object until your procedure needs it.



### Custom Project 24.1. (Part 6) Creating an Instance of a Class

1. In Visual Basic Editor window, choose **Insert | Module** to add a standard module to your application.
2. Use the Name property in the Properties window to change the name of the new module from **Module1** to **EmpOperations**.

3. Type the following declarations at the top of the EmpOperations module:

```
' create an instance of a class
' and define a collection to store
' employee data
Dim emp As New CEmployee
Dim CEmployee As New Collection
```

The first declaration statement (`Dim`) declares the variable `emp` as a new instance of the `CEmployee` class. The second statement declares a custom collection. The `CEmployee` collection will be used to store all employee data. Recall that creating and using collections in VBA was covered in Chapter 8, where you used collections like arrays for storing multiple values. Collections can also store objects. For example, the `CEmployee` object can contain a collection of individual employee objects.

### Event Procedures in Class Modules

---

An *event* is basically an action recognized by an object. Custom classes recognize only two events: `Class_Initialize` and `Class_Terminate`. These events are triggered when an instance of the class is created and destroyed, respectively.

The `Initialize` event is generated when an object is created from a class (see the preceding section on creating an instance of a class). In the `CEmployee` class example, the `Initialize` event will also fire the first time that you use the `emp` variable in code. Because the statements included inside the `Initialize` event are the first ones to be executed for the object before any properties are set or any methods are executed, the `Initialize` event is a good place to perform initialization of the objects created from the class. As you recall, we made the `ID` read-only in the `CEmployee` class. You will use the `Initialize` event to assign a unique five-digit number to the `m_ID` variable as shown in Part 7 of this custom project.

The `Class_Initialize` event procedure uses the following syntax:

```
Private Sub Class_Initialize()
    [code to perform tasks as the object is created goes here]
End Sub
```

The `Terminate` event occurs when all references to an object have been released. This is a good place to perform any necessary cleanup tasks.

The `Class_Terminate` event procedure uses the following syntax:

```
Private Sub Class_Terminate()
    [cleanup code goes here]
End Sub
```

To release an object variable from an object, use the following syntax:

```
Set objectVariable = Nothing
```

When you set the object variable to `Nothing`, the `Terminate` event is generated. Any code placed in this event is executed then.



### Custom Project 24.1 (Part 7) Adding Class\_Initialize Procedure

1. Activate the CEmployee class module code window and choose Class from the Object drop-down at the top left side of the code module.
2. Choose Initialize from the Procedure drop-down at the top right side of the code module window.
3. Complete the `Class_Initialize` procedure as follows:

```
Private Sub Class_Initialize()
    m_ID = SetEmpID
End Sub
```

The `SetEmpID` function procedure (see Part 9, step 11) will generate the 5-digit employee ID and assign this value to the `m_ID` variable.

**NOTE**

*Using custom events in class modules requires that you include the `WithEvents` keyword when the object is defined. This is explained and demonstrated in the next chapter.*

## CREATING THE USER INTERFACE

---

Implementing our custom CEmployee object requires that you design a form that will allow you to enter and manipulate employee data.



### Custom Project 24.1 (Part 8) Designing a User Form

1. Choose **File | Close and Return to Microsoft Access**.
2. Click the **Form Design** in the Forms section of the Create tab. Access will display a blank form in the Design view.
3. Save the form as **frmEmployeeSalaries**.
4. Use the tools in the Controls section of the Form Design tab to place controls on the form as shown in Figure 24.2.

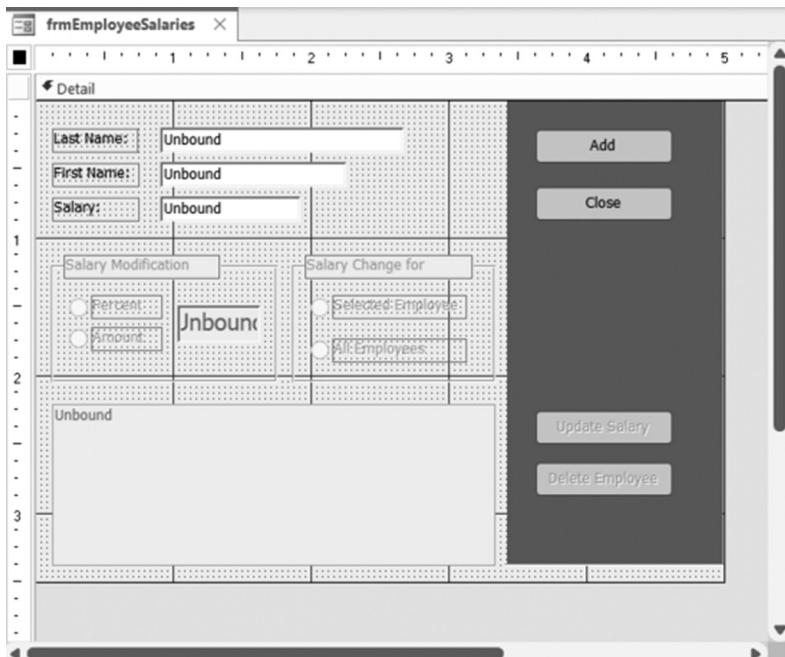


FIGURE 24.2 This form demonstrates the use of the CEmployee custom object.

5. Activate the property sheet and set the following properties for the form controls. To set the specified property, first click the control on the form to select it. Then, in the property sheet type the information shown in the Settings column next to the property indicated in the Property column.

Object	Property	Settings
Label1	Caption	Last Name
Text box next to the Last Name label	Name	txtLastName
Label2	Caption	First Name
Text box next to the First Name label	Name	txtFirstName
Label3	Caption	Salary
Text box next to the Salary label	Name	txtSalary
Option group 1	Name Caption	frSalaryMod Salary Modification

(Contd.)

Object	Property	Settings
Text box in the option group titled “Salary Modification”	Name	txtRaise
Option button 1	Name Caption	optPercent Percent
Option button 2	Name Caption	optAmount Amount
Option group 2	Name Caption	frSalaryFor Salary Change for
Option button 3	Name Caption	optSelected Selected Employee
Option button 4	Name Caption	optAll All Employees
Listbox	Name Row Source Type Column Count Column Widths	lboxPeople Value List 4 0.5"; 0.9"; 0.7"; 0.5"
Command Button 1	Name Caption	cmdAdd Add
Command Button 2	Name Caption	cmdClose Close
Command Button 3	Name Caption	cmdUpdate Update Salary
Command Button 4	Name Caption	cmdDelete Delete Employee

Now that the form is ready, you need to write a few event procedures to handle various events, such as clicking a command button, selecting option buttons and loading the form.



### Custom Project 24.1 (Part 9) Writing Event Procedures in the Form Class Module

- With the form in the Design View, activate the Code window behind the form by choosing the **View Code** button in the **Tools** section of the **Form Design** tab.
- Enter the following variable declarations at the top of the form’s Code window:

```
' variable declarations
Dim choice As Integer
Dim amount As Long
```

**NOTE**

*Please ensure that the Option Explicit statement appears at the top of the module, above the variable declaration statements.*

3. Type the following **UserForm\_Initialize** procedure to enable or disable controls on the form:

```
Private Sub UserForm_Initialize()
    txtLastName.SetFocus
    cmdUpdate.Enabled = False
    cmdDelete.Enabled = False
    lboxPeople.Enabled = False
    lboxPeople.RowSource = GetValues
    frSalaryFor.Enabled = False
    frSalaryFor.Value = 0
    frSalaryMod.Enabled = False
    frSalaryMod.Value = 0
    txtRaise.Enabled = False
    txtRaise.Value = ""
End Sub
```

4. Type the following **Form\_Load** event procedure:

```
Private Sub Form_Load()
    Call UserForm_Initialize
End Sub
```

When the form loads, the **UserForm\_Initialize** procedure will run.

5. Enter the following **cmdAdd\_Click** procedure to add the employee to the collection:

```
Private Sub cmdAdd_Click()
    Dim strLast As String
    Dim strFirst As String
    Dim curSalary As Currency

    ' Validate data entry
    If IsNull(txtLastName.Value) Or txtLastName.Value = "" Or IsNull(txtFirstName.Value) Or txtFirstName.Value = "" Or IsNull(txtSalary.Value) Or txtSalary.Value = "" Then
        MsgBox "Enter Last Name, First Name and Salary."
        txtLastName.SetFocus
        Exit Sub
    End If
    If Not IsNumeric(txtSalary) Then
        MsgBox "You must enter a value for the Salary."
        txtSalary.SetFocus
        Exit Sub
    End If
End Sub
```

```
End If
If txtSalary < 0 Then
    MsgBox "Salary cannot be a negative number."
    Exit Sub
End If

' assign text box values to variables
strLast = txtLastName
strFirst = txtFirstName
curSalary = txtSalary

' enable buttons and other controls
cmdUpdate.Enabled = True
cmdDelete.Enabled = True
lboxPeople.Enabled = True
frSalaryFor.Enabled = True
frSalaryMod.Enabled = True
txtRaise.Enabled = True
txtRaise.Value = ""
lboxPeople.Visible = True

' enter data into the CEmployees collection
EmpOperations.AddEmployee strLast, strFirst, curSalary

' update listbox
lboxPeople.RowSource = GetValues

' delete data from text boxes
txtLastName = ""
txtFirstName = ""
txtSalary = ""
txtLastName.SetFocus
End Sub
```

The cmdAdd\_Click procedure starts off by validating the user's input in the Last Name, First Name, and Salary text boxes. If the user entered correct data, the text box values are assigned to the variables `strLast`, `strFirst`, and `curSalary`. Next, several statements enable buttons and other controls on the form so that the user can work with the employee data. The following statement calls the AddEmployee procedure in the EmpOperations standard module and passes the required parameters to it:

```
EmpOperations.AddEmployee strLast, strFirst, curSalary
```

Once the employee is entered into the collection, the employee data is added to the listbox (see Figure 24.3) with the following statement:

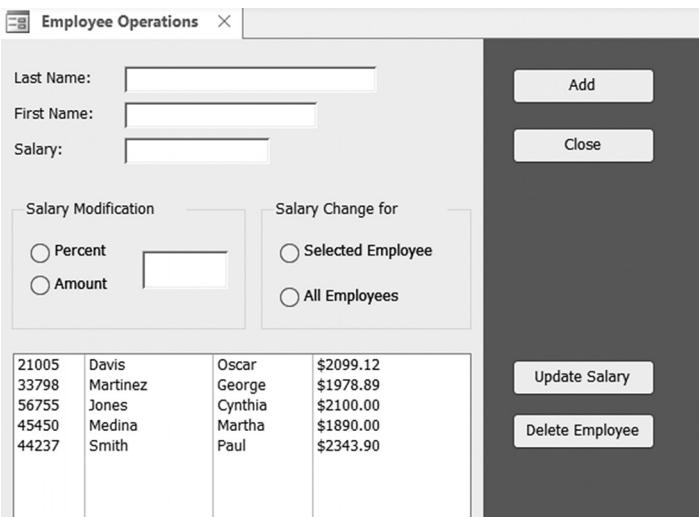
```
lboxPeople.RowSource = GetValues
```

GetValues is the name of a function procedure in the EmpOperations module (see step 12 further on). This function cycles through the CEmployee collection to create a string of values for the listbox row source.

The cmdAdd\_Click procedure ends by clearing the text boxes, and then setting the focus to the Last Name text box so the user can enter new employee data.

**6.** Enter the following **cmdClose\_Click** procedure to close the form:

```
Private Sub cmdClose_Click()
    DoCmd.Close
End Sub
```



**FIGURE 24.3** The listbox control displays employee data as entered in the custom collection CEmployee.

**7.** Write the following **Click** procedure for the cmdUpdate button:

```
Private Sub cmdUpdate_Click()
    Dim numOfPeople As Integer
    Dim colItem As Integer

    ' validate user selections
```

```
If frSalaryFor.Value = 0 Or frSalaryMod.Value = 0 Then
    MsgBox "Please choose appropriate option button in " & _
        vbCr & "the 'Salary Modification' and " & _
        "'Change the Salary for' areas.", vbOKOnly, _
        "Insufficient selection"
    Exit Sub
ElseIf Not IsNumeric(txtRaise) Or txtRaise = "" Then
    MsgBox "You must enter a number."
    txtRaise.SetFocus
    Exit Sub
ElseIf frSalaryMod.Value = 1 And _
    lboxPeople.ListIndex = -1 Then
    MsgBox "Click the employee name.", , _
        "Missing selection in the List box"
    Exit Sub
End If

If frSalaryMod.Value = 1 And lboxPeople.ListIndex = -1 Then
    MsgBox "Enter data or select an option."
    Exit Sub
End If
'get down to calculations
amount = txtRaise
colItem = lboxPeople.ListIndex + 1
If frSalaryFor.Value = 1 And frSalaryMod.Value = 1 Then
    'by percent, one employee
    choice = 1
    numOfPeople = 1
ElseIf frSalaryFor.Value = 1 And frSalaryMod.Value = 2 Then
    'by amount, one employee
    choice = 2
    numOfPeople = 1
ElseIf frSalaryFor.Value = 2 And frSalaryMod.Value = 1 Then
    'by percent, all employees
    choice = 1
    numOfPeople = 2
ElseIf frSalaryFor.Value = 2 And frSalaryMod.Value = 2 Then
    'by amount, all employees
    choice = 2
    numOfPeople = 2
End If
UpdateSalary choice, amount, numOfPeople, colItem
lboxPeople.RowSource = GetValues
End Sub
```

When the Update Salary button is clicked, the procedure checks to see whether the user selected the appropriate option buttons and entered the adjusted figure in the text box. The update can be done for the selected employee or for all the employees listed in the listbox control and collection. You can increase the salary by the specified percentage or amount (see Figure 24.4). Depending on which options are specified, values are assigned to the variables: `choice`, `amount`, `numOfpeople`, and `colItem`. These variables serve as parameters for the `UpdateSalary` procedure located in the `EmpOperations` module (see Step 13 further on). The last statement in the `cmdUpdate_Click` procedure sets the row source property of the listbox control to the result obtained from the `GetValues` function in the `EmpOperations` standard module.

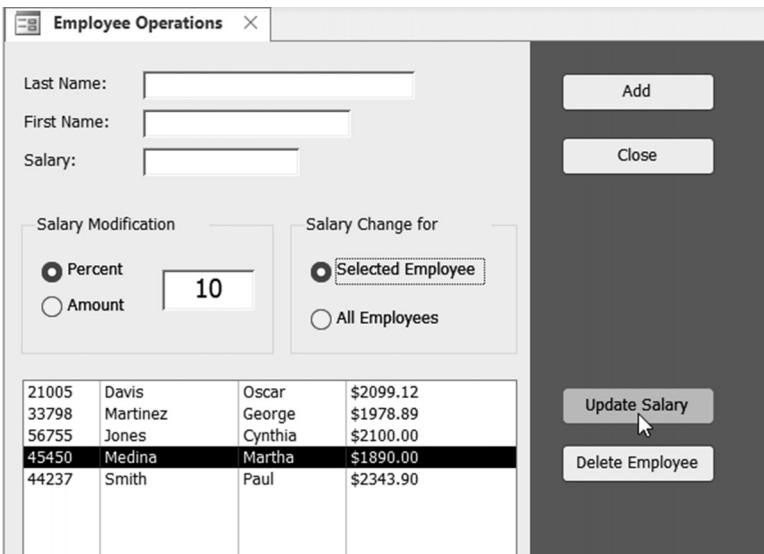


FIGURE 24.4 The employee salary can be increased or decreased by the specified percentage or amount.

## 8. Enter the following `cmdDelete_Click` procedure:

```
Private Sub cmdDelete_Click()
    ' make sure an employee row is highlighted
    ' in the listbox control
    If lboxPeople.ListIndex > -1 Then
        DeleteEmployee lboxPeople.ListIndex + 1
        If lboxPeople.ListCount = 1 Then
            lboxPeople.RowSource = GetValues
```

```

UserForm_Initialize
Else
    lboxPeople.RowSource = GetValues
End If
Else
    MsgBox "Click the item you want to remove."
End If
End Sub

```

The cmdDelete\_Click procedure lets you remove an employee from the custom collection CEmployee. If you click an item in the listbox and then click the Delete Employee button, the DeleteEmployee procedure is called. This procedure requires an argument that specifies the index number of the item selected in the listbox. After the employee is removed from the collection, the row source of the listbox control is reset to display the remaining employees. When the last employee is removed from the collection, the UserForm\_Initialize procedure is called to tackle the task of disabling controls that cannot be used until at least one employee is entered into the CEmployee collection.

9. To activate the **EmpOperations** module that you created earlier, double-click its name in the Project Explorer window. The top of the module should contain the following declaration lines, the first two automatically added by Access:

```

Option Compare Database
Option Explicit

Dim emp As New CEmployee
Dim CEmployee As New Collection

```

10. In the **EmpOperations** standard module, enter the following **AddEmployee** procedure:

```

Sub AddEmployee(empLast As String, empFirst As String, _
    empSalary As Currency)
    With emp
        .LastName = empLast
        .FirstName = empFirst
        .Salary = CCur(empSalary)
        If .Salary = 0 Then Exit Sub
        CEmployee.Add emp
    End With
End Sub

```

The AddEmployee procedure is called from the cmdAdd\_Click procedure attached to the form's Add button. This procedure takes three arguments. When VBA reaches the `With emp` construct, a new instance of the CEmployee

class is created. The LastName, FirstName, and Salary properties are set with the values passed from the cmdAdd\_Click procedure. The ID property is set with the number generated by the result of the SetEmpId function (see the following step). Each time VBA sees the reference to the instanced `emp` object, it will call upon the appropriate Property Let procedure located in the class module. (The next section of this chapter demonstrates how to walk through this procedure step by step to see exactly when the Property procedures are executed.) The last statement inside the `With emp` construct adds the user-defined object `emp` to the custom collection called `CEmployee`.

11. In the **EmpOperations** standard module, enter the following **SetEmpID** function procedure:

```
Function SetEmpID() As String
    Dim ref As String

    Randomize
    ref = Int((99999 - 10000) * Rnd + 10000)
    SetEmpId = ref
End Function
```

This function will assign a unique five-digit number to each new employee. To generate a random integer between two given integers where `ending_number = 99999` and `beginning_number = 10000`, the following formula is used:

```
= Int((ending_number - beginning_number) * Rnd + beginning_number)
```

The `SetEmpID` function procedure also uses the `Randomize` statement to reinitialize the random number generator. For more information on using the `Rnd` and `Integer` functions, as well as the `Randomize` statement, refer to the online help.

Note: We will call this function from the `Class_Initialize` event procedure to assign the value to the `m_ID` variable that we made read-only by not writing the Property Let for the `ID` property.

12. Enter the following **GetValues** function procedure.

```
Function GetValues()
    Dim myList As String

    myList = ""
    For Each emp In CEmployee
        myList = myList & emp.ID & ";" & _
            emp.LastName & ";" & _
            emp.FirstName & ";" & $" & _
```

```

        Format(emp.Salary, "0.00") & ";"

    Next emp
    GetValues = myList
End Function

```

The GetValues function, which is called from the cmdAdd\_Click, cmdUpdate\_Click, and cmdDelete\_Click procedures, provides the values for the listbox control to synchronize it with the current values in the CEmployee collection.

### 13. Enter the following **UpdateSalary** procedure:

```

Sub UpdateSalary(choice As Integer, myValue As Long, _
peopleCount As Integer, colItem As Integer)
    Set emp = New CEmployee

    If choice = 1 And peopleCount = 1 Then
        CEmployee.Item(colItem).Salary = _
            emp.CalcNewSalary(1, CEmployee.Item( _
                colItem).Salary, myValue)
    ElseIf choice = 1 And peopleCount = 2 Then
        For Each emp In CEmployee
            emp.Salary = emp.Salary + ((emp.Salary * myValue) _
                / 100)
        Next emp
    ElseIf choice = 2 And peopleCount = 1 Then
        CEmployee.Item(colItem).Salary = _
            CEmployee.Item(colItem).Salary + myValue
    ElseIf choice = 2 And peopleCount = 2 Then
        For Each emp In CEmployee
            emp.Salary = emp.Salary + myValue
        Next emp
    Else
        MsgBox "Enter data or select an option."
    End If
End Sub

```

The UpdateSalary procedure is called from the cmdUpdate\_Click procedure, which is assigned to the Update Salary button on the form. The click procedure passes four parameters that the UpdateSalary procedure uses for the salary calculations. When a salary for the selected employee needs to be updated by a percentage or amount, the CalcNewSalary method residing in the class module is called. For modification of salary figures for all the employees, we iterate over the CEmployee collection to obtain the value of the Salary property of each `emp` object, and then perform the required calculation by using a formula.

By entering a negative number in the form's txtRaise text box, you can decrease the salary by the specified percentage or amount.

**14. Enter the DeleteEmployee procedure:**

```
Sub DeleteEmployee(colItem As Integer)
    Dim getcount As Integer

    CEmployee.Remove colItem
End Sub
```

The DeleteEmployee procedure uses the `Remove` method to delete the selected employee from the `CEmployee` custom collection. Recall that the `Remove` method requires one argument, which is the position of the item in the collection. The value of this argument is obtained from the `cmdDelete_Click` procedure. The class module procedures were called from the standard module named `EmpOperations`. This was done to avoid creating a new instance of a user-defined class every time we needed to call it.

## **RUNNING THE CUSTOM APPLICATION**

---

Now that you have finished writing the necessary VBA code, let's load `frmEmployeeSalaries` to enter and modify employee information.



### **Custom Project 24.1 (Part 10) Running the Custom Project**

- 1. Choose File | Save Chap24** to save all the objects in the VBA project.
- 2. Switch to the Access application window and activate `frmEmployeeSalaries` in the Form view.**
- 3. Enter the employee last and first name and salary and click the **Add** button.**  
The employee information now appears in the listbox. Notice that an employee ID is automatically entered in the first column. All the disabled form controls are now enabled.
- 4. Enter data for another employee, and then click the **Add** button.**
- 5. Enter information for at least three more people.**
- 6. Increase the salary of the third employee in the listbox by 10%.** To do this, click the employee's name in the listbox, click the **Percent** option button, and type **10** in the text box in the Salary Modification section of the form. In the Change the Salary for section of the form, click the **Selected Employee** option button. Finally, click the **Update Salary** button to perform the update operation.

7. Now increase the salary of all the employees by \$5.
8. Remove the fourth employee from the listbox. To do this, select the employee in the listbox and click the **Delete Employee** button.
9. Close the **frmEmployeeSalaries** by clicking the **Close** button.

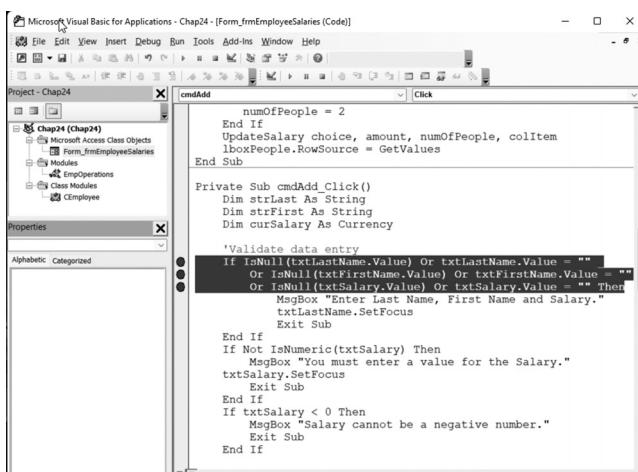
## **WATCHING THE EXECUTION OF YOUR CUSTOM OBJECT**

To help you understand what's going on when your code runs and how the custom object works, let's walk through the **cmdAdd\_Click** procedure. Treat this exercise as a refresher to the debugging techniques that were covered in detail in Chapter 9.

### **Custom Project 24.1 (Part 11) Custom Project Code Walkthrough**

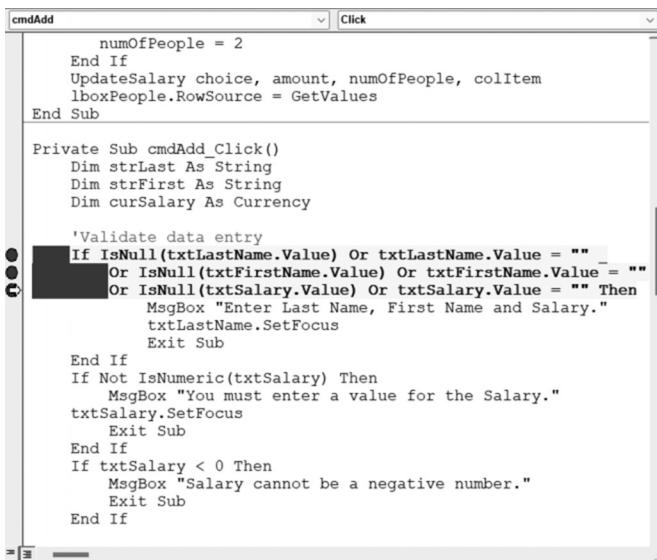
1. Open the **frmEmployeeSalaries** form in Design view and click **View Code** in the Tools section of the Design tab.
2. Select **cmdAdd** from the combo box at the top left of the Code window.
3. Set a breakpoint by clicking in the left margin next to the following line of code, as shown in Figure 24.5:

```
If IsNull(txtLastName.Value) Or txtLastName.Value = ""  
Or IsNull(txtFirstName.Value) Or txtFirstName.Value = "" _  
Or IsNull(txtSalary.Value) Or txtSalary.Value = "" Then
```



**FIGURE 24.5** A red circle in the margin indicates a breakpoint. The statement with a breakpoint is displayed as white text on a red background.

4. Press **Alt+F11** to return to the form **frmEmployeeSalaries**, and then switch to the Form view.
5. Enter data in the Last Name, First Name, and Salary text boxes, and then click the form's **Add** button. Visual Basic should now switch to the Code window because it came across the breakpoint in the first line of the **cmdAdd\_Click** procedure (see Figure 24.6).
6. Step through the code one statement at a time by pressing **F8** or click **Step Into (F8)** button on the Debug toolbar. Visual Basic runs the current statement, then automatically advances to the next statement and suspends execution. The current statement is indicated by a yellow arrow in the margin and a yellow background. Keep pressing **F8** to execute the procedure step by step. After Visual Basic switches to the **EmpOperations** module to run the **AddEmployee** procedure and encounters the **With emp** statement, it will jump to the **Class\_Initialize** procedure in the **CEmployee** class module and execute the **SetEmpID** function to generate and assign the value of the employee ID to the **m\_ID** variable. Next, it will go on to set other properties of the **emp** object, executing the corresponding Property Let procedures in the **CEmployee** class module (see Figure 24.7).



```
cmdAdd    Click

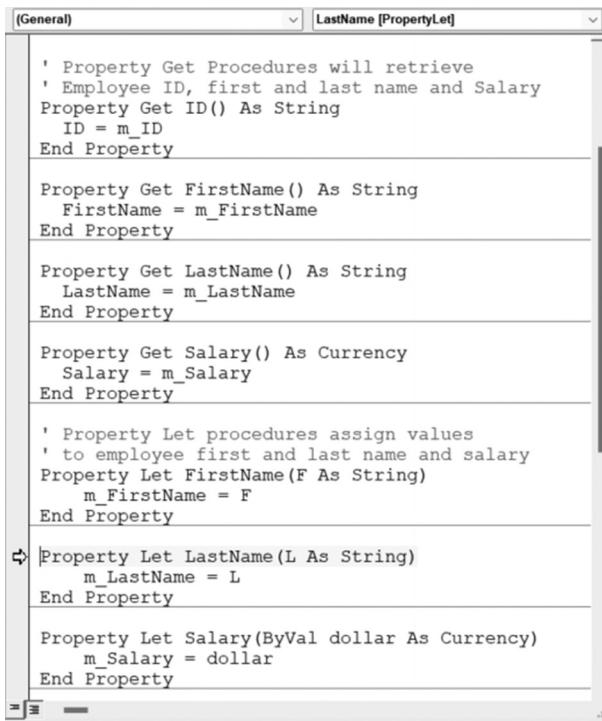
    numOfPeople = 2
End If
UpdateSalary choice, amount, numOfPeople, colItem
lboxPeople.RowSource = GetValues
End Sub

Private Sub cmdAdd_Click()
Dim strLast As String
Dim strFirst As String
Dim curSalary As Currency

'Validate data entry
If IsNull(txtLastName.Value) Or txtLastName.Value = "" Then
    MsgBox "Enter Last Name, First Name and Salary."
    txtLastName.SetFocus
    Exit Sub
End If
If Not IsNumeric(txtSalary) Then
    MsgBox "You must enter a value for the Salary."
    txtSalary.SetFocus
    Exit Sub
End If
If txtSalary < 0 Then
    MsgBox "Salary cannot be a negative number."
    Exit Sub
End If
```

FIGURE 24.6 When Visual Basic encounters a breakpoint while running a procedure, it switches to the Code window and displays a yellow arrow in the margin to the left of the statement at which the procedure is suspended.

- Using the **F8** key, continue executing the cmdAdd\_Click procedure code to the end. When VBA encounters the end of the procedure (`End Sub`), the yellow highlighter will be turned off. At that time, press **F5** to finish execution of the remaining code. Next, switch back to the active form by pressing **Alt+F11**.



```

(General) [LastName [PropertyLet]
' Property Get Procedures will retrieve
' Employee ID, first and last name and Salary
Property Get ID() As String
    ID = m_ID
End Property

Property Get FirstName() As String
    FirstName = m_FirstName
End Property

Property Get LastName() As String
    LastName = m_LastName
End Property

Property Get Salary() As Currency
    Salary = m_Salary
End Property

' Property Let procedures assign values
' to employee first and last name and salary
Property Let FirstName(F As String)
    m_FirstName = F
End Property

Property Let LastName(L As String)
    m_LastName = L
End Property

Property Let Salary(ByVal dollar As Currency)
    m_Salary = dollar
End Property

```

**FIGURE 24.7** Setting the properties of your custom object is accomplished through the Property Let procedures.

- Enter data for a new employee, and then click the **Add** button. When Visual Basic displays the Code window, choose **Debug | Clear All Breakpoints**. Now press **F5** to run the remaining code without stepping through it.
- In the Visual Basic Editor window, choose **File | Save Chap24**, and then save changes to the modules when prompted.
- Choose **File | Close and Return to Microsoft Access**.
- Close the **Chap024.accdb** database.

## CREATING AND WORKING WITH COLLECTION CLASSES

---

In Chapter 8 of this book, you used collections to track and manipulate data in your VBA program. At that time, you learned that collections complement the capabilities of arrays, allowing you to store and process large amounts of data without the need to create multiple variables. To track data, you declared the collection at the top of the standard module using the Dim keyword:

```
Dim myCollection as New Collection
```

Then you used the Add method to add specific items to your collection:

```
myCollection.Add "item1"  
myCollection.Add "item2"  
myCollection.Add "item3"
```

You also learned how to add elements in any position by using the optional Before and After arguments. You counted your collection items using the Count property and removed your items using the Remove method. Now that you are familiar with the concept of the collection, let's take it a bit further.

### The Collection Object

---

Collections are often created to hold objects and they play an important role in the object models of Microsoft 365 applications (Access, Excel, Word, PowerPoint, Outlook, and OneNote) as well as other applications that support VBA. You've already used many of the built-in collections in Access. For example, recall the TableDefs or QueryDefs collections that you worked with in earlier chapters of this book. In Word, the Documents collection stores all open documents, and the Paragraphs collection stores each paragraph in the current document; in Excel, there is a collection of Workbooks, Worksheets, and so on. One can manipulate these collections using the familiar For Each...Next loop. Each collection is a class that can contain instances of other classes or entire groups (collections) of related classes.

The VBA collection objects have the familiar Add, Remove, and Items methods, and the Count property. The Add method adds a member to the collection and has the following syntax:

```
CollectionObjectName.Add(Item, [Key], [Before]. [After])
```

Notice that the parameters in brackets are optional. The Key is a unique string that specifies a key that can be used to identify the collection member instead of its ordinal position in the collection. The Before and After arguments specify

that the new member is to be added at the position immediately preceding an existing member in the collection or immediately following the existing member in the collection. You can only specify a `Before` or `After` position, but not both.

The only required parameter of the `Add` method is `Item`, which is a pointer to the object instance. Do not confuse the `Item` parameter with the `Item` method that is used to specify an object by its ordinal position in the collection:

```
CollectionObjectName.Item(Index)
```

The `Index` argument specifies the position of an existing collection member. The index must be between 1 and the collection's `Count` property. Unlike the built-in Access collections, which are zero-based, the first item in the custom collection object is at position 1. The `Index` argument is also used to provide the position of the existing collection member when removing it from a collection:

```
CollectionObjectName.Remove(Index)
```

The read-only `Count` property returns a `Long Integer` containing the number of objects in a collection:

```
Debug.Print CollectionObjectName.Count
```

## The Collection Class

---

So far, you have used untyped collections. These collections allowed you to put anything in them and were easy to create. But what if you're writing an application that must restrict the type of item that can be added to the collection? In this case, a strongly-typed collection class is what you need. Strongly-typed collections are custom collection classes that will only accept objects of the same type. While they are not as easy to create as untyped collections that are built into VBA, custom collection classes give you more control over what is added to the collection and allow you to catch errors early in the development process instead of at runtime. You create a collection class in a class module.

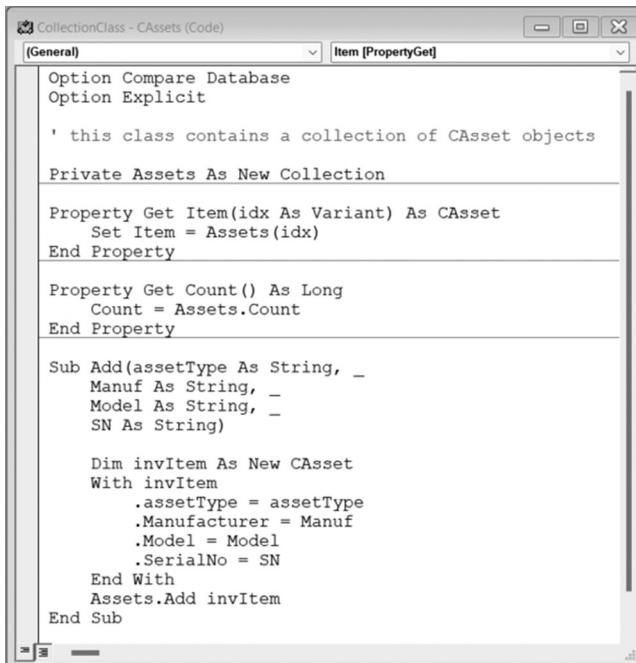
To get started with creating and using collection classes in Access, assume that you are writing an application that keeps tracks of a computer inventory in your company. In the VBE Editor screen, you've added two class modules and a standard module. The first Class Module named `CAsset` contains the several public variable declarations for your asset object and one `Property Get AssetInfo` procedure that is used to retrieve asset information:

```
' class module: CAsset
Option Compare Database
Option Explicit
```

```
' Declare asset variables
Public assetType As String
Public Manufacturer As String
Public Model As String
Public SerialNo As String

Property Get AssetInfo() As String
    AssetInfo = assetType & _
        " | " & Manufacturer & _
        " | " & Model & _
        " | " & SerialNo & vbCrLf
End Property
```

The second Class Module is named CAssets and contains the code shown in Figure 24.8.



The screenshot shows the Microsoft Visual Basic Editor with the title bar 'CollectionClass - CAssets (Code)'. The code is displayed in the 'General' tab under the 'Item [PropertyGet]' section. The code defines a collection of CAsset objects and includes methods for retrieving items by index, getting the count, and adding new assets.

```
Option Compare Database
Option Explicit

' this class contains a collection of CAsset objects
Private Assets As New Collection

Property Get Item(idx As Variant) As CAsset
    Set Item = Assets(idx)
End Property

Property Get Count() As Long
    Count = Assets.Count
End Property

Sub Add(assetType As String, _
        Manuf As String, _
        Model As String, _
        SN As String)
    Dim invItem As New CAsset
    With invItem
        .assetType = assetType
        .Manufacturer = Manuf
        .Model = Model
        .SerialNo = SN
    End With
    Assets.Add invItem
End Sub
```

FIGURE 24.8 Class Collection Module named CAssets.

Notice that the collection object variable is named `Assets` and is declared at the top of the `CAssets` module. The first `Property Get Item` allows you to retrieve asset information for a specific asset object while the second `Property Get Count` returns the total number of items placed in the collection. The cus-

tom `Add` method is used to create an inventory item and add it to the collection. The `invItem` object variable represents an asset object created from the `CAsset` class.

To work with the Asset objects and the Assets collection, we need a procedure in a standard module. This procedure should call the `Add` method from the `CAssets` class module to create an asset and add it to the Assets collection. The following code is to be entered in a standard module:

```
Option Compare Database
Option Explicit

Sub AddItemsToInventory()
    ' create a new collection based on CAassets class
    ' to store computer assets
    Dim Assets As New CAassets

    'add assets to the collection
    Assets.Add "Laptop", "Dell", "XPS-13", "XYZ89JWZ28"
    Assets.Add "Desktop", "Lenovo", "ThinkCentre M90q Gen 2 Tiny", _
               "YGU23HY7899"

    'list all inventory items
    Dim invAsset As CAsset
    Dim invItemNo As Integer

    For invItemNo = 1 To Assets.Count
        Debug.Print Assets.Item(invItemNo).AssetInfo
    Next invItemNo
End Sub
```

For this demo procedure, we've hardcoded the asset data. You can create a nice Access form to collect asset information from the user or read it directly from a text file. The `AddItemsToInventory` procedure declares the `Assets` collection class and calls the `Add` method from the `CAssets` class module to create a new asset and add it to inventory. Next, the `For... Loop` is used to iterate through the collection and retrieve `AssetInfo` for each stored asset. To better understand how an asset is created, added, and retrieved from the collection, run the `AddItemsToInventory` procedure using the debugging techniques, as demonstrated in the previous Hands-On project.

After running the procedure, the collection content is shown in the Immediate Window:

Laptop | Dell | XPS-13 | XYZ89JWZ28

Desktop | Lenovo | ThinkCentre M90q Gen 2 Tiny | YGU23HY7899

The same output can be obtained using another procedure that utilizes the built-in VBA collection instead of the custom collection class:

```
Sub CreateNewAssets_BuiltInCollection()
    ' declare an untyped built-in VBA collection object
    Dim colAssets As New Collection

    'create two assets
    Dim oAsset1 As New CAsset
    Dim oAsset2 As New CAsset

    ' assign values to asset properties
    With oAsset1
        .assetType = "Laptop"
        .Manufacturer = "Dell"
        .Model = "XPS-13"
        .SerialNo = "XYZ89JWZ28"
    End With

    With oAsset2
        .assetType = "Desktop"
        .Manufacturer = "Lenovo"
        .Model = "ThinkCentre M90q Gen 2 Tiny"
        .SerialNo = "YGU23HY7899"
    End With

    ' Print asset information to the Immediate Window
    Debug.Print oAsset1.AssetInfo & vbCrLf & oAsset2.AssetInfo

    ' add both assets into a untyped (built-in VBA) collection
    colAssets.Add oAsset1
    colAssets.Add oAsset2

    ' use the for each loop to loop over the collection
    Dim myAsset As CAsset
    For Each myAsset In colAssets
        Debug.Print myAsset.AssetInfo
    Next
End Sub
```

In the code above, all seems to work fine until you happen to add something else to the collection that is not an object:

```
' add both assets into a untyped (built-in VBA) collection
colAssets.Add oAsset1
```

```
colAssets.Add oAsset2  
colAssets.Add CurrentDb.Name
```

The third code line in the snippet above will generate an error at runtime but not at compile time. What does this mean? When you write your code and choose Debug | Compile, Access checks for syntax-errors. If no errors are found, you assume that you've written perfect code and things will go smoothly, but then you are faced with Run-time error: "424 Object Required." Don't you just hate this error? The program fails while looping through the collection, expecting an object and not a string. By using a custom collection class as we did in the previous example, you can expose errors at compile time so you can handle your errors while you're still in the development phase.

**NOTE**

*All code used in the CAassets Collection example can be found in the CollectionClass.accdb database in the Companion files.*

---

## SUMMARY

---

In this chapter, you learned how to create and use your own objects and collections in VBA procedures. Although you can easily build Access database applications without using your own classes, understanding the essentials of object-oriented programming can help you manage more advanced database projects.

You learned here that class is a template that you can use to create one or more objects. A class consists of the following:

1. Properties that contain information about it
2. Methods that are used to take some actions and process some information
3. Events that notify program of some changes

You used a class module to create a user-defined (custom) object. You saw how to define your custom object's properties using the Property Get and Property Let procedures. You also learned how to write a method for your custom object and saw how to make the class module available to the user with a custom form. Finally, you learned how to analyze the class code you wrote by stepping through it. You finished this chapter by learning a bit about creating and using Collection Classes in Access.

This chapter barely scratched the surface of what's possible with custom classes and collections. To thoroughly understand and utilize these new concepts in your Access programming journey, it will take time and practice. In the next chapter, you continue working with the class module and learning about events.

# Chapter 25 *ADVANCED EVENT PROGRAMMING*

**S**o far in this book you've worked with event procedures that executed from the form or report class module when a certain event occurred for a form, report, or control. You have probably noticed that event programming, as you've seen it implemented in the form and report class modules, requires that you copy and paste your existing event code into new form or report events in order to obtain exactly the same functionality. For instance, say you added certain features to a text box on one form and now you'd like to have a text box on other forms behave in the same way. You could react to the text box's events in the same way on all your forms by entering the same event procedure code in a form class module for each form, or you could save keystrokes by learning how to centralize and reuse your event code.

You can avoid typing the same event procedure code again and again by using classes. Recall that we've already used classes in the previous chapter of this book where you learned how you can design your own objects in VBA by writing code in a standalone class module. You worked with property procedures that allowed data to be read or written to the object. You also learned how to create functions in a class module that worked as object methods. In this chapter, you learn how to react to an object's events from a standalone class module.

Before we get started, let's review some of the VBA terms you will need to understand.

- **Event sink**—A class that implements an event. Only classes can sink events.
- **Event source**—An object that raises events. An event source can have multiple event sinks. Note that source and sink terminology is derived from electronics. A device that outputs current when active is said to be sourcing current. A device that draws current into it when active is said to be sinking current.
- **WithEvents**—A keyword that allows you to handle an object's events inside classes other than form or report classes. The variable that you declare for the `WithEvents` keyword is used to handle an object's events.
- **Event**—A statement used to declare a user-defined event. The `Event` declaration must appear in a class module.
- **RaiseEvent**—A statement used to call a custom event. The custom event must first be declared using the `Event` statement.

## SINKING EVENTS IN STANDALONE CLASS MODULES

---

Instead of writing your event procedures in the form and report class modules, you can make the maintenance of your Microsoft Access applications much simpler by writing the event code in standalone class modules.

A *standalone class module* is a special type of class module that is not associated with any form or report. This class module can be inserted in the Visual Basic Editor window by choosing *Insert | Class Module*. In addition to creating custom objects (see examples in Chapter 24), standalone class modules can implement object events.

The process of listening to an object's events is called *sinking the event*. To sink (handle) events in a standalone class module, you must use the `WithEvents` keyword. This keyword will tell the class that you want to sink some or all of the object's events in the class module. You determine which events you want to sink by writing appropriate event code (see Custom Project 25.1). Only classes can sink events. Therefore, the `WithEvents` keyword can only be used in classes. You can use the `WithEvents` keyword to declare as many individual variables as you need; however, you cannot create arrays using `WithEvents`.

An object that generates events is called an *event source*. The process of broadcasting an event is called *sourcing the event*. To handle events raised by an event source, you must declare an object variable using the `WithEvents`

keyword. For example, to react to form events in a standalone class module, you would need to enter the following module-level variable declaration:

```
Private WithEvents m_frm As Access.Form
```

In this statement, `m_frm` is the name of the object variable that references the `Form` object. While you can use any variable name you want, this variable cannot be a generic object type. That means you cannot declare it as `Object`. If the variable were declared as `Object`, Visual Basic wouldn't know what type library should be used. Therefore, it would not be able to provide you with the names of events for which you can write code.

Now, let's walk through these new concepts step by step. Custom Project 25.1 demonstrates how to create a record logger class that handles a form's `AfterUpdate` event. Each time the `AfterUpdate` event occurs, this class will enter information about the newly created record into a text file.

**NOTE**

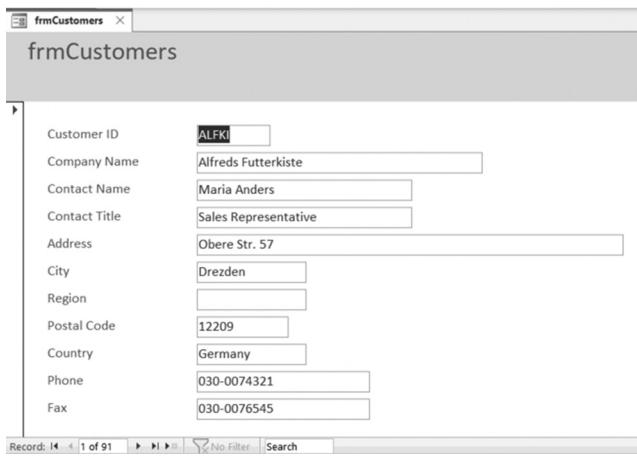
*All code files and figures for the hands-on projects may be found in the companion files.*



## Custom Project 25.1 Sinking Events in a Standalone Class Module

### **Part 1: Database File Preparation**

1. Start Access and create a new database named **Chap25.accdb** in your **C:\VBAAccess2021\_ByExample** folder.
2. Import the Customers, Products, Suppliers, and Categories tables from the sample Northwind.mdb database. To do this, in the Access window, choose **External Data | New Data Source | From Database | Access**. In the File name box, type **C:\VBAAccess2021\_ByExample\Northwind.mdb** and click **OK**. In the Import Objects window, on the **Tables** tab, select the **Customers, Products, Suppliers, and Categories** tables and click **OK** to begin importing. Click the **Close** button when done.
3. In the Navigation pane of the Access application window, select the **Customers** table and choose **Create | Form Wizard** to create a new form based on the Customers table. Select all the fields from the Customers table, choose **Columnar** layout, and specify **frmCustomers** as the form's title. After you click **Finish**, the newly designed frmCustomers form will appear in the Form view as shown in Figure 25.1.



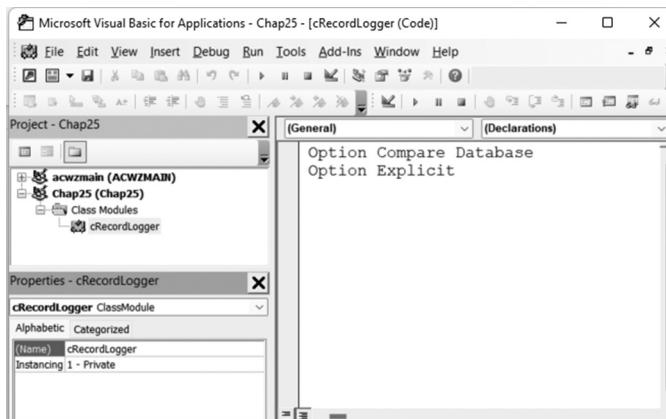
**FIGURE 25.1.** The frmCustomers form is used in Custom Project 25.1 to demonstrate how an object's event can be handled outside of the form class module.

4. Close the frmCustomers form created in Step 3.

### ***Part 2: Creating the cRecordLogger Class***

---

1. Press **Alt+F11** to activate the Visual Basic Editor window.
2. In the Project Explorer window, select **Chap25 (Chap25)** and choose **Insert | Class Module**. A new class called **Class1** will appear in the Project Explorer window. Use the **Name** property in the Properties window to change the name of the class to **cRecordLogger** (see Figure 25.2).



**FIGURE 25.2.** Use the **Name** property in the Properties window to change the name of the class module from the default **Module1**.

3. In the cRecordLogger class module's Code window, enter the following module-level variable declaration just below the Option Compare Database and Option Explicit statements:

```
Private WithEvents m_frm As Access.Form
```

After declaring the object variable using WithEvents, the variable name m\_frm appears in the Object box in your class module (see Figure 25.3). When you select this variable from the drop-down list, the valid events for that object will appear in the Procedure box (see Figure 25.4). By choosing an event from the Procedure drop-down list, an empty procedure stub will be added to the class module where you can write your code for handling the selected event. By default, Access adds the Load event procedure stub after an object is selected from the Object drop-down list.

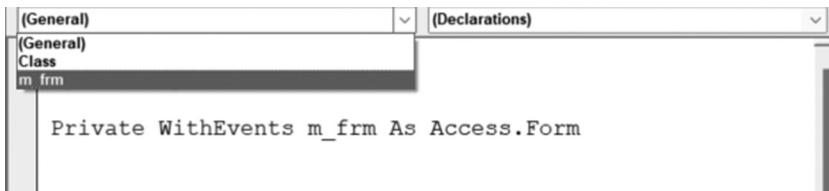


FIGURE 25.3. The Object drop-down list in the cRecordLogger's Code window lists the m\_frm object variable that was declared using the WithEvents keyword.

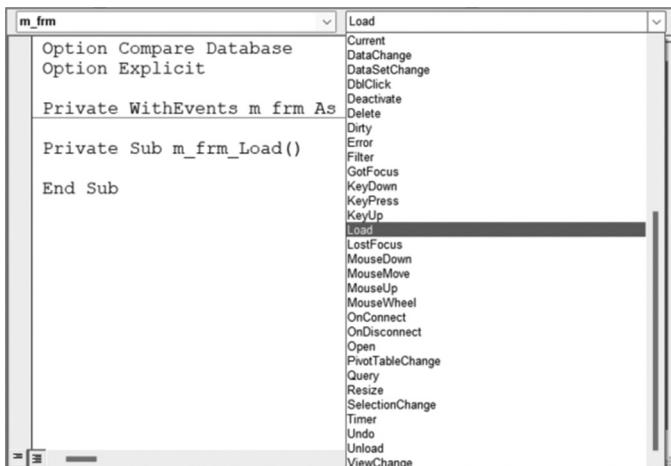


FIGURE 25.4. The Procedure drop-down list in the cRecordLogger's Code window lists the valid events for the object declared with the WithEvents keyword.

4. In the cRecordLogger class module's Code window, enter the following Property procedure just below the variable declaration:

```
Public Property Set Form(cur_frm As Form)
    Set m_frm = cur_frm
    m_frm.AfterUpdate = "[Event Procedure]"
End Property
```

To sink events in a standalone class module, you must tell the class which specific form's events the class should be responding to. You do this by writing the Property Set procedure. Recall from Chapter 24 that Property Set procedures are used to assign a reference to an object. Therefore, the statement:

```
Set m_frm = cur_frm
```

will assign the current form (passed in the `cur_frm` variable) to the `m_frm` object variable declared in step 3. Pointing the object variable (`m_frm`) at the object (`cur_frm`) isn't enough. Access will not raise the event unless the object's Event property is set to [Event Procedure]. Therefore, the second statement in the preceding procedure:

```
m_frm.AfterUpdate = "[Event Procedure]"
```

will ensure that Access knows that it must raise the form's `AfterUpdate` event. Choose **Tools | References** and add the reference to the **Microsoft Scripting Runtime Library**. You will need this library to gain access to the `FileSystemObject` in the next step. Close the References dialog box after setting the specified reference.

5. In the cRecordLogger class module's Code window, enter the following **m\_frm\_AfterUpdate** event procedure:

```
Private Sub m_frm_AfterUpdate()
    Dim fso As FileSystemObject
    Dim myFile As Object
    Dim strFileN As String
    Dim ctrl As Control

    On Error Resume Next

    Set fso = New FileSystemObject
    strFileN = "C:\VBAAccess2021_ByExample\MyCust.txt"
    Set myFile = fso.GetFile(strFileN)

    If Err.Number = 0 Then
        ' open text file
```

```
Set myFile = fso.OpenTextFile(strFileN, 8)
Else
    ' create a text file
    Set myFile = fso.CreateTextFile(strFileN)
End If

For Each ctrl In m_frm.Controls
    If ctrl.ControlType = acTextBox And _
        InStr(1, ctrl.Name, "ID") Then
        myFile.WriteLine "ID:" & ctrl.Value & _
            " Created on: " & Date & " " & Time & _
            " (Form: " & m_frm.Name & ")"
        MsgBox "See the audit trail in " & strFileN & "."
        Exit For
    End If
Next
myFile.Close
Set fso = Nothing
End Sub
```

The code inside the `m_frm_AfterUpdate` event procedure will be executed after Access finds that the form's `AfterUpdate` property is set to [Event Procedure]. This code tells Access to open or create a text file named `MyCust.txt` and write a line consisting of the value of the ID control on the form, the date and time the record was inserted or modified, and the name of the form. Notice how the `InStr` function is used to locate the control whose name contains the "ID" string. The first argument of the `InStr` function determines the character position where the search should begin, the second argument is the string being searched, and the third argument is the string expression being sought within the string specified in the second argument.

6. Save the code that you wrote in the class module by clicking the **Save** button on the toolbar or choosing **File | Save**. When the Save As dialog box appears with `cRecordLogger` in the text box, click **OK**.

For the events to actually fire now that you've written the code to handle the event in the standalone class module, you need to instantiate the class and pass it the object whose events you want to track. This requires that you write a couple of lines of code in your form's class module.

---

### ***Part 3: Creating an Instance of the Custom Class in the Form's Class Module***

1. Switch to the Access window by pressing **Alt+F11**. In the Navigation pane, right-click the **frmCustomers** form you created in step 3 of Part 1 and select **Design View**.

2. Click the Property Sheet button on the Ribbon and select **Form** in the drop-down list, then, activate the **Event** tab. Click next to the **On Open** property and select **[Event Procedure]** then click the **Build (...)** button.

Access activates the Code window and writes the procedure stub for the **Form\_Open** event. Complete the code of this event procedure and type in the **Form\_Close** event procedure, as shown here:

```
Private clsRecordLogger As cRecordLogger

Private Sub Form_Open(Cancel As Integer)
    Set clsRecordLogger = New cRecordLogger
    Set clsRecordLogger.Form = Me
End Sub

Private Sub Form_Close()
    Set clsRecordLogger = Nothing
End Sub
```

To instantiate a custom class module, we begin by declaring a module-level object variable, `clsRecordLogger`, as the name of our custom class, `cRecordLogger`. You can choose any name you wish for your variable name. Next, we instantiate the class in the `Form_Open` event procedure by using the following `Set` statement:

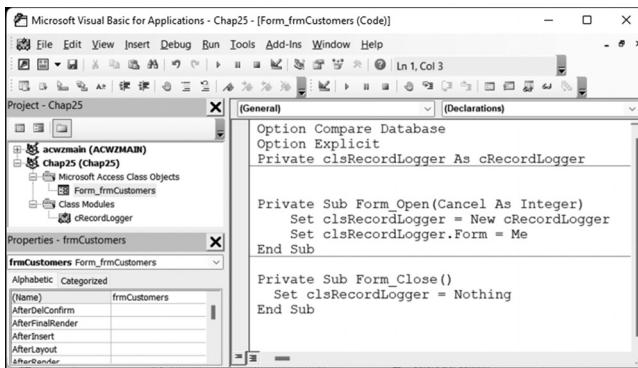
```
Set clsRecordLogger = New cRecordLogger
```

Notice that you must use the `New` keyword to create a new object of a particular class. By setting the reference to an actual instance of the object when the form first opens, we ensure that the object refers to an actual object by the time the event is first fired. The second statement in the `Form_Open` event procedure:

```
Set clsRecordLogger.Form = Me
```

sets the `Form` property defined by the `Property` procedure in the class module (see step 4 of Part 2) to the `Form` object whose events we want to sink. The `Me` keyword represents the current instance of the `Form` class.

When you are done pointing the object variable to the instance of the custom class, it is a good idea to release the variable reference. We've done this by setting the object variable `clsRecordLogger` to `Nothing` in the `Form_Close` event procedure. The complete code entered in the `frmCustomers` form class module is shown in Figure 25.5.

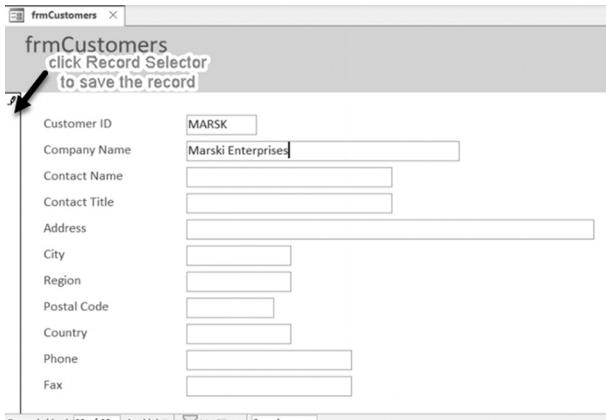


**FIGURE 25.5** To sink form events in a custom class module, you must enter some code in the form class module.

3. Press **Ctrl+S** to save the code you entered in step 2 or click the **Save** button on the toolbar.
  4. In the Access main window, close the frmCustomers form.  
Now that the required code has been written in the standalone class module and in the form class module, it's time to test our project.

## **Part 4: Testing the *cRecordLogger* Custom Class**

1. Open the **frmCustomers** form in Form view.
  2. In the **Records** group of the **Home** tab, click **New** to add a new record.
  3. Enter **MARSK** in the Customer ID text box and **Marski Enterprises** in the Company Name text box (see Figure 25.6).
  4. Press the record selector on the left side of the form to save the record.



**FIGURE 25.6.** The frmCustomers form in the Data Entry mode is used for testing out the custom cRecordLogger class.

When you save the newly entered record, a message box appears with the text “See the audit trail in C:\VBAAccess2021\_ByExample\MyCust.txt.” Recall that this message was programmed inside the `m_frm_AfterUpdate()` event procedure in the `cRecordLogger` class module. It looks like our custom class has successfully sunk the `AfterUpdate` event. The form’s `AfterUpdate` event was propagated to the custom class module.

5. Click **OK** to close the message box.
6. Activate File Explorer and open the `C:\VBAAccess2021_ByExample\MyCust.txt` file.

The `MyCust.txt` file (see Figure 25.7) displays the record log. You may want to revise the `m_frm_AfterUpdate()` event procedure so that you can track whether a record was created or modified.

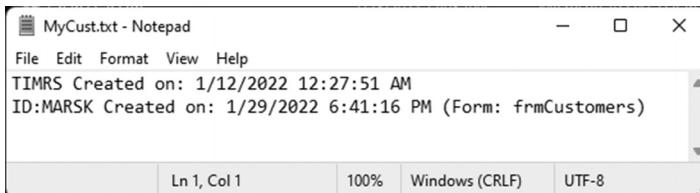


FIGURE 25.7. The `MyCust.txt` file is used by the `cRecordLogger` custom class for tracking record additions.

7. Close the `MyCust.txt` file.
8. Add a few more records to the `frmCustomers` form and check out the `C:\VBAAccess2021_ByExample\MyCust.txt` file.
9. Close the `frmCustomers` form.

After learning how to sink the form’s `AfterUpdate` event outside the form class module, you can use the same idea to sink other form events in a class module and make your code easier to implement and maintain. Just remember that if you want to sink events in a standalone class module, you must write code in two places: in your class module and in your form or report class module. The standalone class module must contain a module-level `WithEvents` variable declaration, and you must set the reference to an actual instance of the object in the form or report module.

#### ***Part 5: Using the `cRecordLogger` Custom Class with another Form***

The code you’ve written so far in this project can be reused in another Access form. In the remaining steps, we will hook it up to the `frmProducts` form. Let’s begin by creating this form.

1. In the Navigation pane of the Access window, select the **Products** table and choose **Create | Form**. Access creates a form and displays it in the Form Layout view.
2. Press **Ctrl+S** or click the **Save** button on the toolbar to save the form. In the Save As dialog box, type **frmProducts** for the form name and click **OK**.
3. Activate the **frmProducts** form in Design view. In the property sheet, make sure **Form** is selected from the drop-down box and click the **Other** tab. Set the **Has Module** property of the form to **Yes**. Save the changes to the form by pressing **Ctrl+S**.
4. Switch to the Visual Basic Editor window, and double-click the **Form frmProducts** object in the Project Explorer window. Access opens the Code module for the form.
5. Copy the code from the frmCustomers Code window to the frmProducts Code window. The code in the frmProducts Code window should match Figure 25.5 shown earlier.
6. In the frmProducts Code window, replace all the references to the object variable **clsRecordLogger** with **clsRecordLogger2**.

**NOTE**

*To quickly perform this operation, position the cursor inside the first clsRecordLogger variable name and choose Edit | Replace. The Find What text box should automatically display the name of the variable you want to replace. Type clsRecordLogger2 in the Replace With text box and click the Replace All button. Click OK to confirm the replacement of four instances of variable names. Click Cancel to exit the Replace dialog box.*

7. Save and close the frmProducts form.
8. Open the **frmProducts** form in Form view and in the **Records** group of the **Home** tab choose **New**.
9. Enter **Delicious Raisins** in the Product Name text box and press the record selector on the left side of the form to save the record.  
At this point you should receive the custom message about the audit trail that you defined in the AfterUpdate event procedure within the custom cRecordLogger class module. This indicates that the AfterUpdate event that was raised by the form when you saved the newly entered record was successfully propagated to the custom class module.
10. Click **OK** to close the message box.
11. Close the frmProducts form.
12. Open the **C:\VBAAccess2021\_ByExample\MyCust.txt** file to view the record log. Close this file when you are finished.

You may want to choose a more generic name for your record log text file if it will be used for tracking various types of information.

## **WRITING EVENT PROCEDURE CODE IN TWO PLACES**

If you write event procedure code for the same event both in the form module and in the class module, the code defined in the form class module will run first, followed by the code in the custom class module. You can easily test this by entering the following Form\_AfterUpdate event procedure code in the form class module of the frmCustomers or frmProducts forms prepared in Custom Project 25.1:

```
Private Sub Form_AfterUpdate()
    MsgBox "Transferring control to the custom class."
    ' when you click OK to this message, the code
    ' inside the AfterUpdate procedure in the custom
    ' class module will run
End Sub
```

When you open the form and add and save a new record, the Form\_AfterUpdate event will fire and you will see the message about transferring control to the custom class. Next, the AfterUpdate event procedure will run in the custom class, and you will see a message informing you that you can view the audit trail in the specified text file.

## **RESPONDING TO CONTROL EVENTS IN A CLASS**

---

Everyone designing Microsoft Access forms sooner or later realizes that it takes a long time to customize some of the controls placed on the form. It's no wonder then that once the control is working correctly, there is a tendency toward copying the control and its event procedures to a new form that requires a control with the same functionality. If you followed this chapter carefully, you already know a better (and a neater) solution. By using the `WithEvents` keyword you can create an object variable that points to the control raising the events. Instead of responding to control events in the form module, you will react to these events in a different location: a standalone class module. This lets you write centralized code that is easy to implement in other form controls of the same type.

Suppose you need a text box that converts lowercase letters to uppercase and disallows numbers. Hands-On 25.1 demonstrates how to create a text box with these features and hook it up with any Microsoft Access form.



### Hands-On 25.1. Responding to Control Events in a Class

This hands-on exercise requires prior completion of Custom Project 25.1.

1. Activate the Visual Basic Editor window and choose **Insert | Class Module**. A new class named Class1 will appear in the Project Explorer window.
2. In the Properties window, click the **(Name)** property and type **UCaseBox** as the new name of Class1. Click again on the **(Name)** property to save the new name. You should see the UCaseBox entry under the Class Modules folder in the Project Explorer.
3. In the UCaseBox class module's Code window, enter the following code:

```
Private WithEvents txtBox As Access.TextBox

Public Function InitializeMe(myTxt As TextBox)
    Set txtBox = myTxt
    txtBox.OnKeyPress = "[Event Procedure]"
End Function

Private Sub txtBox_KeyPress(KeyAscii As Integer)
    Select Case KeyAscii
        Case 48 To 57
            MsgBox "Numbers are not allowed!"
            KeyAscii = 0
        Case Else
            ' convert to uppercase
            KeyAscii = Asc(UCase(Chr(KeyAscii)))
    End Select

    With txtBox
        .FontBold = True
        .FontItalic = True
        .BackColor = vbYellow
    End With
End Sub
```

Notice that to respond to a control's events in a standalone class module you start by declaring a module-level object variable using the **WithEvents** keyword. In our text box example, we declared the object variable **txtBox** as an Access text box control.

Because the form can contain more than one text box control, we should tell the class which text box it needs to respond to. We do this by creating a Property Set procedure (like the one created in Custom Project 25.1) or a function procedure like the one shown above. We called this function **InitializeMe**,

but you can use any name you wish. Recall from Chapter 24 that a function entered in a class module serves as an object's method. We will call the `InitializeMe` method later from a form class module and pass it the actual control we want it to respond to (see Step 7). The `InitializeMe` method will assign the passed in control to the `WithEvents` object variable like this:

```
Set txtBox = myTxt
```

Next, we set the text box `KeyPress` property to [Event Procedure] to tell the class that we are interested in tracking this event.

Finally, we write the event procedure code for the text box control's `KeyPress` event. This code begins by checking the value of the key that was pressed by the user. If a number was entered, the user is advised that numbers aren't allowed, and the digit is removed from the text box by setting the value of `KeyAscii` to zero (0). Otherwise, if the user typed a lowercase letter, the character is converted to uppercase using the following statement:

```
KeyAscii = Asc(UCase(Chr(KeyAscii)))
```

`KeyAscii` is an integer that returns a numerical ANSI keycode. To convert the `KeyAscii` argument into a character, we use the `Chr` function:

```
Chr(KeyAscii)
```

Once we've converted a key into a character, we use the `UCase` function to convert it to uppercase:

```
UCase(Chr(KeyAscii))
```

Finally, we translate the character back to an ANSI number by using the `Asc` function:

```
Asc(UCase(Chr(KeyAscii)))
```

The `txtBox_KeyPress` event procedure ends by adding some visual enhancements to the text box. The text entered in it will appear in bold italic type on a yellow background.

4. Save the code you entered in the `UCaseBox` class module's Code window by pressing the **Save** button on the toolbar.
5. In the Project Explorer window, double-click the **Form\_frmProducts**. The `Form_frmProducts` class module's Code window should already contain code you entered while working with Part 3 of Custom Project 25.1 earlier in this chapter. To connect the `UCaseBox` class module with the actual text box, you

would need to enter the following code in a form's class module (do not enter it yet):

```
' module-level variable declaration
Private clsTextBox1 As UCaseBox

Private Sub Form_Open(Cancel As Integer)
    Set clsTextBox1 = New UCaseBox
    clsTextBox1.InitializeMe Me.Controls("ProductName")
End Sub

Private Sub Form_Close()
    Set clsTextBox1 = Nothing
End Sub
```

Because the frmProducts form already contains a call to the cRecordLogger class created earlier, all the procedures we need are already in place; therefore, we will simply add the appropriate lines of code to the existing procedures.

6. In the Form\_frmProducts Code window, enter the following module-level variable declaration just above the Form\_Open event procedure (see Figure 25.8):

```
Private clsTextBox1 As UCaseBox
```

This statement declares the `clsTextBox1` class variable. This variable is used in instantiating the `UCaseBox` object and connecting it with the actual text box control on the form (see the next step).

7. Enter the following lines of code before the `End Sub` statement of the Form\_Open event procedure (see Figure 25.8):

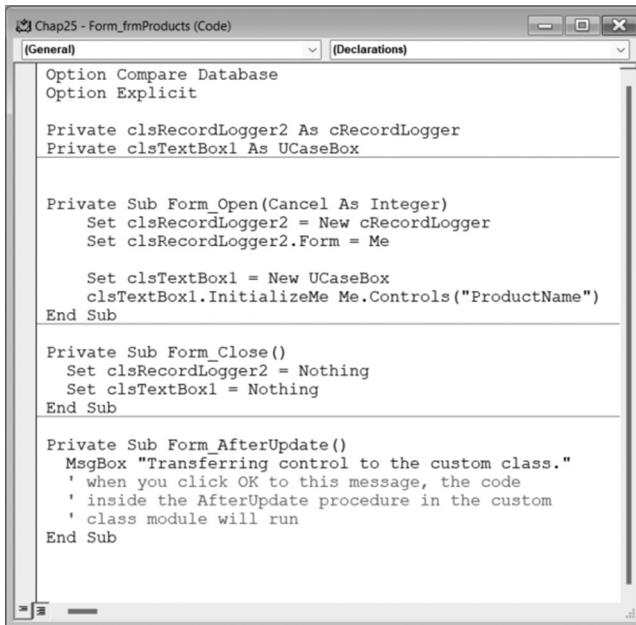
```
Set clsTextBox1 = New UCaseBox
clsTextBox1.InitializeMe Me.Controls("ProductName")
```

Before our `UCaseBox` class can respond to a text box's events, you need these two lines of code; the first one sets the class variable `clsTxtBox1` to a new instance of the `UCaseBox` class, and the second one calls the class `InitializeMe` method and supplies it with the name of the text box control.

8. Enter the following line of code before the `End Sub` statement of the Form\_Close event procedure (see Figure 25.8):

```
Set clsTextBox1 = Nothing
```

When we are done with the object variable, we set it to `Nothing` to release the resources that have been assigned to it.



The screenshot shows the Microsoft Access VBA code editor with the title bar "Chap25 - Form\_frmProducts (Code)". The code window displays the following VBA code:

```
Option Compare Database
Option Explicit

Private clsRecordLogger2 As cRecordLogger
Private clsTextBox1 As UCaseBox

Private Sub Form_Open(Cancel As Integer)
    Set clsRecordLogger2 = New cRecordLogger
    Set clsRecordLogger2.Form = Me

    Set clsTextBox1 = New UCaseBox
    clsTextBox1.InitializeMe Me.Controls("ProductName")
End Sub

Private Sub Form_Close()
    Set clsRecordLogger2 = Nothing
    Set clsTextBox1 = Nothing
End Sub

Private Sub Form_AfterUpdate()
    MsgBox "Transferring control to the custom class."
    ' when you click OK to this message, the code
    ' inside the AfterUpdate procedure in the custom
    ' class module will run
End Sub
```

FIGURE 25.8. The form class module shows code that instantiates and hooks up objects created in the cRecordLogger and UCaseBox class modules with the form and text box control.

9. Save the changes made in the Code window by clicking the **Save** button on the toolbar.
10. Open the **frmProducts** form in the Form view and in the **Records** group of the **Home** tab choose **New**.
11. Enter **prune butter** in the Product Name text box.
12. Notice that as you type, the characters you enter are converted to uppercase. They are also made bold and italic, and appear on a yellow background. If you happen to press a number key, which is disallowed by your custom **KeyPress** event, you receive an error message.
13. Click on the record selector to save the record.
14. Because this form also responds to the **AfterUpdate** event that we programmed earlier, you should see an additional message when you save the form.
15. Close the **frmProducts** form.

## DECLARING AND RAISING EVENTS

Standalone class modules automatically support two events: *Initialize* and *Terminate*.

Use the Initialize event to give the variables in your classes initial values. The Initialize event is called when you make a new instance of a class.

The Terminate event is called when you set the instance to `Nothing`. In addition to these default events, you can define custom events for your class module.

To create a custom event, use the `Event` statement in the declaration section of a class module. For example, the following statement declares an event named `SendFlowers` that requires two arguments:

```
Public Event SendFlowers(ByVal strName As String, cancel As Boolean)
```

The `Event` statement declares a user-defined event. This statement is followed by the name of the event and any arguments that will be passed to the event procedure. Arguments are separated by commas. An event can have `ByVal` and `ByRef` arguments. Recall that when passing the variable `ByRef`, you are actually passing the memory location of the variable. If you pass a variable `ByVal`, you are sending a copy of the variable.

When declaring events with arguments, bear in mind that events cannot have named arguments, optional arguments, or `ParamArray` arguments. The `Public` keyword is optional as events are public by default.

Use the `RaiseEvent` statement to fire the event. This is usually done by creating a method in a class module. For example, here's how you could trigger the `SendFlowers` event:

```
Public Sub Dispatch(ByVal toWhom As String, cancel As Boolean)
    RaiseEvent SendFlowers(toWhom, True)
End Sub
```

Notice that the `RaiseEvent` statement must be followed by name of the event you want to fire and optionally one or more arguments you want to pass to the event. Arguments must be enclosed by parentheses.

Events can only be raised in the module in which they are declared using the `Event` statement. After declaring the event and writing the method that will be used for raising the event, you need to switch to the form class module and perform the following tasks:

- Declare a module-level variable of the class type using the `WithEvents` keyword
- Assign an instance of the class containing the event to the object defined using the `WithEvents` statement
- Write a procedure that calls the class method
- Write the event handler code

The next hands-on exercise demonstrates how a user-defined event can be used in a class module. We will learn how to raise the SendFlowers event from a Microsoft Access form.



## Hands-On 25.2 Declaring and Raising Events

1. Activate the Visual Basic Editor window and choose **Insert | Class Module**. A new class named Class1 will appear in the Project Explorer window.
2. In the Properties window, click the **(Name)** property and type **cDispatch** as the new name of Class1. Click again on the **(Name)** property to save the new name. You should see the cDispatch entry under the Class Modules folder in the Project Explorer.
3. In the cDispatch class module's Code window, enter the following code:

```
Public Event SendFlowers(ByVal strName As String, _
    cancel As Boolean)

Sub Dispatch(ByVal ToWhom As String, cancel As Boolean)
    If ToWhom = "Julitta" Then
        cancel = True
        MsgBox "Dispatch to " & ToWhom & " was cancelled.", _
            vbInformation + vbOKOnly, "Reason Unknown"
    Else
        RaiseEvent SendFlowers(ToWhom, True)
    End If
End Sub
```

The first statement in the preceding code declares a custom event called **SendFlowers**. This event will accept two arguments: the name of the person to whom flowers should be sent and a Boolean value of **True** or **False** that will allow you to cancel the event if necessary.

Next, the **Dispatch** procedure is used as a class method. The code states that the flowers should be sent to the person whose name is passed in the **ToWhom** argument if the person's name is not "Julitta." The **RaiseEvent** statement will call the event handler that we will write in a form module in a later step.

4. Save the changes made in the Code module by clicking the Save button on the toolbar.
5. Switch to the Access window and click **Create | Form Design**. A blank form opens in the Design view. Place the text box control and a button control as shown in Figure 25.9. This form isn't bound to any data source. Use the property sheet to set the **Name** property of the text box control to **Recipient** and the **Caption** property of the accompanying label control to **Recipient**.

**Name:** Set the `Name` property of the command button to **cmdFlowers** and its `Caption` property to **Send Flowers**. Save this form as **frmFlowers**.

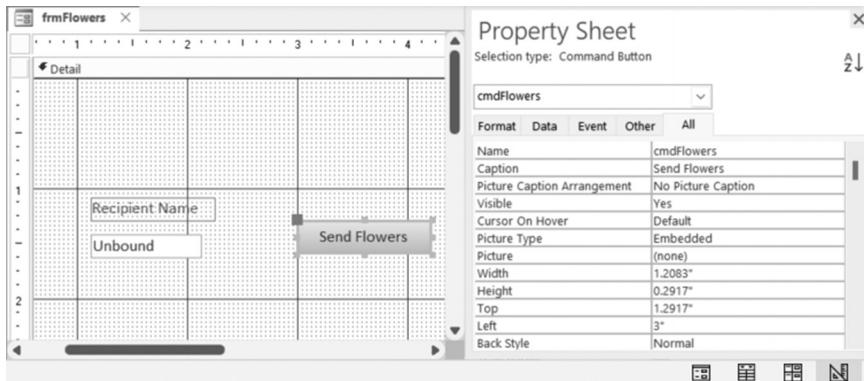


FIGURE 25.9. The frmFlowers form is used in Hands-On 25.2 to demonstrate the process of raising and handling custom events.

6. While the frmFlowers form is displayed in Design view, click the **View Code** button in the Tools area of the Design tab.
7. Enter the following code in the **Form\_frmFlowers** Code window:

```
Private WithEvents clsDispatch As cDispatch

Private Sub Form_Load()
    Set clsDispatch = New cDispatch
End Sub

Private Sub Form_Close()
    Set clsDispatch = Nothing
End Sub
```

Recall that the form class can respond to events from an object only if it has a reference to that object. Therefore, at the top of the form class module we declare the object variable `clsDispatch` by using the `WithEvents` keyword. This means that from now on the instance of the `cDispatch` class is associated with events.

The next step involves setting the object variable to an object. In the `Form_Load` event procedure, we create a class object with the `Set` statement and the `New` keyword. When the object variable is no longer needed, we release the reference to the object by setting the object variable to `Nothing` (see the preceding `Form_Close` event procedure).

Now that we are done with declaring, setting, and resetting the object variable, let's proceed to write some code that will allow us to raise the SendFlowers event when we click on the Send Flowers button.

8. In the Form\_frmFlowers Code window, enter the following **Click** event procedure for the cmdFlowers command button that you placed on the frmFlowers form:

```
Private Sub cmdFlowers_Click()
    If Len(Me.Recipient) > 0 Then
        clsDispatch.Dispatch Me.Recipient, False
    Else
        MsgBox "Please specify the recipient's name."
        Me.Recipient.SetFocus
        Exit Sub
    End If
End Sub
```

Notice that this event procedure begins by checking if the user has entered data in the Recipient text box. If the data exists, the `Dispatch` method is called; otherwise, the user is asked to enter data in the text box. When calling the `Dispatch` method, we must provide two arguments that this method expects: the name of the recipient and the value for the Boolean variable `Cancel`. Recall that the `Dispatch` method has the necessary code that raises the `SendFlowers` event (see Step 3). Now what's left to do is to write an event handler for the `SendFlowers` event.

9. Select the `clsDispatch` variable from the Object drop-down list in the upper-left corner of the Form\_frmFlowers Code window. As you make this selection, a template of the event procedure is inserted into the Code window, as shown here:

```
Private Sub clsDispatch_SendFlowers(ByVal strName As String, _
cancel As Boolean)

End Sub
```

The code that you write within this procedure stub will be executed when the event is generated by the object.

10. Enter the following statement inside the `clsDispatch_SendFlowers` procedure stub:

```
MsgBox "Flowers will be sent to " & strName & ".", , _
"Order taken"
```

Our custom event is not overly exciting but should give you an understanding of how custom events are declared and raised in a standalone class module and how they are consumed in a client application (form class module). The complete code entered in the frmFlowers form class module is shown in Figure 25.10.

```
Chap25 - Form_frnFlowers (Code)
cmdFlowers Click

Option Compare Database
Option Explicit

Private WithEvents clsDispatch As cDispatch

Private Sub clsDispatch_SendFlowers(ByVal strName As String, _
cancel As Boolean)
    MsgBox "Flowers will be sent to " & _
strName & ".", , "Order taken"
End Sub

Private Sub Form_Load()
    Set clsDispatch = New cDispatch
End Sub

Private Sub Form_Close()
    Set clsDispatch = Nothing
End Sub

Private Sub cmdFlowers_Click()
    If Len(Me.Recipient) > 0 Then
        clsDispatch.Dispatch Me.Recipient, False
    Else
        MsgBox "Please specify the recipient's name."
        Me.Recipient.SetFocus
        Exit Sub
    End If
End Sub
```

FIGURE 25.10 The form class module shows code that uses a custom object with its events.

11. Save the changes you made in the Code window by clicking the **Save** button on the toolbar.
12. To test the code, open the **frmFlowers** form in Form view, type any name in the Recipient text box, and click the **Send Flowers** button.
13. You should see the message generated by the SendFlowers custom event. Also see what happens when you type Julitta in the text box.

**NOTE**

*To get more hands-on experience with raising events in class modules, try out the example provided in the online help for the RaiseEvent statement topic. The quickest way to find this example is by positioning the cursor in the RaiseEvent statement (located in the cDispatch class module) and pressing F1.*

## SUMMARY

In this chapter, you were introduced to advanced concepts in event-driven programming. You learned how you can make your code more manageable and portable to other objects by responding to events in class modules other than form modules. This chapter has also shown you the process of creating your own events for a class and raising them from a public method by calling the `RaiseEvent` statement with the arguments defined for the event.

The important thing to understand is that while events happen in Access all the time whether or not you respond to them, you are the one to decide where to respond to the built-in events. And, if you ever find yourself short of an event, you can always create one that does exactly what you need by using the knowledge acquired in this chapter.

In the next chapter, you will learn about creating and using different types of macros in Access and converting embedded macros to VBA. You will also learn about the inner workings of Access database templates.

# Part VIII VBA AND MACROS

Writing VBA code is not the only way to provide rich functionality to your Access database users. Macros have long been used to enhance the user experience without writing a single line of VBA code. The Macro Builder allows you to include complex logic, business rules, and error handling in your macros.

In this part of the book, you are introduced to three types of macros that you can create in Access 2021. In addition, you learn how to convert macros to VBA and get started with built-in templates that extensively use macros.

## Chapter 26 Macros and Templates



# Chapter 26 MACROS AND TEMPLATES

When programming Access applications, there are two other areas of Access that you need to become acquainted with: macros and templates. Macros in Access have been around longer than the Visual Basic for Applications language. When Access 2 came out in 1992, it included a macro language called Access Basic that contained a subset of Visual Basic 2.0's core syntax. Access 95 replaced Access Basic with Visual Basic for Applications, but until Access 97, macros were the most common means of automating database tasks. When Access 2000 came out, many successful macro users had already moved to the new programming platform to take advantage of the language model that offered more control over Access. In fact, in versions 2000 through 2003, Microsoft recommended VBA to automate Access applications, and macros were supported mainly for backward compatibility.

The outlook on macros changed with the release of Access 2007. After performing some extensive research, Microsoft found out that many users were intimidated by the programming environment that Access provided but were quite successful at creating macros. It seems that it is much simpler to pick a macro action and set a couple of parameters than it is to write VBA code. Because most of the Access applications created by end users are loaded with macros, Microsoft decided to improve the “macro experience” in Access 2007 by adding event handling, temporary variables (TempVars), better error handling, and a new type of macro called an embedded macro. In Access 2010, Microsoft

added a Macro sandbox, which was related to the security model introduced in the 2007 release. Access 2010 also brought a powerful enhancement known as data macros. In this chapter, you will work with the redesigned Macro Builder and learn how to create standalone and embedded macros in Access 2021. After we've discussed macros, we will take a look at the .accdt file format used with Access desktop database templates.

## **MACROS OR VBA?**

---

You can use both VBA and macros to automate your Access applications. While macros have become very powerful, whether you use macros or VBA will depend on what you want to do. Macros can perform just about any task you can do with the Access user interface by using the keyboard or the mouse. They provide an easy way of opening and closing various Access objects (tables, queries, forms, and reports). You can also use them to automate repetitive tasks, execute commands on the Access Ribbon, set values for form and report controls, import and export spreadsheet and text files, display informative messages, or even sound a beep. With data macros you can also enforce business rules at a table level. These are just a few examples of what macros can do.

What macros cannot do is create and manipulate database objects the way we did in VBA earlier in this book by using DAO or ADO, or step through the records in a recordset and perform an operation on each record. You need to write VBA code to perform these types of operations. You must also use VBA when you need to pass parameters to your Visual Basic procedures, call dynamic link libraries (DLLs), create custom functions, or find out whether a file exists on the system. Even if you don't want to get started with macros now that you know how to write code in VBA, you still need to understand how macros are used in Access, as Microsoft makes extensive use of macros in their templates and Access built-in Button Wizard creates embedded macros.

## **ACCESS 2021 MACRO SECURITY**

---

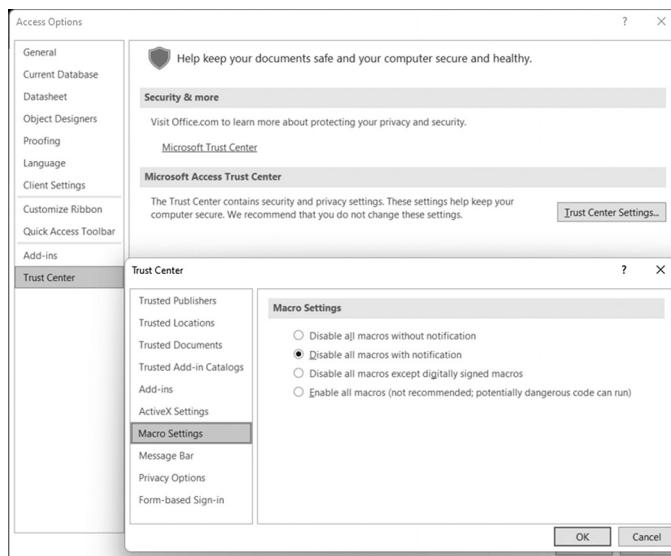
In Microsoft's documentation, the term "macro security" applies to macros and VBA, as well as other executable content that could be harmful when allowed to run. In Chapter 1, we specified that Access should trust any database file opened from a designated trusted folder (see Hands-On 1.4). This enabled you to work with this book's examples without having to constantly deal with the Access

security warning. However, if you attempt to open a file that contains macros and that file is not located in a trusted location, Access will determine whether to display a security alert by checking your macro settings (see Figure 26.1).

<b>NOTE</b>	<i>As Microsoft continues to improve security in Office, the default behavior and banners displayed in Access and other Office applications may be different from those presented in the instructions and images included in this book. For the most recent guidelines, please see <a href="https://docs.microsoft.com/en-us/deployoffice/security/internet-macros-blocked">https://docs.microsoft.com/en-us/deployoffice/security/internet-macros-blocked</a>.</i>
-------------	---

You can change your macro settings at any time by following these steps:

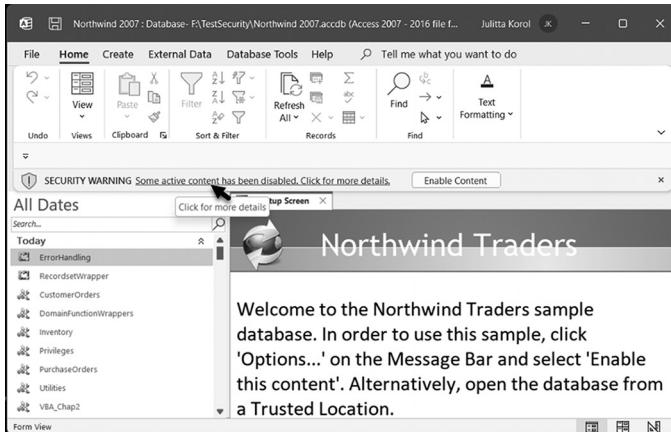
1. Click the **File** tab, then click **Options**.
2. In the Access Options dialog box, click the **Trust Center** tab, then click **Trust Center Settings**.
3. In the Trust Center dialog box, select **Macro Settings**.



**FIGURE 26.1** The Macro Settings options allow you to specify whether the macros should be disabled or allowed to run and whether you should see a notification when macros are disabled.

If the Disable all macros with notification option is selected, you may want to leave that setting as is. This option allows you to enable the disabled content only for this session by clicking the Enable Content button in the Security Warning message bar when a database file is opened.

You can access advanced security options by clicking the message text to the left of the Enable Content button in the Security Warning message bar (Figure 26.2).



**FIGURE 26.2** The Security Warning message appears on an attempt to load the database file containing content that could possibly harm your computer.

This will activate the Backstage View Info tab where you can click the Enable Content button to bring up a menu of additional options as shown in Figure 26.3. When you click Advanced Options, Access displays the dialog box shown in Figure 26.4.



**FIGURE 26.3** The Info tab in Backstage View displays information related to the Security Warning message and a brief description of the active content. By clicking on the Enable Content button, you can either enable all content in the current database or choose advanced options that allow you to specify which active content should be enabled.



FIGURE 26.4 The Microsoft Office Security Options dialog box allows you to temporarily enable disabled programming content by selecting the Enable content for this session radio button.

If you select the first radio button in Figure 26.4, Access will open the database in Sandbox mode, which means that it will turn off all executable content such as

- VBA code and any references to it
- Unsafe expressions  
An unsafe expression contains functions that could allow a user to modify the database or gain access to resources outside the database.
- Unsafe macro actions  
These are actions that could allow a user to modify the database or gain access to resources outside the database.
- Certain types of queries such as:
  - Action Queries  
These are queries that could allow a user to make unauthorized additions, changes or deletions of database data.
  - Data Definition Language (DDL) Queries  
These are queries that are used to create or alter objects in a database, such as tables and procedures.

- SQL Pass-Through Queries

These queries allow a user to send commands directly to a database server that supports the Open Database Connectivity (ODBC) standard.

- ActiveX controls

These are small programs that have unrestricted access to your computer's file system that could be used to take control of your computer.

If you plan on distributing your Access database in the .accdb file format, you can use the IsTrusted property of the CurrentProject object to test whether your application has its executable content disabled. Use this property in an AutoExec macro to check whether your application can load (see the next section).

## USING THE AUTOEXEC MACRO

---

The most important macro that every Access programmer needs to be familiar with is the AutoExec macro. This macro is not new in Access 2021; it's been with Access since the very beginning. An AutoExec macro in your Access application will automatically run when the database is opened. This is very convenient, especially when you need to check whether the rest of your application will load. Let's see how Microsoft does this in the Northwind 2007 database.

<b>NOTE</b>	<i>All code files and figures for the hands-on projects may be found in the companion files.</i>
-------------	--



### Hands-On 26.1 Understanding and Using the AutoExec Macro

1. Copy **Northwind 2007.accdb** database from the companion files to your desktop or any other new folder that you want to create for this hands-on exercise.
2. Double-click the copied database file to open it.  
When Access starts, notice the appearance of the Security Warning message bar with the Enable Content button, as was shown earlier in Figure 26.2.
3. Click the **Enable Content** button.
4. Dismiss the Login dialog box by clicking the “X” in the upper-right corner.
5. In the Navigation pane, select **All Access Objects** (you may need to select Object Type from the drop-down menu to show this option) and activate the **Macros** group. Right-click the **AutoExec** macro name and choose **Design View**. Access displays the contents of the AutoExec macro, as shown in Figure 26.5.

Notice that the Macro Design tab displays the Ribbon with various tools related to running macros and working within the Macro Builder window. Read through the next section where we discuss various elements of this window and look at the content of the AutoExec macro.

- Close the AutoExec macro in the same way you close any other Access window (by clicking the “X” button in the window’s upper-right corner or right-clicking the **AutoExec** tab and choosing **Close**).

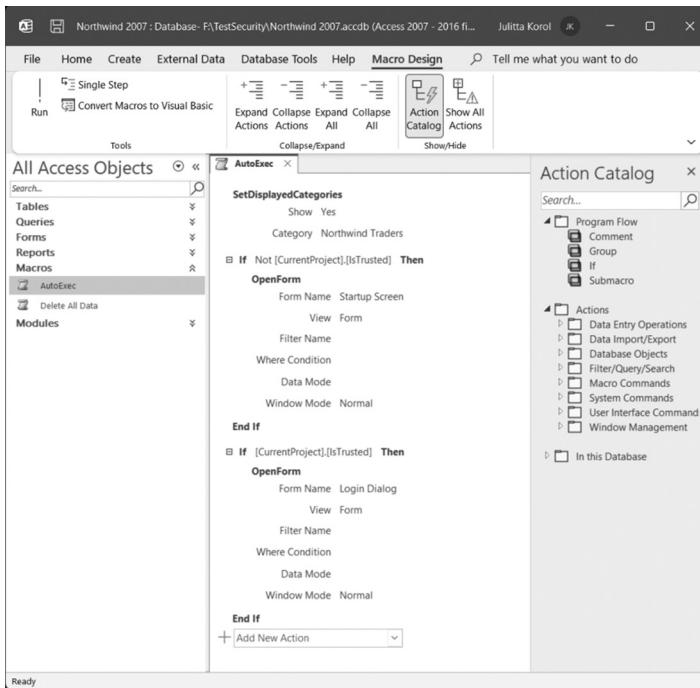


FIGURE 26.5 The contents of the AutoExec macro in the Northwind 2007 sample database as shown in Access 2021.

### Understanding Macro Actions, Arguments, and Program Flow

A macro can have more than one action, but you must specify at least one action when you create a macro. When you open the Macro Builder with an existing macro, you will see macro actions with various conditions filled in. At the bottom of the macro, there is an Add New Action drop-down list preceded by the plus sign image. This box is where you select the next action you want to add to the macro. When you select an action from the drop-down list, the Macro design area will expand to show more options. For example, if the selected

action requires additional data, a list of arguments is displayed. Access has a long list of macro actions to pick from. If you are not sure which action to select to perform a particular task, you can browse the Action Catalog that appears to the right of the macro content window (see Figure 26.5 earlier). All available macro actions are grouped by subject in the Action Catalog. When you expand the list of actions and select a specific action, you will see the description of the selected macro action at the bottom of the Action Catalog (see Figure 26.6). In addition to a hierarchical listing of macro actions, the Action Catalog contains several program flow constructs that you can apply to your macros. These are shown at the top of the Action Catalog. Comments should be used to document your macros. The Groups make it easy to organize your macro actions in a named block that can be easily collapsed, moved, or copied. The IF construct allows you to create macros based on a condition. Your condition could test a value in a field or evaluate the result of a function. You can use any expression that evaluates to True/False (Yes/No). To add conditional logic to your macro, double-click or drag the `If` to the macro design area. The macro actions will execute when the condition defined at the top of the `If` block is true. If the condition is not true, the action will be skipped, and the macro control will move to the next row. The actions that should not be executed when the condition is true are preceded with `Not`, as shown in Figure 26.5 earlier.



FIGURE 26.6 The Action Catalog in Access 2021. The description of the selected macro action appears at the bottom of the window.

To see all the available actions in the catalog, click the Show All Actions button on the Ribbon.

Actions that are considered unsafe are denoted by a yellow warning sign to the left of the macro action name.

The AutoExec macro included in the Northwind 2007 database and shown in Figure 26.7 uses the following macro actions:

- **SetDisplayedCategories**—This action (found in the User Interface Command section of the Action Catalog) is used to specify which categories are displayed under Navigate to Category in the title bar of the Navigation pane. This action has two arguments. The `Show` argument can be set to Yes to show the category name or No to hide it. The `Category` argument specifies the name of the category you want to show or hide. The Northwind 2007 database contains a custom category named Northwind Traders, so the macro starts by displaying this category in the Navigation pane.

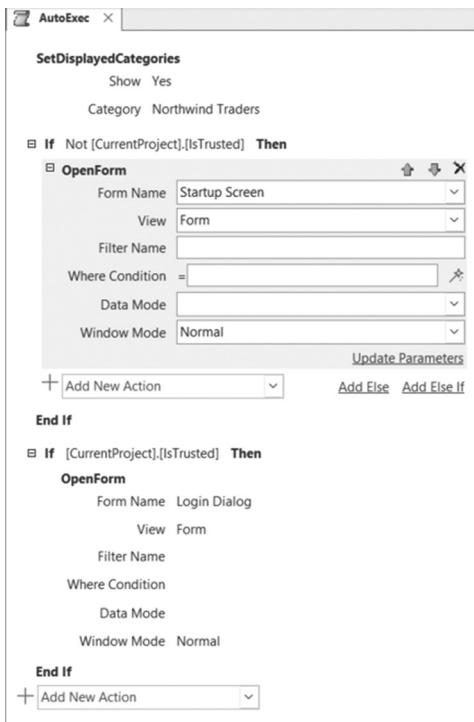


FIGURE 26.7 The AutoExec macro is shown here in Edit mode.

- **OpenForm**—This action (found in the Database Objects section of the Action Catalog) is used to open any form. The form can be selected from a drop-down list when you click the Form Name box. All forms in the current database will be shown. You can also specify the view in which the form will open. The default view is Form; you can select the view from the View drop-down box. Not all arguments need to be filled in. You can easily look up the meaning of an action's arguments by moving your mouse over the argument name. The second `If` block in the AutoExec example tells Access to open the Login dialog box in Form view using the Normal window. Notice that values for some arguments (`Filter Name`, `Where Condition`, and `Data Mode`) are not provided.

Whether a form opens can depend on a certain condition being met; in this example, the first `If` block tells Access to show the startup screen only if the current project (the database) is not trusted:

```
If Not [CurrentProject].[IsTrusted]
```

Notice how the `IsTrusted` property of the `CurrentProject` object is used to test whether your application has its executable content disabled. You saw this code block execute when you opened the Northwind 2007 database.

The next `If` block loads the Login dialog box if `[CurrentProject].[IsTrusted]` is true. This code block was executed when you told Access to “Enable Content” (see Step 3 in Hands-On 26.1).

**NOTE**

*To open an Access database without running the AutoExec macro, hold down the Shift key while opening the database.*

## CREATING AND USING MACROS IN ACCESS 2021

---

Access 2021 supports three types of macros:

- standalone macros (also used in versions of Access prior to 2007)
- embedded macros (introduced in Access 2007)
- data macros (introduced in Access 2010)

Standalone macros are visible in the Navigation pane under Macros. Embedded macros are part of the object in which they are embedded (form, report, or control) and therefore are not visible in the Navigation pane. Data macros allow developers to implement business rules in an Access application. These macros

do not have a user interface; they are applied at the table level and cannot be used to open a form or a report.

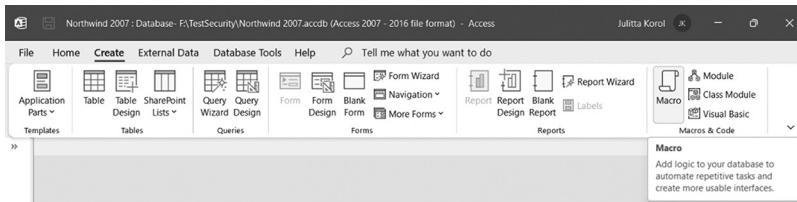
In the following sections, we take a closer look at each of these macro types.

## **Creating Standalone Macros**

The AutoExec macro we looked at in the previous section is a standalone macro. Once created, this macro appears in the Navigation pane.

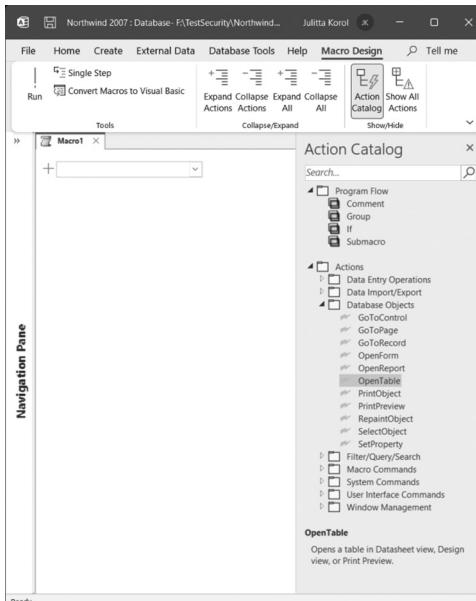
The general steps to create a standalone macro are as follows:

1. Click **Macro** in the Macros & Code group of the Create tab (Figure 26.8).



**FIGURE 26.8** Creating a standalone macro.

Access displays the Macro Builder window with one drop-down box, as shown in Figure 26.9. As you can see, the layout has a collapsible drop-down interface.



**FIGURE 26.9** The initial Macro Designer window.

2. Choose the action from the drop-down list. When the Ribbon's Show All Actions button is selected, this list displays all the available macro actions. When this button is not selected, you will see a shorter list of actions that are allowed to run even if the database is not trusted.

**NOTE**

*You can also add a macro action to the macro design surface by double-clicking an action in the Action Catalog or dragging an action onto the macro design surface. To activate the Action Catalog, click the Action Catalog button in the Ribbon's Macro Tools Design tab.*

3. If the macro action you select requires arguments, Access displays an inline dialog box where you can specify the required values (see Figure 26.10). Default argument values are prefilled for you. Notice that the code blocks are collapsible. You can expand or collapse the code areas by clicking the +/- controls to the left side of the code block or using buttons in the Collapse/Expand group of the Ribbon.

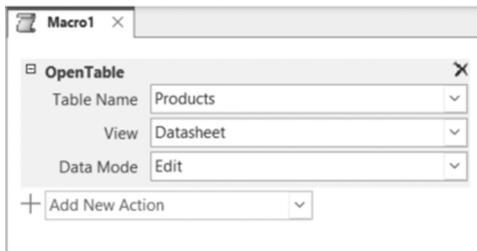


FIGURE 26.10 The OpenTable macro action opens a table. You need to specify the required arguments: the name of the table to open, the type of the view for the presentation of the data, and the Data Mode.

4. If desired, add another macro action as shown in Figure 26.11.

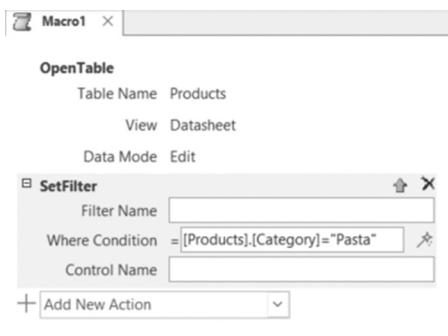


FIGURE 26.11 You can restrict the number of records in a table by using the SetFilter macro action.

5. If the macro action should be conditionally executed (see Figure 26.12), choose the `If` block from the Actions drop-down box. Type your conditional expression in the text box or click the Builder button next to the expression box to invoke the Expression Builder. Because the macro actions within the `If` block only run when the conditional expression resolves to True, the expression you enter must be of the Boolean type (True/False). If the condition evaluates to False, the action specified within the `If` block will be skipped.

<b>NOTE</b>	<p><i>In Access 2007 and earlier, you could write only simple conditional statements in the Macro builder. In Access 2010–2021, Macro Builder allows you to create complex conditions by using the <code>If</code>, <code>Else If</code>, and <code>Else</code> statements. To include these statements, click the <code>Add Else If</code> or <code>Add Else</code> hyperlinks in the lower part of the code block (see Figure 26.12).</i></p>
-------------	---



FIGURE 26.12 Adding conditions to the macro.

### SIDE BAR *Expression Builder in Access 2021*

To access the Expression Builder, click the Expression Builder icon shown in Figure 26.12.

Expressions are an important part of an Access application. They are used in tables, queries, forms, reports, and macros to evaluate and test data, perform calculations, manipulate character strings, and specify the logic that drives the behavior of your database application. Expressions in Access are like formulas and functions used in Excel. Depending on their complexity, they can contain user-defined or built-in functions, operators, identifiers, and constants.

Building expressions in Access is easy thanks to the Expression Builder (see Figure 26.13). The Expression Builder offers the IntelliSense feature that provides guidance as you type an expression. If you remember syntax and available functions and properties, you can enter your expression from scratch in the provided expression box. Otherwise, you can select the expression elements, categories, and values from the appropriate panes in the lower part of the Expression Builder window. Notice in Figure 26.13 that in addition to expression elements (Functions, Constants, and Operators), the Expression Elements pane also provides quick access to the Common Expressions. These include prebuilt expressions for displaying page numbers and the current date and time.

6. To make your macro actions easy to understand for yourself and others, you can add comments to the macro (see Figure 26.14). Comments are optional. To add a comment, choose **Comment** from the Actions drop-down box and type the text in the provided box. You can also type // in an Add New Action drop-down box. Comments are easy to spot because they appear as green text. You can move the comment to the appropriate location in your macro by clicking the Move Up or Move Down arrows to the right of the comment box.

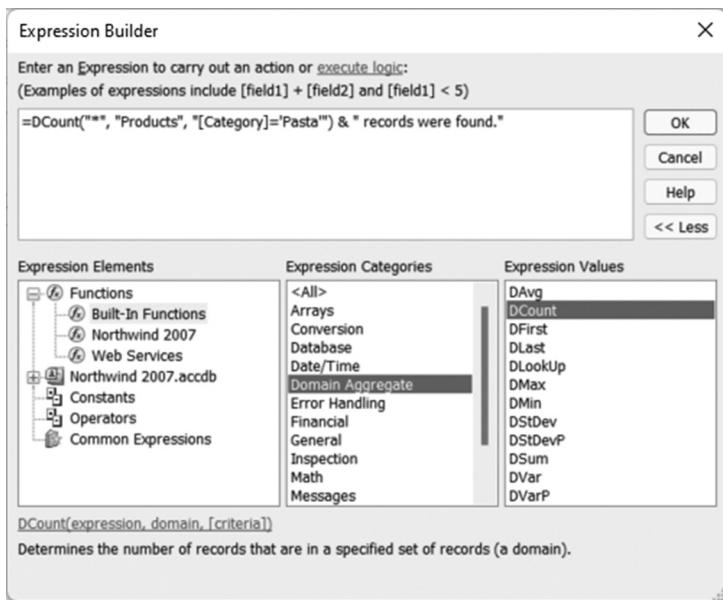
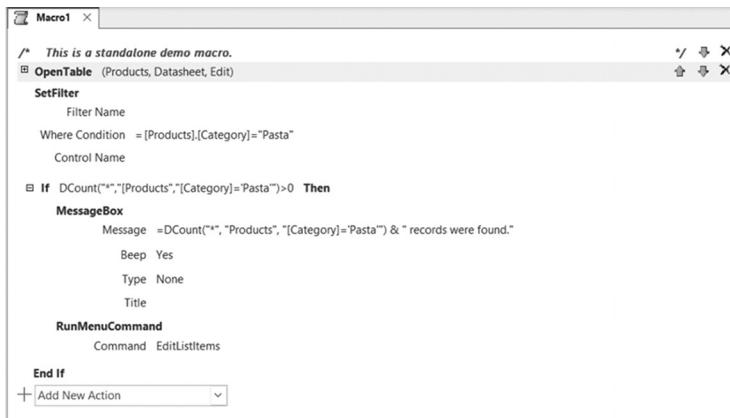


FIGURE 26.13 Building a macro expression using Expression Builder.



**FIGURE 26.14** Comments can be added anywhere within your macro code block.

- To add another action to your macro, select an action from the Actions drop-down.

To add an action between the actions you've already entered, first select the desired action from the Actions drop-down, and then move it to the appropriate location within your macro using the Move Up or Move Down arrows.

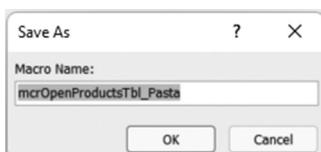
To delete an action, select it and click the X button. You can also right-click the action and choose **Delete** from the menu.

**NOTE**

*If you add an action that is considered “unsafe,” Access displays a yellow warning sign to the left of the macro action name. An unsafe action will not execute if the database is not trusted.*

For more complex macros you may want to use a program flow construct known as a *group*. With this construct you can put multiple actions and program flow into a group block so you can expand or collapse an entire group for better readability.

- When you are done entering all actions for your macro, press **Ctrl+S** to save your macro, or right-click the Macro tab and choose **Save**. Enter the macro name (`mcrOpenProductsTbl_Pasta`) in the Save As dialog box and click **OK** (see Figure 26.15).



**FIGURE 26.15** Saving a macro.

9. Close the Macro Designer window. The saved macro appears in the Navigation pane.

### ***Running Standalone Macros***

---

You can run standalone macros from the Design view, the Navigation pane, another macro, or a VBA procedure, or in response to an event on a form, report, or control.

- **Running a macro from the Design view**—If the standalone macro is open in the Design view, you can click the Run button in the Tools group of the Macro Design tab to run the macro.

**NOTE**

*You can also run your macro one action at a time by selecting the Single Step button and then clicking the Run button (see “Error Handling in Macros” later in this chapter).*

- **Running a macro from the Navigation pane**—A standalone macro can be run directly from the Navigation pane. Simply right click the macro name and choose Run from the shortcut menu, but make sure you know what the macro will do before you run it. A badly designed macro could wipe out all the data in your database without asking you if you want to proceed.

**NOTE**

*When you right-click a macro in the Navigation pane and that macro contains submacros, the Run command will only execute the first submacro (see “Creating and Using Submacros” in the next section).*

- **Running a macro from another macro**—To run a macro from another macro, you must create at least two macros. The main macro should include the RunMacro action. Set the `Macro Name` argument of this action to the name of the macro you want to run. When you run the main macro, both macros will execute.
- **Running a macro from a VBA procedure**—The `RunMacro` method of the `DoCmd` object carries out the `RunMacro` action in VBA.

To run a standalone macro, use the following statement:

```
DoCmd.RunMacro "YourMacroName"
```

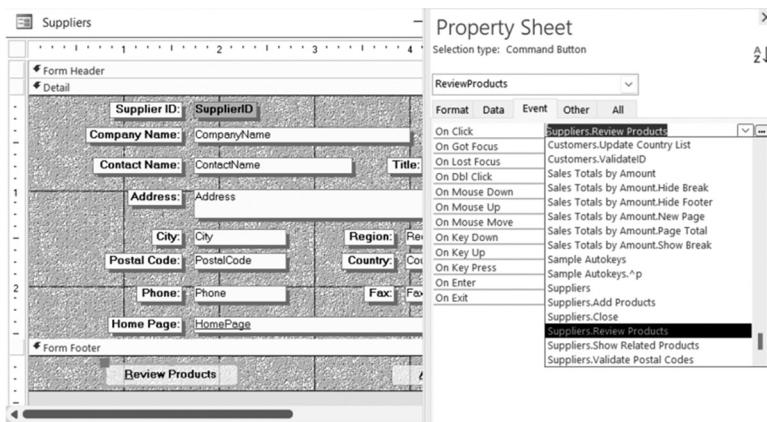
Optionally, you may specify how many times the macro should be run:

```
DoCmd.RunMacro "YourMacroName", 2
```

To run a macro with submacros, use the name of the main macro followed by a period and the name of the submacro:

```
DoCmd.RunMacro "Sales.AddProducts"
```

- **Running a macro in response to an event on a form, report, or control**—A standalone macro can be bound to events for forms, reports, or controls. For example, if your form contains a button that needs to open another form and you have previously created a macro that performs this action, you can specify the macro name in the OnClick property of the button, as shown in Figure 26.16. To do this, you must open the form in Design or Layout view, click the Button control on the form, and open the property sheet. On the property sheet for the button, click the Event tab, and then click the event property for the event you want to trigger. The macro will run when you return to Form view and click the button. Notice that Access lists all the available macros when you open a dropdown list next to an event property. Macros that contain submacros are listed in two parts—the name of the standalone macro and the name of the submacro (e.g., Suppliers.Review Products).



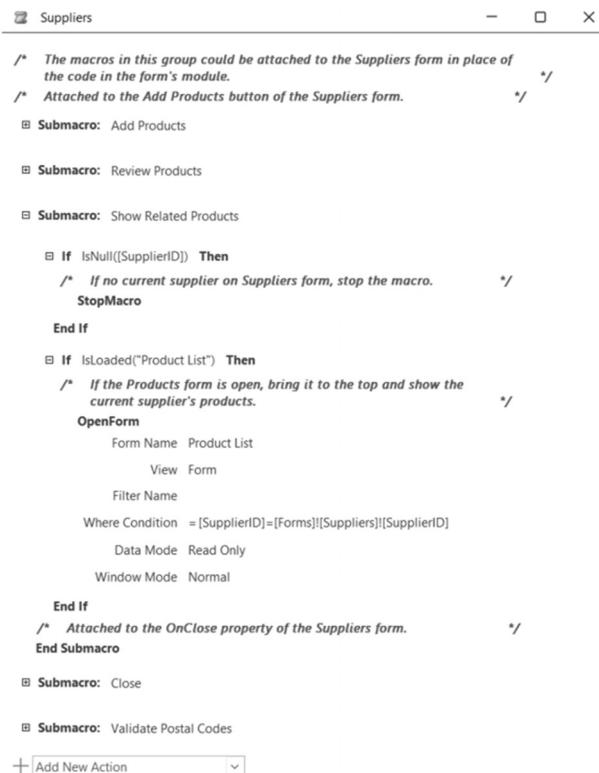
**FIGURE 26.16.** Binding a standalone macro to an event property. Shown here is the Suppliers form in the sample Northwind.mdb database from an earlier version of Access.

## Creating and Using Submacros

Instead of having a large number of standalone macros listed in the Navigation pane, consider storing related macros together using submacros. Submacros are similar to VBA subroutines in VBE modules. Figure 26.17 shows submacros that can be attached to the Suppliers form in the Northwind.mdb database. Notice how this single macro object named Suppliers stores a number of submacros, each of which performs a different action. To create submacros within a particular macro, you must give each submacro a unique name.

The general steps to create submacros are as follows:

1. Click the **Macro** button in the Macros & Code group of the Create tab.
2. Select **Submacro** from the Add New Action drop-down list. Access enters the default name for your submacro. Replace the suggested name with the desired name.



```

/* The macros in this group could be attached to the Suppliers form in place of
the code in the form's module.
/* Attached to the Add Products button of the Suppliers form. */
Submacro: Add Products

Submacro: Review Products

Submacro: Show Related Products

If IsNull([SupplierID]) Then
/* If no current supplier on Suppliers form, stop the macro. */
StopMacro
End If

If IsLoaded("Product List") Then
/* If the Products form is open, bring it to the top and show the
current supplier's products.
OpenForm
  Form Name Product List
  View Form
  Filter Name
  Where Condition =[SupplierID]=[Forms]![Suppliers]![SupplierID]
  Data Mode Read Only
  Window Mode Normal
End If
/* Attached to the OnClose property of the Suppliers form.
End Submacro

Submacro: Close

Submacro: Validate Postal Codes

```

**FIGURE 26.17** The Suppliers macro in the sample Northwind.mdb database contains submacros that can be used in the Suppliers form.

3. Specify the macro actions for your submacro.
4. Add another submacro if desired and specify the actions to perform.
5. Save the macro by pressing **Ctrl+S** and typing the name for the macro. The name you specify is the name of the main macro that contains the submacros. This name will appear in the Navigation pane under Macros.
6. Close the Macro Builder window.

Recall from an earlier section that when a macro contains submacros and you right-click the macro in the Navigation pane and choose Run, only the first submacro will execute.

Submacros are frequently implemented in forms and reports. To gain a better understanding of submacros, study the Suppliers macro and the Suppliers form in the Northwind.mdb database. Another excellent example of using submacros is the Customer Labels Dialog macro attached to the Customer Labels Dialog form in the Northwind.mdb database.

### **Creating and Using Embedded Macros**

---

Beginning with the release of Access 2007, macros can be embedded in any of the events provided by a form, report, or control. These embedded macros are not visible in the Navigation pane.

The general steps to create an embedded macro are as follows:

1. Open a form or report in Design or Layout view.
2. Select an object to which you want to assign an embedded macro (a form, report, or control).
3. Activate the property sheet. In Design view, the Property Sheet button is located in the Tools group of the Form or Report Design tab. In Layout view, you will find this button in the Tools group of the Arrange tab.
4. In the property sheet, click the **Event** tab, and then click the **Build** button (...) next to the desired property.
5. In the Choose Builder dialog box, select **Macro Builder**, then click **OK**. Access will open the same Macro Builder window that you use for creating standalone macros.
6. Choose the actions for your macro, and specify the arguments and conditions if required.
7. Press **Ctrl+S** to save your macro, and click the **Close** button in the Close group of the Macro Design tab. Access closes the Macro Design view and enters [Embedded Macro] in the event property (see Figure 26.18).

<b>NOTE</b>	<p><i>To modify the embedded macro, click on the Build button (...) next to the property with [Embedded Macro]. Access will open the Macro Builder (Design view) where you can make the required modifications.</i></p>
-------------	---

Keep in mind that you cannot reference an embedded macro from other macros. To reference a macro from another macro, you must create a standalone macro.

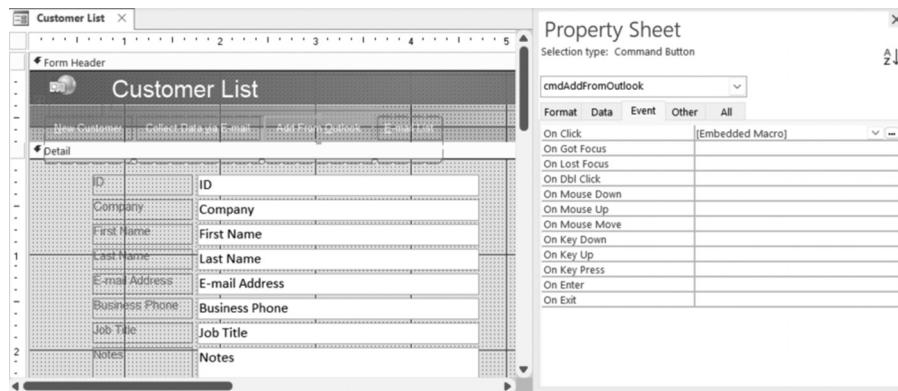


FIGURE 26.18 Assigning an embedded macro to the event property of a form's command button in the Northwind 2007.accdb database.

### Copying Embedded Macros

Because embedded macros are part of the object in which they are created, the macro behind the control is automatically copied when you copy the form, report, or control.

You can also copy an embedded macro from one event property to another. This is possible thanks to so-called “shadow properties.” What this means is that for each event property of a control, form, or report there is a “shadow” event property that contains the embedded macro for that property. For example, if your form’s On Load event property is set to [Embedded Macro], then the shadow property called On Load Macro contains its embedded macro. The On Click event property has the On Click Macro property if you are using the embedded macro to trigger the On Click event. If the event property is empty, then there is no shadow property.

Hands-On 26.2 demonstrates how to use VBA to copy an embedded macro from the Shipper Details form to the Supplier List form in the Northwind 2007.accdb database.



## Hands-On 26.2 Copying Embedded Macros

1. In the Northwind 2007.accdb database, open the **Supplier List** form in Design view. You may use the same version of the database that you opened in Hands-On 26.1.
2. In the Form Header section, right-click the **Home** button and choose **Copy**.
3. Right-click anywhere in the empty area of the Form Header section and choose **Paste**. The copied button appears in the upper-left corner of the Form Header section. Leave the button in this location for now until we change some of its properties. The button has the same label as the original button and a default name beginning with Command and followed by some numbers, such as Command231. You need to change the button's Name and Caption properties.
4. While the button is selected, click the **Property Sheet** button in the Tools group of the Form Design tab. Click the **All** tab and change the button's Name property to **cmdClose** and the Caption property to **&Close**. The ampersand in front of the letter "C" assigns a keyboard shortcut to the button.
5. Position the Close button to the left of the Home button as shown in Figure 26.19.

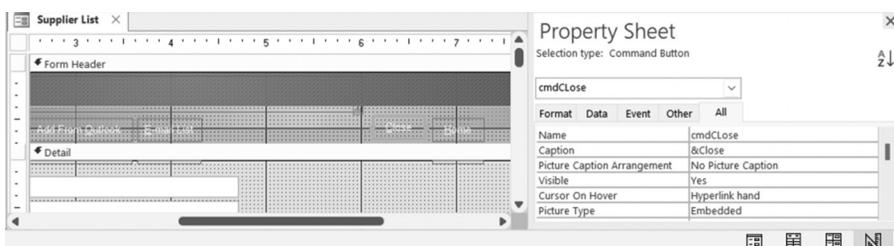
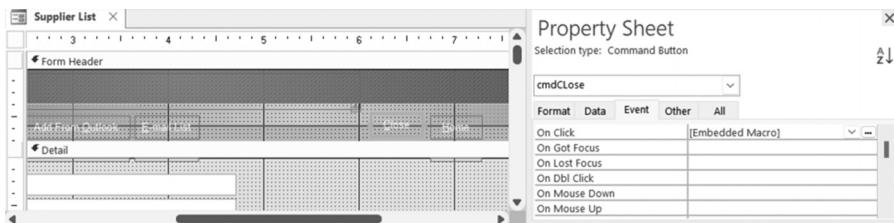


FIGURE 26.19 Use the property sheet to change the Name and Caption properties of the Close command button.

6. Press **Ctrl+S** to save the changes to the form.
7. While the Close button is selected, click the **Event** tab in the property sheet. Notice that when you copied the Home button, Access also copied the embedded macro attached to the On Click event property (see Figure 26.20). At this point, you could simply click the Build button (...) to modify this macro to have it close the Supplier List form instead of opening the Home form. However, the purpose of this exercise is to show you how to use VBA to copy an embedded macro from one property to another. We will overwrite this embedded macro with a different one by writing a VBA procedure in the next steps.



**FIGURE 26.20** When you copied the Home button, the new button inherited the embedded macro assigned to the On Click event property.

8. Press **Alt+F11** to activate the Visual Basic Editor window, and choose **Insert | Module**.
9. In the module's Code window, enter the following **Copy\_OnClickMacro** procedure:

```
Sub Copy_OnClickMacro()
    Dim ctl As Control

    ' open in the Design view the Supplier List form
    DoCmd.OpenForm "Supplier List", acDesign

    ' only run the code if the specified control
    ' exists on the form
    For Each ctl In Forms("Supplier List").Controls
        If TypeOf ctl Is CommandButton Then
            If StrComp(ctl.Name, "cmdClose", vbTextCompare) = 0 Then

                ' open in the Design view the Shipper Details form
                ' this form contains an embedded macro in the OnClick
                ' event of cmdClose button

                DoCmd.OpenForm "Shipper Details", acDesign

                ' copy macro from the OnClick event property of the
                ' cmdClose button on the Shipper Details form
                ' to the OnClick event property of the cmdClose button
                ' on the Supplier List form

                Forms("Supplier List").Controls("cmdClose").OnClickMacro = _
                    Forms("Shipper Details").Controls("cmdClose").OnClickMacro
                DoCmd.Save acForm, "Supplier List"
                DoCmd.Close acForm, "Shipper Details"
                MsgBox "The embedded macro was successfully copied."
            End If
        End If
    Next
End Sub
```

```
    End If
End If
Next

MsgBox "Operation could not be performed. " & vbCrLf & _
"Ensure that the specified control exists."
End Sub
```

In this procedure, we begin by opening the Supplier List form in Design view and iterate through the form's controls to find out whether the form contains the control named cmdClose. We use the `TypeOf...Is` expression to specifically look for the CommandButton control. Because the Supplier List form contains several buttons, we can use the `StrComp` function to determine if we found the correct button. This function will tell us if the string specified in the second argument is found in the string specified in the first argument. The third argument of the `StrComp` function tells Access to perform the comparison of the two text strings. If the `StrComp` function returns zero (0), then we found the control we were looking for and we can proceed to open the Shipper Details form and copy the embedded macro assigned to the On Click event property of this form's cmdClose button to the On Click event property of the Supplier List's equivalent button. The following statement copies the embedded macro from the On Click event property to another On Click event property:

```
Forms("Supplier List").Controls("cmdClose").OnClickMacro = _
Forms("Shipper Details").Controls("cmdClose").OnClickMacro
```

Once we are finished copying, we can simply exit the procedure using the early exit expression `Exit Sub`.

If the Supplier List form does not contain the button with the specified name, we display a message.

**10.** Run the `Copy_OnClickMacro` procedure.

If you followed all the steps of this hands-on exercise, you should see a message stating that the embedded macro was successfully copied. Click **OK** to close the message box. If you got a different message, check the code for any errors and ensure that the Supplier List form has the cmdClose button. Then rerun the procedure.

**11.** Press **Ctrl+S** to save changes in the module. Access will ask you to assign a new name to the module. Click **OK** to accept the default name.

**12.** Close the Visual Basic Editor window and return to the main Access window.

13. In the property sheet for the cmdClose button, click the **Build (...)** button next to the On Click event property on the Event tab. Access opens the Macro Design view, as shown in Figure 26.21. This macro will close the form when the user clicks the Close button on the Supplier List form.

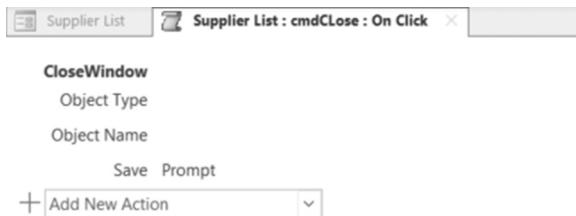


FIGURE 26.21 Examining an embedded macro after it's been copied from another event property.

14. Exit the Macro Design view by clicking the Close button in the Ribbon.  
 15. Right-click the **Supplier List** tab, and choose **Form View**.  
 16. Click the **Close** button in the Header section of the Supplier List form to close this form. This will execute the embedded macro you copied in this hands-on.

### Examining Shadow Properties

You can see the contents of the OnClickMacro shadow property by typing the following statement in the Immediate window and pressing Enter (the form must be open for this to work):

```
?Forms("Supplier List").Controls("cmdClose").OnClickMacro
```

You should see the output shown in Figure 26.22.

The screenshot shows the Microsoft Access Immediate window. The command entered was '?Forms("Supplier List").Controls("cmdClose").OnClickMacro'. The output displayed is the XML definition of the macro:

```

?Forms("Supplier List").Controls("cmdClose").OnClickMacro
Version =196611
ColumnsShown =8
Begin
  Action ="Close"
  Argument ="-1"
  Argument =""
  Argument ="0"
End
Begin
  Comment ="_AXL:<?xml version='1.0' encoding='UTF-16' standalone='no'?><015\012<UserInterfaceMacro For='cmdClose' xmlns='http://schemas.microsoft.com/office/accessservices/2009/11/application'><Statements><Action Name='CloseWindow' /></Statements></UserInterfaceMacro>"
End

```

FIGURE 26.22 Examining the Shadow property - OnClickMacro.

Access has a large number of hidden properties that make it possible to get and set embedded macros. The property name begins with the name of the event

property and ends with “EmMacro,” such as OnClickEmMacro, AfterUpdateEmMacro, and so on. Try the following statement in the Immediate window (the form must be open for this to work), and notice that it produces the same output as the previous statement:

```
?Forms("Supplier List").Controls("cmdClose") .  
Properties("OnClickEmMacro") .Value
```

With this knowledge, it is easy to create a standalone macro from an embedded macro. Here’s an example VBA procedure that does just that:

```
Sub SaveEmToStandalone()  
Dim strMacro As String  
Dim objFileSys As Object  
Dim objFile As Object  
Dim strFileName As String  
  
' open in the Design view the form that contains  
' the embedded macro  
DoCmd.OpenForm "Login Dialog", acDesign  
  
' to write an embedded macro to a file use the  
' Value property  
strMacro = Forms("Login Dialog") .  
Controls("cboCurrentEmployee") .  
Properties("AfterUpdateEmMacro") .Value  
  
' close the form  
DoCmd.Close acForm, "Login Dialog"  
  
' Create a text file  
strFileName = "C:\VBAAccess2021_ByExample\cboAfterUpdate.txt"  
  
Set objFileSys = CreateObject("Scripting.FileSystemObject")  
Set objFile = objFileSys.CreateTextFile(strFileName, True)  
  
' Write strMacro to the text file  
objFile.Write strMacro  
' Close the file  
objFile.Close  
  
' Use the undocumented LoadFromText method of  
' the Application object to create a standalone macro  
' from the text file  
Application.LoadFromText acMacro, _
```

```
"cboEmployeeAfterUpdate", strFileName  
End Sub
```

The `LoadFromText` method of the `Application` object makes it possible to create various Access database objects (including macros) from information that was previously saved to a text file. The `LoadFromText` method requires that you specify the object type, the object name, and the name of the text file.

After running this procedure, you should see the `cboEmployeeAfterUpdate` macro listed in the Navigation pane under Macros.

#### SIDE BAR *Working in Sandbox Mode*

By default Access runs in Sandbox mode, which means that the program blocks all the expressions in field properties and controls that are considered unsafe. A *safe expression* is one that does not use functions that could be used to access drives or other resources on a user's computer to damage data or files. When Access is running in Sandbox mode, any expressions that use unsafe macro actions are marked with a yellow warning sign.

Access allows you to disable Sandbox mode by setting the macro security level to low; however, for security reasons this setting is not recommended. If you trust the database and want to run unsafe expressions that the Sandbox mode blocks without having to change your current macro security, you can disable Sandbox mode by changing a Registry key. Modifying the Registry is beyond the scope of this chapter.

#### SIDE BAR *Generating Macros Using the Command Button Wizard*

You do not have to write all your macros from scratch. Access provides a built-in tool known as the Command Button Wizard. If you are working with the database in .accdb format, the wizard will generate embedded macros to open forms, run queries, find records, apply filters, or print reports. For older databases in the .mdb file format, the wizard creates VBA code.

### **Using Data Macros**

Prior to Access 2010, macros could only be attached or embedded in forms and reports. Access programmers had long asked for a feature similar to SQL triggers that would enable them to automatically update data in a table or track when a record was last modified or deleted. Microsoft answered this programming request in Access 2010 by introducing *data macros*. A data macro contains one or more actions that execute in response to a table event.

With data macros programmers can enforce complex business rules at table level. For example, by attaching a data macro to a table you can control what happens to a table's data when the user interacts with the data via an Access form. You can specify what occurs after data is inserted, updated, or deleted. For instance, you may want to verify the accuracy of table data, send an email notification to the database manager about the changes that occurred to the data, or automatically update fields in another table. By using the data macros attached to the After Insert, After Update, and After Delete events, you can check and modify records in the current table or other tables. You can use the `For Each Record` construct to iterate through a set of records in a table to update records that meet certain criteria or accumulate the totals. You can also perform specific actions before data is inserted, changed, or deleted. The Before Change data macro event will allow you to check a value in another table and, if necessary, prevent a change or insert from happening. You can use an `IsInsert` property to detect whether it's an insert or an update operation. You can find out whether the value of a specific field has changed by using the `Updated` function, and, if the value of a field has changed, you can use the `Old` property to find out the previous value of the field. Before deleting records you can use the Before Delete data macro event to determine whether the record can be deleted. You can also update an audit file to indicate that the record was deleted.

By using data macros you can guarantee that your business logic is executed even if the user modifies a record outside the forms you provide such as in a Datasheet view or by running another macro or a VBA procedure. Your data macros will run silently in the background regardless of how the data is accessed. With data macros, you no longer need to attach the same macro to a number of forms. All you need to do is add the logic to the table. Any form based on that table will inherit that logic.

In addition to *event data macros* that are triggered by table events, you can create standalone *named data macros*. Named data macros allow you to save time by incorporating the common tasks into one macro. Instead of repeating the same actions in multiple data macros, simply create a named data macro and call it from a data event. Named macros can be called using the `RunDataMacro` action.

Keep in mind that data macros do not have any user interface (UI); they are stored within a table itself and therefore do not show up in the Navigation pane. Do not attempt to use data macros to handle multivalue and attachment data types as they are not supported. Also, keep in mind that data macros can only be attached to events in local tables, not linked tables.

### ***Creating a Data Macro***

In the following hands-on exercise, you will work with the Purchase Order Details table in the Northwind 2007 database. You'll write a data macro to ensure that the order quantity cannot be modified if the order was already posted to inventory or the Date Received field contains a date value. The following VBA procedure has already been written by the Microsoft team to validate the Quantity field in the Form\_Purchases subform for Purchase Order Details:

```
Private Sub Quantity_BeforeUpdate(Cancel As Integer)
    If Me! [Posted To Inventory] Or_
        Not IsNull(Me! [Date Received]) Then
            MsgBoxOKOnly CannotModifyPurchaseQuantity
            Cancel = True
    End If
End Sub
```

While this procedure works just fine for controlling data entry operations on the form, it has no effect on data manipulations performed directly at the table level. By creating a Before Change data macro, you can ensure that this test scenario is addressed no matter how data is being accessed.



### ***Hands-On 26.3 Creating and Testing a Data Macro***

1. Start the **Northwind2007.accdb** database located in your VBAAccess2021\_ByExample folder. Log in as **Andrew Cencini**.
2. Close the **Home** form that is automatically launched upon login.
3. Open the **Purchase Order Details** table.
4. Select the **Table** tab on the Ribbon and click the **Before Change** button (Figure 26.23).

#### **NOTE**

*Data macros can be created from the table Datasheet view or Design view (see Figure 26.23 and 26.24).*

ID	Purchase Order ID	Product	Quantity	Unit Cost	Discontinued
99	99	Northwind Traders Chai	40	\$14.00	
239	91	Northwind Traders Syrup	100	\$8.00	
240	91	Northwind Traders Cajun Seasoning	40	\$16.00	
241	91	Northwind Traders Olive Oil	40	\$16.00	
242	92	Northwind Traders Boysenberry Spread	100	\$19.00	
243	92	Northwind Traders Dried Pears	40	\$22.00	
244	92	Northwind Traders Curry Sauce	40	\$30.00	
245	92	Northwind Traders Walnuts	40	\$17.00	
246	92	Northwind Traders Fruit Cocktail	40	\$29.00	

**FIGURE 26.23** Creating a data macro from the Datasheet view.

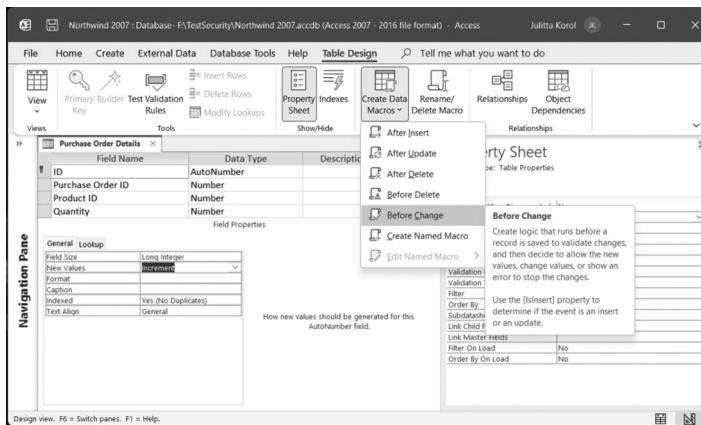


FIGURE 26.24 Creating a data macro from the Table Design view.

Table 26.1 lists five events that can trigger a data macro.

TABLE 26.1 Data macro events

Event Name	Event Description
Before Change	Runs before a record is about to be updated. Use it to validate changes before saving them to the table. You can include logic to allow new values or show an error to reject the changes. Use the <code>IsInsert</code> property to determine whether the change is an insert or an update.
Before Delete	Runs before a record is about to be deleted. You can include logic that validates the deletion and allows it, or cancels the deletion and raises an error.
After Insert	Runs after a new record has been added to the table.
After Update	Runs after a record has been edited in the table. Use the <code>Updated("Field Name")</code> function to determine if a specific field has changed. Use <code>Old.[Field Name]</code> to find out the value the field had before the record was changed.
After Delete	Runs after a record has been deleted from the table. Use <code>Old.[Field Name]</code> to find out the value the field had before the record was deleted.

- When you click the event name, Access opens the Macro Builder (Figure 26.25).

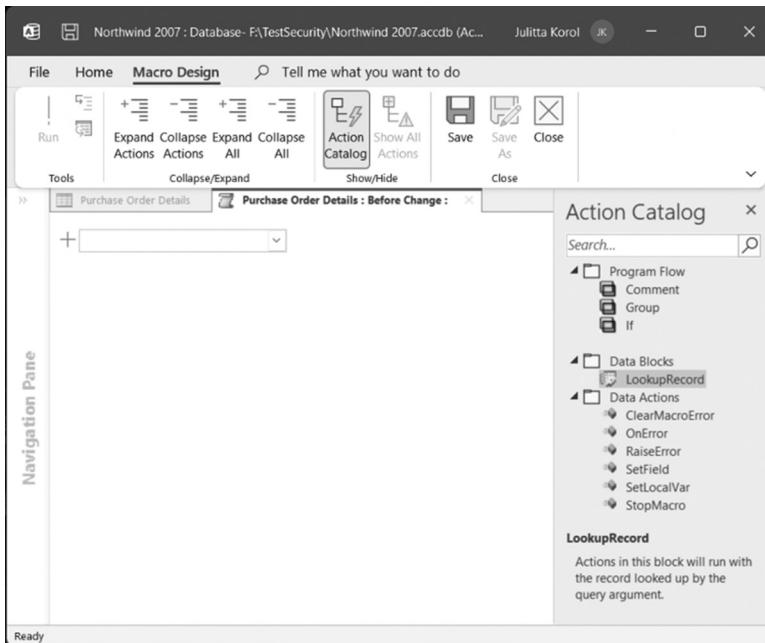


FIGURE 26.25 Macro Builder for writing data macros.

The Action Catalog shows three categories of actions that can be specified for a data macro: Program Flow, Data Blocks, and Data Actions. The actions listed in each category depend on the type of table event you have selected. When you are working with data macros, the only Program Flow constructs are comments (used for documenting your data macro), groups (used for organizing your macro), and If blocks (for applying a conditional logic). Data Blocks contain constructs that are used to perform specific operations on database records like looking up a record in a table (LookupRecord), adding a record to a table (CreateRecord), modifying an existing record in a table (EditRecord), and looping through every record in a table (ForEachRecord). Notice that only the LookupRecord data block is available for the Before Change event. When you select a construct from the Data Blocks category, you can add one or more actions and these actions will be performed as part of the data block. You can even nest data blocks. For example, you can set up a ForEachRecord data block to iterate through every record in a table and, depending on your conditional logic, create the CreateRecord data block to add a record to another table based on the found record.

The Data Actions category in the Action Catalog lists the available data actions. Some table events have more actions than others. You can find the description of an event by selecting it and then checking the bottom of the Action Catalog (see Figure 26.25).

6. Double-click the **If** construct in the Program Flow section. Access adds a conditional block as shown in Figure 26.26.

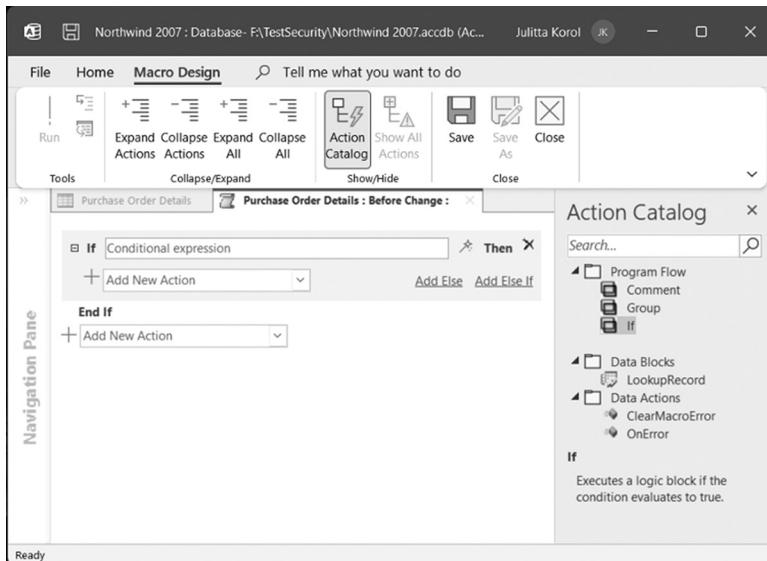


FIGURE 26.26 Adding an If block to the data macro.

7. In the If box, enter the following conditional expression on one line:

```
Updated("Quantity") And ([Posted To Inventory] Or Not  
IsNull([Date Received]))
```

8. Select **SetLocalVar** from the Add New Action drop-down located within the If...Then...End If block. Enter **strMsg** in the Name box and “” (an empty string) in the Expression box, as shown in Figure 26.27.

The SetLocalVar action allows you to create a local variable. In this macro, you'll use a local variable named `strMsg` to store the error message text that you'll retrieve from the Strings table that is a part of the Northwind 2007.accdb database. Notice that the initial value of the `strMsg` variable is set to an empty string.

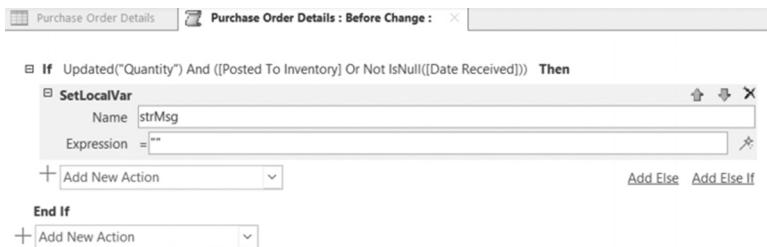


FIGURE 26.27 Adding a local variable to your data macro.

9. Select **LookupRecord** from the Add New Action drop-down located within the `If...Then...End If` block. The Macro Designer adds a LookupRecord block. Fill in the block as depicted in Figure 26.28. Choose **Strings** from the Look Up A Record In drop-down box, and enter `[Strings].[String ID] = 31` for the Where Condition. This condition tells the macro to find the 31st record in the Strings table. Notice that as you start typing in the Where Condition box the IntelliSense technology is at work displaying appropriate choices for you to select.

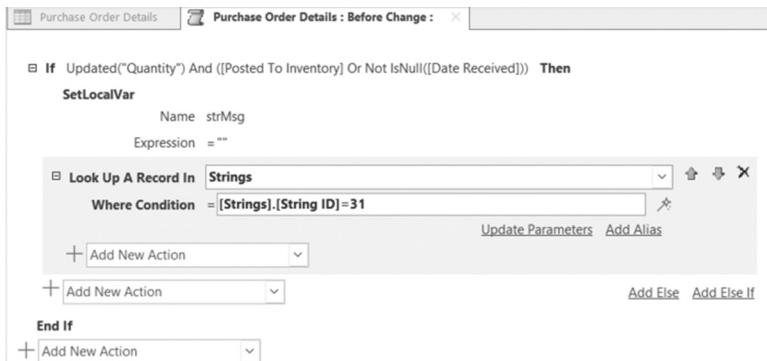


FIGURE 26.28 Adding an action to look up a record in a table.

10. Within the LookupRecord block, add a new **SetLocalVar** action. In the Name box, enter the name of the local variable **strMsg** that you declared at the beginning of the macro. In the Expression box, enter **[Strings].[String Data]**, as shown in Figure 26.29.

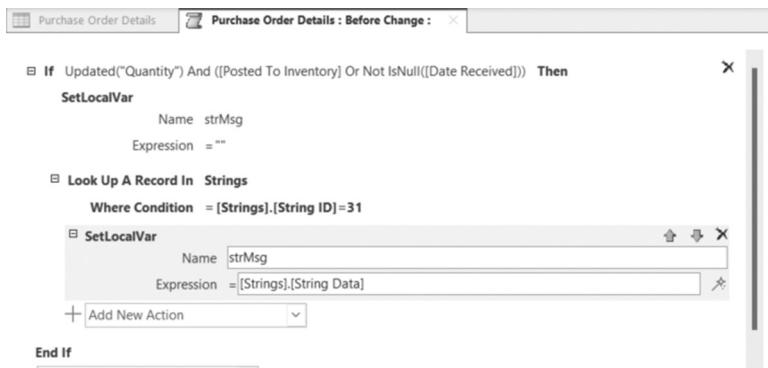


FIGURE 26.29 Storing data retrieved by the LookupRecord action in a local variable.

11. In the Add New Action drop-down box within the LookupRecord block, choose **Comment**. When a text box appears, enter the following text: **Record Lookup Completed**. Figure 26.30 shows the result of adding a comment. The comments appear in green italics between the /\* and \*/ delimiters.

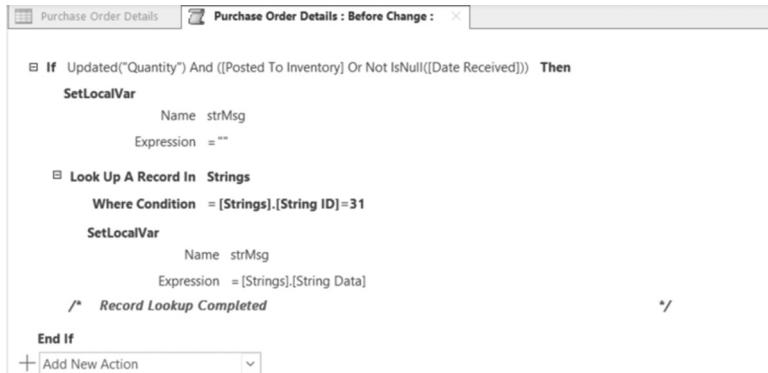


FIGURE 26.30 Adding a comment to a macro.

12. In the Add New Action drop-down box located outside the LookupRecord block, choose **RaiseError**. Enter **100** in the Error Number box and **= [strMsg]** in the Error Description box. This will tell the macro to display the text stored in the local variable `strMsg` when an error occurs. Be sure to enter the equals sign before the variable name. Figure 26.31 displays the completed macro.

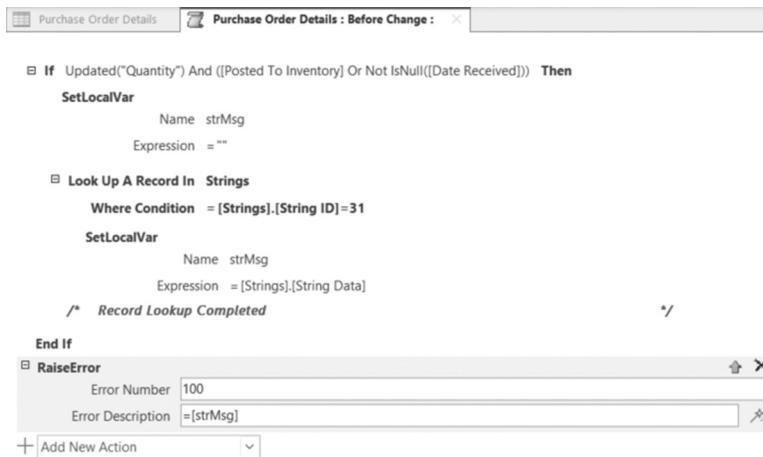


FIGURE 26.31 Adding a macro action to raise an error.

- Click the **Save** button to save your macro, then click the **Close** button to close the Macro Builder. Notice that when a macro is defined for a table event, the button with the event name has a shaded background (see Figure 26.32).

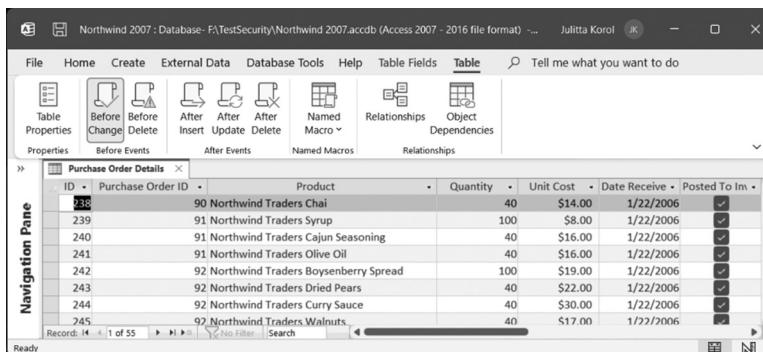


FIGURE 26.32 The highlighted Before Change button on the Ribbon indicates that there is a data macro attached to this event.

- To test your macro, you need to perform the action that will trigger the event for which you defined the macro. In the Purchase Order Details table, enter a different value in the Quantity field for any record that has both a checkmark in the Posted To Inventory field and a value in the Date Received field. When you attempt to save the record after making a change to the Quantity field, Access displays the error message shown in Figure 26.33. The error message has been retrieved from the Strings table. Click **OK** to the message and then press the **Esc** key to exit the edit mode.

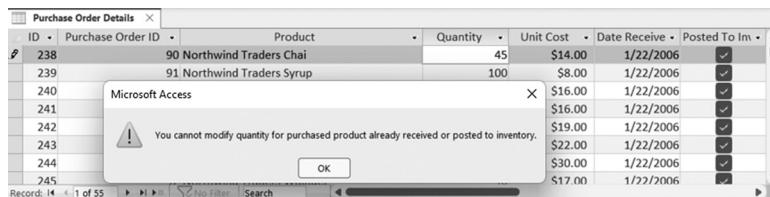


FIGURE 26.33 The error raised by the data macro assigned to the Before Change event.

### NOTE

*A form based on a table that contains a data macro will inherit the logic defined in the table. This means that you no longer need to write separate VBA code in the form class modules to respond to events that are already handled at a table level.*

### Creating a Named Data Macro

As mentioned earlier, in addition to writing data macros that are triggered by a table event, you can create named data macros. You can pass arguments to these macros and call them from anywhere within your application. To create a named data macro, follow these general guidelines:

1. In the Navigation pane, double-click the desired table to open it.
2. Select the **Table** tab on the Ribbon.
3. In the Named Macros group, choose **Named Macro | Create Named Macro** (Figure 26.34).

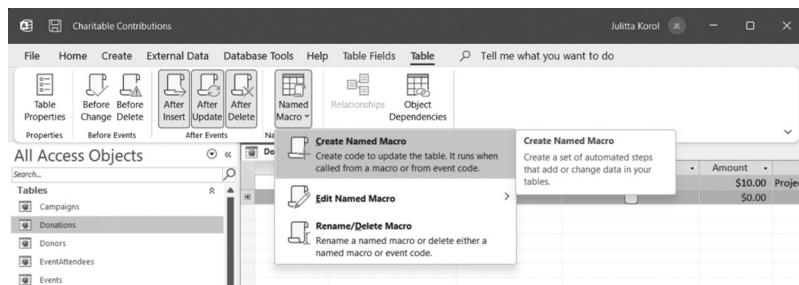


FIGURE 26.34 Creating a named data macro.

Access opens the Macro Builder, as shown in Figure 26.35. Notice that the Action Catalog lists a number of data actions that you can use in your named data macro logic.

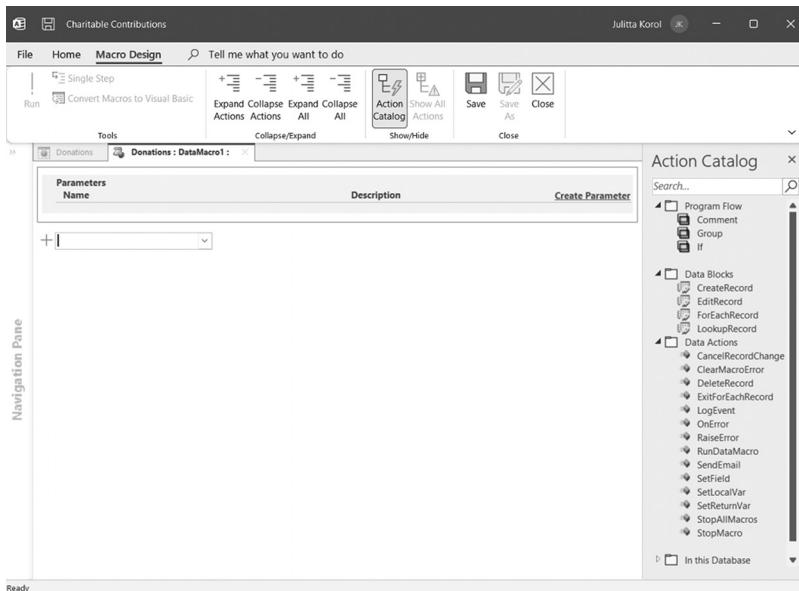


FIGURE 26.35 The Macro Builder window for creating a named data macro.

4. If you need to pass parameters to your macro, click the **Create Parameter** hyperlink at the top of the Macro Builder screen. Enter the name of the parameter in the Name box. You may also enter a description in the Description box (Figure 26.36).



FIGURE 26.36 Specifying parameters in the named data macro.

5. Select an appropriate action from the Add New Action drop-down box to specify your macro logic. Figure 26.37 shows the completed named data macro.
6. When you are done with the macro logic, save the macro by clicking the Save As button on the Ribbon.

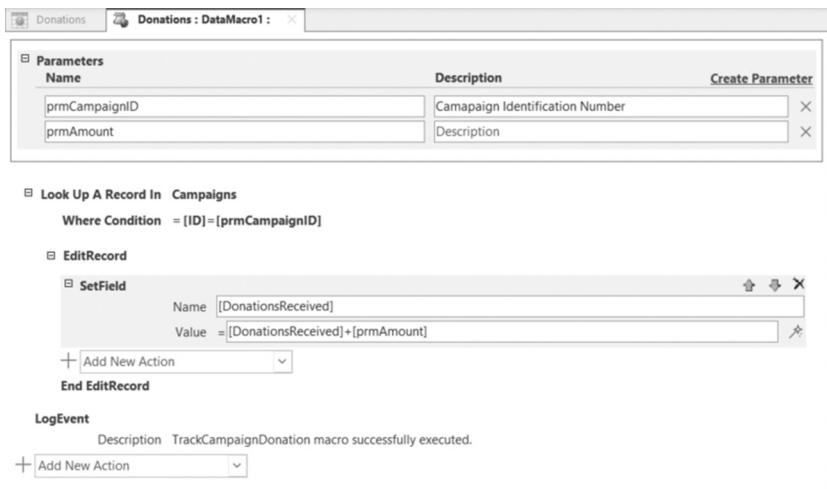


FIGURE 26.37 The completed named data macro.

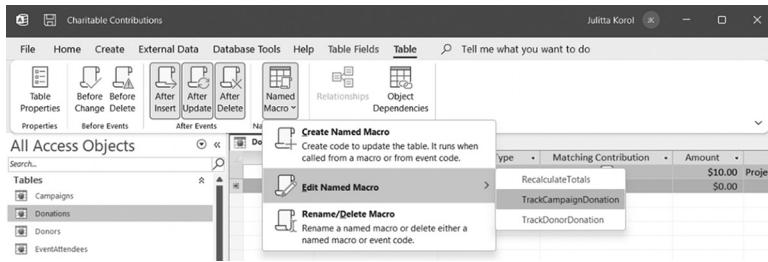
**NOTE**

The named data macro depicted in Figure 26.37 is available in the Charitable Contributions database created from the template supplied with an earlier version of Access and, for your convenience, included in the companion files. Follow these steps to open the database:

1. Copy the Charitable Contributions Web Database.accdb file from the companion files to your C:\VBAAccess2021\_ByExample folder.
2. Double click the copied file to open it in Access.
3. In the Login window, click the **New User** hyperlink.
4. In the User Details window, enter your name in the Full Name text box, and click **Save & Close**.
5. Select your name in the Login window and click **Login**.
6. When prompted, click the **Enable Content** button in the message bar. This will activate the Login window. Select your name and click **Login**.
7. Open the Navigation pane and double-click the **Donations** table.
8. Click the **Table** tab and select **Named Macros | Edit Named Macro | TrackCampaignDonation** to view the data macro and access the named macros in this table.

### ***Editing an Existing Named Macro***

You can edit an existing named data macro by clicking the Named Macro button on the Ribbon and selecting Edit Named Macro. Access will display the list of available macros as shown in Figure 26.38.

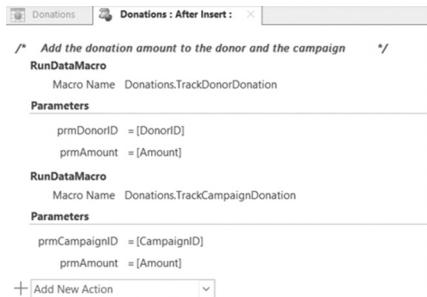


**FIGURE 26.38** If the table contains named data macros, the macro names are listed under Edit Named Macro option.

### ***Calling a Named Macro from Another Macro***

You can run a named macro from another macro using the RunDataMacro action. Figure 26.39 shows two named data macros that are run from within the After Insert data macro in the Donations table. Notice that to run a named macro you need to:

- Specify the RunDataMacro action.
- Specify the named macro name (Donations.TrackDonorDonation, Donations.TrackCampaignDonation).
- Specify the values for the parameters that the named data macro expects.

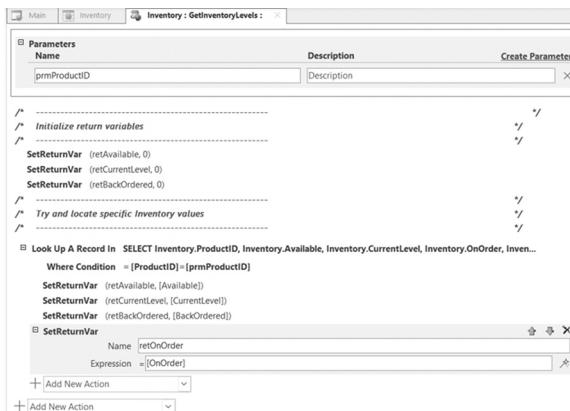


**FIGURE 26.39** Running named macros from the After Insert data macro in the Donations table.

### ***Using ReturnVars in Data Macros***

A powerful feature in data macros is their ability to return values to other macros by using `ReturnVars`. `ReturnVars` can be compared to values returned by

functions in VBA procedures. You can specify the `ReturnVars` by using the `SetReturnVar` action in a named data macro as depicted in Figure 26.40. After selecting the `SetReturnVar` macro action from the Add New Action dropdown box, enter the name of the `ReturnVar` in the Name box and specify the value or expression in the Expression box. For example, to return the number of backordered inventory items, the example macro in Figure 26.40 sets up a `ReturnVar` named `retBackOrdered`, and sets its value in the Expression box to `[BackOrdered]`, which is the name of the field in the `Inventory` table. The number of the backordered items will be returned by the `LookupRecord` macro action for the specified `ProductID`. Notice that all return variables are initialized at the top of the macro.



**FIGURE 26.40** The named data macro `GetInventoryLevels` located in the `Inventory` table of the Northwind Web database created from a template generated by an earlier version of Access demonstrates the use of return variables. You can open this database from the companion files.

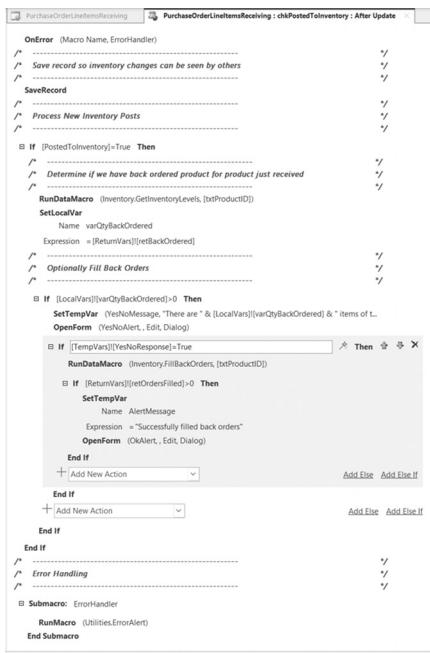
#### NOTE

Access 2010 introduced a new type of database file known as Access Web Database. You could use Access Web Databases to publish your Access data to a Microsoft SharePoint server running Access Services. Once published, your database could be used in an Internet browser. Because Access Web Database is not compatible with VBA, all programming had to be done using macros. In Access 2010, to design an Access Web Application, you had to choose File | New and click Blank Web Database. Well, this option is not available in Access 2016-2021. Simply put, Microsoft has retired the Web Apps. While you can open, design, and publish existing Access 2010 Web databases in Access 2016-2021, it is no longer possible to create new Access Web databases.

To get the return value, you must first call the macro. The GetInventoryLevels macro (shown in Figure 26.40) is called from the embedded macro (see Figure 26.41) that is attached to the After Update event of chkPostedToInventory checkbox control. This control is located on the PurchaseOrderLineItemsReceiving form in the Northwind Web database.

Notice that to reference the return variable in a macro, you must use the `ReturnVars` command like this:

```
= [ReturnVars]![retBackOrdered]
```



**FIGURE 26.41** Referencing return variables (`ReturnVars`) inside a macro attached to the After Update event of a control placed on a form. Notice that the value of the return variable is being retrieved into a local variable named `varQtyBackOrdered`.

### Tracing Data Macro Execution Errors

Access automatically writes all errors encountered during execution of your data macros in a system table called `USysApplicationLog`. Any failure that occurs while executing a named data macro or a data macro attached to an event will be reported in this table. By default, the `USysApplicationLog` table is created the first time Access encounters a data macro error. There are a couple of ways to access this table:

- Using the Backstage View (see Figure 26.42).

If the UsysApplicationLog table is present in your database, select the **File** tab, and click the **View Application Log Table** button to open the table.

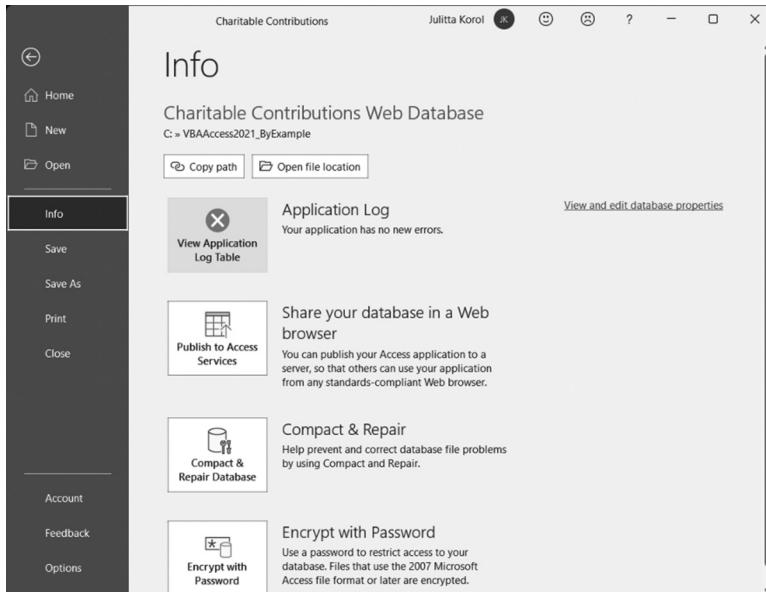


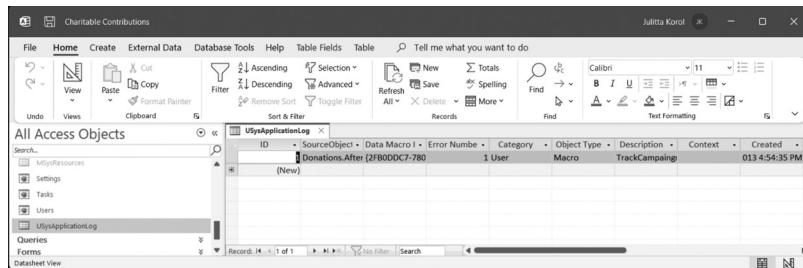
FIGURE 26.42 Accessing the UsysApplicationLog table in the Backstage View.

- Using the Navigation pane (see Figure 26.43).

Before you can access the UsysApplicationLog table from the Navigation pane you must tell Access to display system objects. To do this, select **File | Options**, and click **Current Database**. Scroll down to the Navigation section and click the **Navigation Options** button. Select the **Show System Objects** box at the bottom of the Navigation Options dialog box and click **OK**.

You can use UsysApplicationLog to view the details of errors that occurred during data macro execution. Access provides a special action called LogEvent that allows you to write your own messages to the log table. You can keep track of the data macros that ran by adding the LogEvent action to the end of your named macro and setting its Description field to whatever message you want to write (see Figure 26.37 earlier).

Figure 26.43 displays the contents of the UsysApplicationLog table after adding data to a donations table.



**FIGURE 26.43** Viewing the contents of the UsysApplicationLog table.

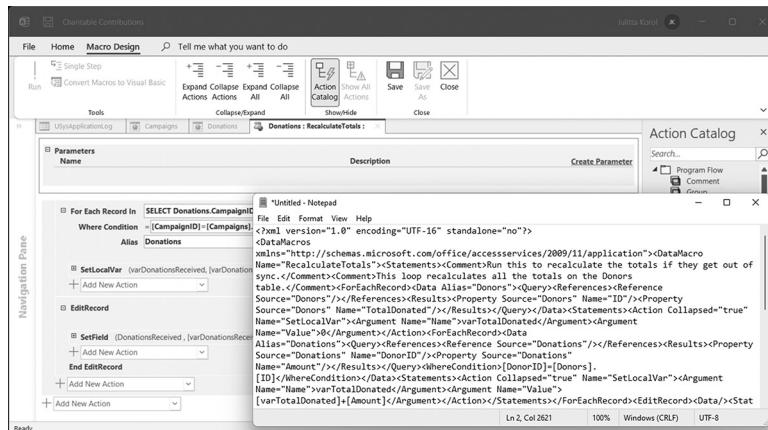
## SIDE BAR *Copying Macros*

Access stores macros as XML. Saving your macro as XML enables you to email it to someone else or create a backup copy of your macro before attempting to edit its logic.

You can copy the XML markup of your data macro to a text editor using these steps:

1. Open the macro and select the action you'd like to copy. A gray box appears around the selected action. To select all actions, press **Ctrl+A**.
  2. Right-click the selected area and choose **Copy**.
  3. Open **Notepad** and choose **Edit | Paste**.

Figure 26.44 shows the RecalculateTotals data macro copied to Windows Notepad.



**FIGURE 26.44** Copying the macro content to a text file.

## Error Handling in Macros

---

Access provides special macro actions that give macros the capability to handle errors: OnError, ClearMacroError, and SingleStep. A MacroError object provides you with information about the error received and allows you to create user-friendly error messages. The OnError action is similar to the `On Error` statement in VBA. This action specifies how errors should be handled when a runtime error occurs. The OnError action has two arguments, as shown in Table 26.2.

TABLE 26.2 OnError action arguments

Arguments	Description
Go To (This argument is required.)	Specifies how macros should handle errors. The Go To argument can be set to: Next, Macro Name, or Fail. Next—The error is recorded in the MacroError object and the execution of the macro moves to the next macro action. This is similar to the <code>On Error Resume Next</code> statement in VBA. Macro Name—Macro execution is passed to the macro that is named in the Macro Name argument. This is similar to the <code>On Error GoTo</code> statement in VBA. Fail—Access will stop the execution of the macro and display an error. This is similar to <code>On Error GoTo 0</code> in VBA.
Macro Name (This argument is optional.)	If the Go To argument is set to Macro Name, the name of the macro in the current macro group will handle the error.

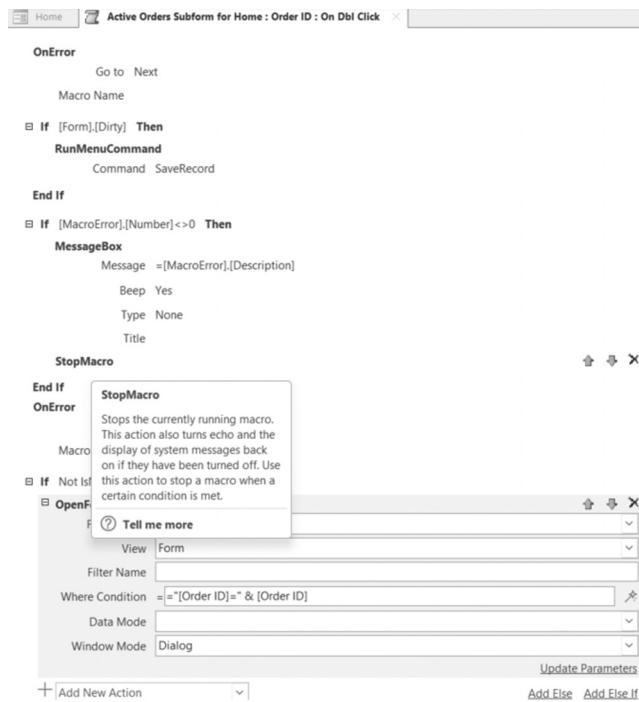
The OnError action suppresses standard error messages displayed by Access when an error occurs. When you use this action in your macro, you should use the error information saved in the MacroError object to display a user-friendly message about the error.

The MacroError object has the following properties: ActionName, Arguments, Condition, Description, MacroName, and Number. You can check the MacroError object's Number property to find out if an error occurred, as shown in Figure 26.44. If there was no error, the Number property will return zero (0). However, if `[MacroError]. [Number] <> 0`, then you should handle the error right away.

By default, the MacroError object is cleared at the end of the macro execution; however, you can clear it right after the error has been handled by using the ClearMacroError action. This action will reset the error number in the MacroError object back to zero and clear other information stored in the object such as macro name, action name, condition, arguments, and description. The MacroError object contains information about only one error at a time; if more

than one error occurred, only the error information about the last error can be retrieved. Therefore, when writing longer macros use the ClearMacroError action right after handling the first error so the `ErrorObject` will be able to capture information about the next error that might occur.

Use the StopMacro action to stop the currently running macro. In Figure 26.45, the StopMacro action is run right after the user receives the message about the macro error.



**FIGURE 26.45** Error handling in an Access macro located in the Northwind 2007.accdb database.

To debug a macro that is not working properly, you can click the Single Step button in the Tools group of the Macro Design tab and then click the Run button, or you can use the SingleStep macro action just before an action that you suspect is causing a problem. This action pauses the macro and opens the Macro Single Step dialog box (see Figure 26.46), which displays information about the current macro action (macro name, condition, action name, arguments, and error number). The Macro Single Step dialog box contains the three buttons described in Table 26.3.

TABLE 26.3 Macro Single Step dialog box buttons

Button Name	Description
Step	Move to the next macro action.
Stop All Macros	Stop the current macro and any other macros that may be running.
Continue	Use this button to exit Single Step mode and continue the normal execution of the macro.

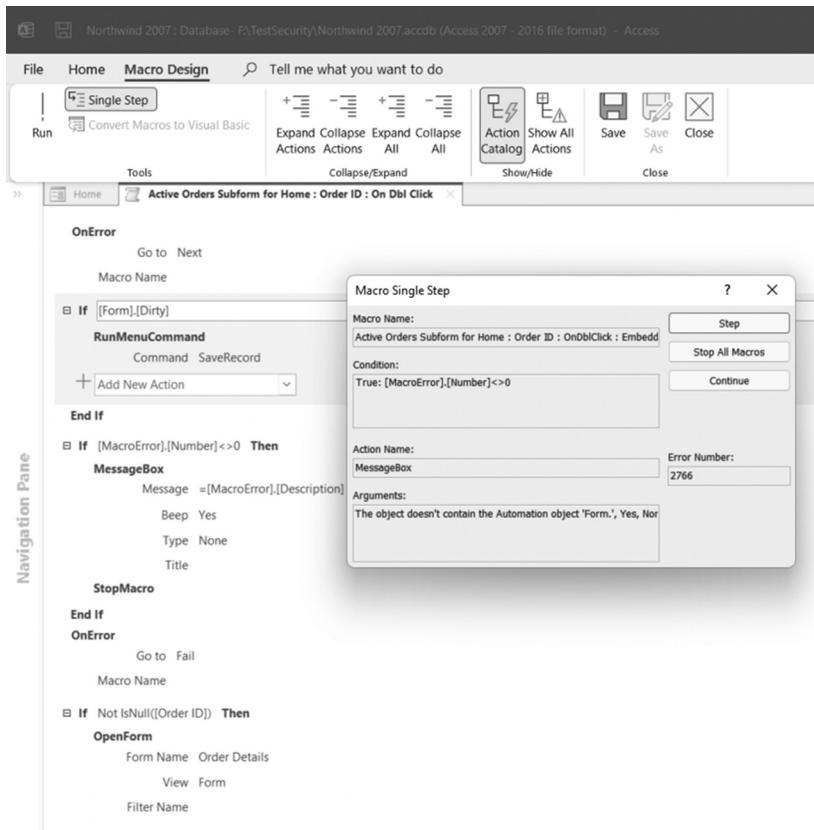


FIGURE 26.46 Debug your macros by selecting the Single Step option on the Ribbon and clicking the Run button.

**NOTE**

If you opened the Macro Single Step dialog box using the Single Step button on the Ribbon, you must click this button again when you are done debugging your macro or the next macros that you run will also be run using Single Step mode.

## Using Temporary Variables in Macros

---

The functionality to add temporary variables (TempVars) has been in Access since its 2007 release. This functionality applies to both VBA and macros. You've seen the VBA side of using the `TempVar` object in Chapter 3. Recall that the `TempVar` object of the `TempVars` collection allows you to get or set a value for a variable. Each `TempVar` object has a name and value property. In macros, there are three macro actions that relate to TempVars:

- **`SetTempVar(name, expression)`**—This macro action is used to create a new temporary variable. This variable can then be used as a condition or argument in subsequent macro actions. Temporary variables are global; therefore, you can use them in another macro, in an event procedure, or on a form or report. The first argument of the `SetTempVar` macro action assigns a name to the temporary variable. The second argument is the expression that Access should use to set the value for this temporary variable. You can define up to 255 temporary variables at one time.
- **`RemoveTempVar(name)`**—This macro action is used to remove the temporary variable. Use the `name` argument to provide the name of the variable to remove. It is recommended that you remove the temporary variable once you've finished working with it. If you don't remove your temporary variables, they will be removed automatically when you close the database.
- **`RemoveAllTempVar`**—This macro action is used to remove all temporary variables from the `TempVars` collection.

Figure 26.47 shows how to specify the name of a report by using a temporary variable.

<b>NOTE</b>	<i>Because both macros and VBA use the same TempVars collection, it is easy to share data between your macros and VBA procedures.</i>
-------------	---



FIGURE 26.47 Using temporary variables in an Access macro.

## Converting Macros to VBA Code

The ability to convert standalone macros to VBA code has been available since Access 97. With the introduction of embedded macros, Access also provides a button to convert to VBA code macros stored in an event property of a form, report, or control (see Figure 26.48).

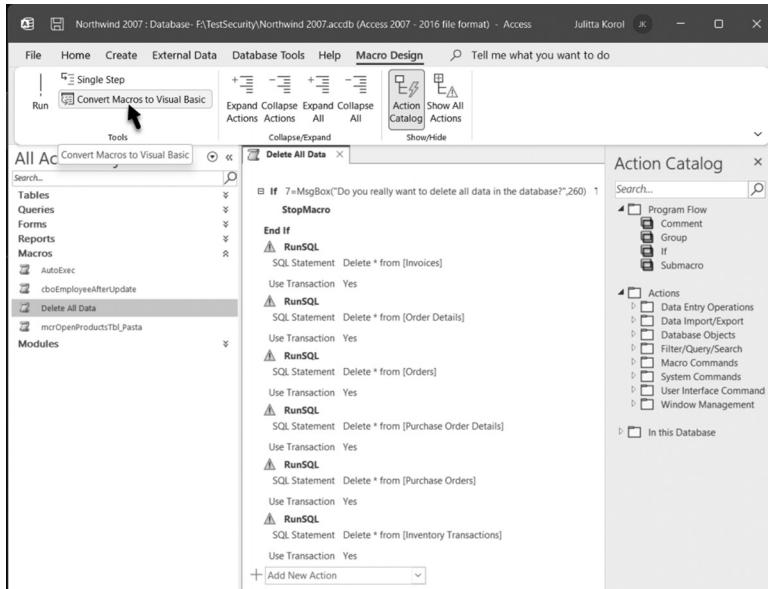
### Converting a Standalone Macro to VBA

To convert a standalone macro to VBA, follow these steps:

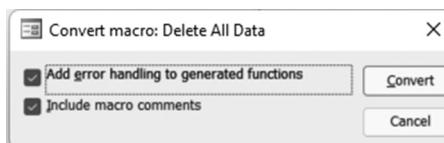
1. In the Navigation pane under Macros, right-click the macro you want to convert, then click **Design View**.
2. In the Tools group of the Design tab, click **Convert Macros to Visual Basic** (see Figure 26.48).

Access will display a dialog box asking whether you want to include error handling and comments in the code (see Figure 26.49). To keep your code very simple, you can clear both checkboxes.

**3.** Start the conversion process by clicking the **Convert** button.

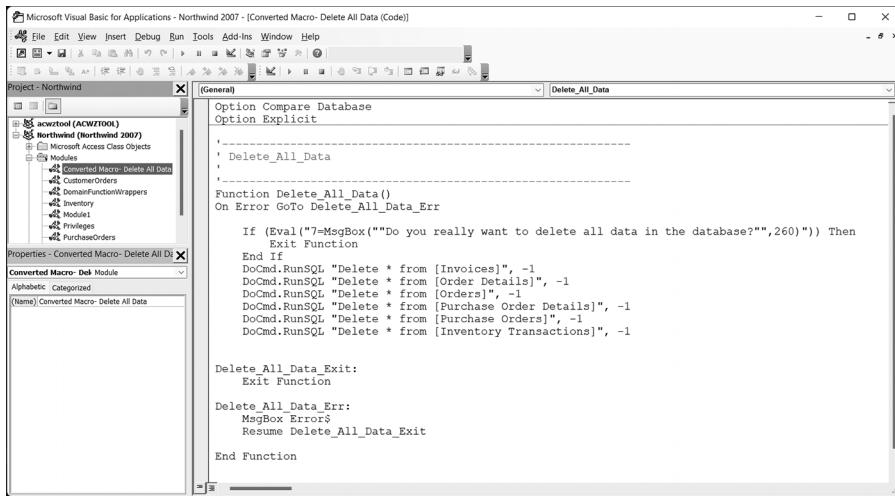


**FIGURE 26.48** Click the Convert Macro to Visual Basic button to convert a standalone macro to Visual Basic for Applications code.



**FIGURE 26.49** Access displays this dialog box when you click the Convert Macros to Visual Basic button (see Figure 26.48).

Upon completion of the macro conversion process, Access displays a message stating that the conversion is finished. Click OK to the message and review the Modules group in the Navigation pane. You should see a separate module for the converted macro. The name of the module is Converted Macro followed by a dash and the name of the macro you converted. For example, after converting the Delete All Data macro, the name of the VBA module is Converted Macro – Delete All Data. To view the converted macro, double-click the converted module's name. This will open the Visual Basic Editor window, as shown in Figure 26.50.



**FIGURE 26.50** This VBA code was generated by Access from a standalone macro.

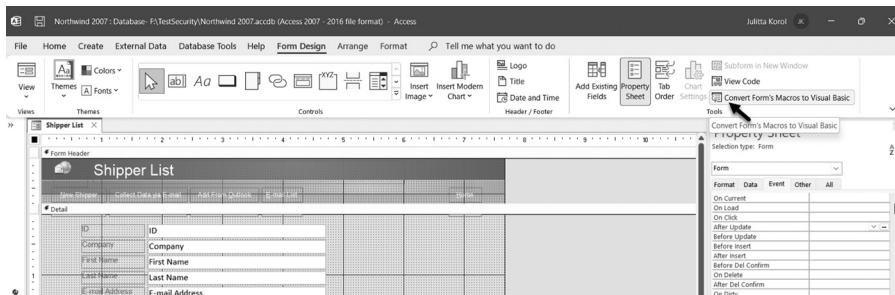
---

**NOTE**

*You can modify the code generated by the macro conversion process to suit your needs.*

*Converting Embedded Macros to VBA*

To convert embedded macros, open the form or report in Design view. You should see the button named Convert Form's Macros to Visual Basic in the Macro group, as shown in Figure 26.51.

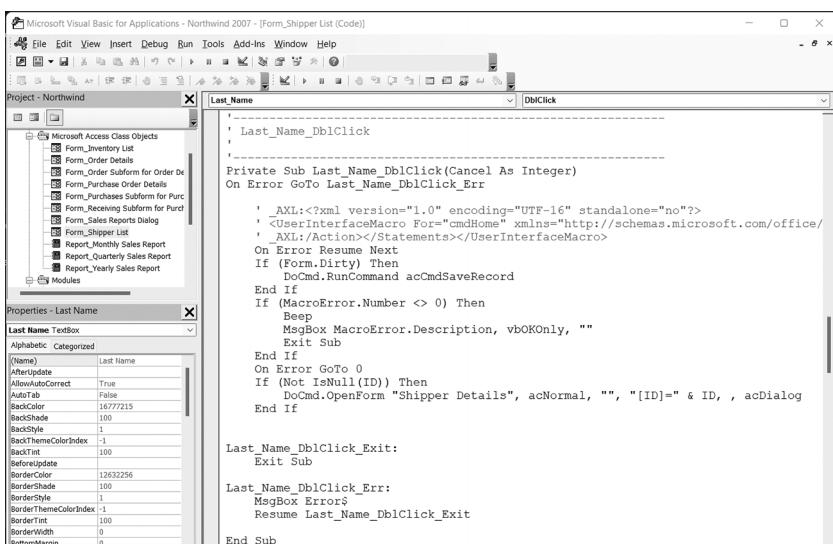


**FIGURE 26.51** Converting embedded macros to VBA code using the Convert Form's Macros to Visual Basic button.

After clicking the Convert Form's Macros to Visual Basic button, Access displays the same dialog box shown earlier in the conversion process for standalone macros (see Figure 26.49). When you click the Convert button, Access begins the conversion process, and when this process completes you will see a message

about the successful completion of the conversion. Click OK to the message. Next, activate the property sheet, and notice that form and control event properties that have previously been set to [Embedded Macro] now display [Event Procedure]. You can click the Build button (...) to view the VBA code. Figure 26.52 shows the VBA procedure that was generated for the embedded macro attached to the DoubleClick event of the Last Name text box control placed on a form. Notice that for each converted macro, Access writes its equivalent XML macro code as a comment at the top of the VBA procedure.

<b>NOTE</b>	<p><i>If after the conversion process you still want to keep the [Embedded Macro] setting in the event properties of a form, report, or control, perform these steps:</i></p> <ol style="list-style-type: none"> <li><i>1. Save the VBA code generated by the macro conversion to a file by choosing <b>File   Export File</b> in the Visual Basic Editor window. Access will create a file with the .cls extension.</i></li> <li><i>2. To view the contents of this file, right-click its name in Windows Explorer and choose <b>Open With   Choose Program</b>. Select <b>Notepad</b> and click <b>OK</b>.</i></li> <li><i>3. In Access, close the form and answer <b>No</b> when prompted for changes. Access will revert the [Event Procedure] setting in the event properties to [Embedded Macro].</i></li> </ol>
-------------	--



```

Microsoft Visual Basic for Applications - Northwind 2007 - [Form_Shipper List (Code)]
File Edit View Insert Debug Run Tools Add-Ins Window Help
Project - Northwind
  Microsoft Access Class Objects
    Form_Inventory List
    Form_Order Details
    Form_Order Details for Order De
    Form_Purchase Order Details
    Form_Purchases Subform for Purch
    Form_Receiving Subform for Purch
    Form_Sales Reports Dialog
    Form_Shipper List
    Report_Monthly Sales Report
    Report_Quarterly Sales Report
    Report_Yearly Sales Report
  Modules
Properties - Last Name
  Last Name TextBox
    Last Name
    AfterUpdate
    AllowAutoCorrect
    AllowTab
    BackColor
    BackColorIndex
    BackStyle
    BackThemeColorIndex
    BackTint
    BorderColor
    BorderColorIndex
    BorderDash
    BorderDashIndex
    BorderStyle
    BorderTint
    BorderThemeColorIndex
    BorderWidth
    BottomMargin
  End Sub

Private Sub Last_Name_DblClick(Cancel As Integer)
On Error GoTo Last_Name_DblClick_Err
  ' AXI:<xml version="1.0" encoding="UTF-16" standalone="no"?>
  '<UserInterfaceMacro Form="cmdHome" xmlns="http://schemas.microsoft.com/office/
  '<AXI:</Action></Statements></UserInterfaceMacro>
  On Error Resume Next
  If (Form.Dirty) Then
    DoCmd.RunCommand acCmdSaveRecord
  End If
  If (MacroError.Number <> 0) Then
    Beep
    MsgBox MacroError.Description, vbOKOnly, ""
    Exit Sub
  End If
  On Error GoTo 0
  If (Not IsNull(ID)) Then
    DoCmd.OpenForm "Shipper Details", acNormal, "", "[ID]="" & ID, , acDialog
  End If

Last_Name_DblClick_Exit:
  Exit Sub

Last_Name_DblClick_Err:
  MsgBox Errors
  Resume Last_Name_DblClick_Exit
End Sub

```

FIGURE 26.52 VBA code from a converted embedded macro.

## ACCESS TEMPLATES

---

Access comes with several prebuilt templates that give users a head start with various types of projects. In Access 2021, the templates listed on the startup screen have built-in tables, queries, forms, and reports in various subject categories. The template files can be easily recognized by their .accdt file extension. By default, Access stores the template files in the *C:\Users\username\AppData\Roaming\Microsoft\Templates* folder. Please note that AppData is a hidden folder and you will need to unhide it in the File Explorer in order to access its content.

### **Creating a Custom Blank Database Template**

---

When you select the Blank database button in the Backstage View (File | New) and click the Create button, Access provides you with an empty database that you can customize to suit your specific needs. If you are like many users, you start your next database project by again clicking the Blank database button and proceed to implement many of the same customizations that you applied to the previous database project. If you have been working like this, however, you are not taking advantage of the startup template—Blank.accdb. Instead of customizing each new blank database, simply create a new database called Blank.accdb in the template folder and customize it to include specific database properties, VBA references and custom functions, Ribbon customizations, default forms and reports, and customized controls, as well as any other special configuration settings that you normally use in your database applications. The next time you click the Blank database button in the Backstage View, Access will make a copy of your Blank.accdb database so you won't need to start from scratch. Your new database will already contain the common settings that you saved in the Blank.accdb file. Moreover, if your database requirements have changed, you can create a new Blank.accdb database with settings that conform to these new requirements.

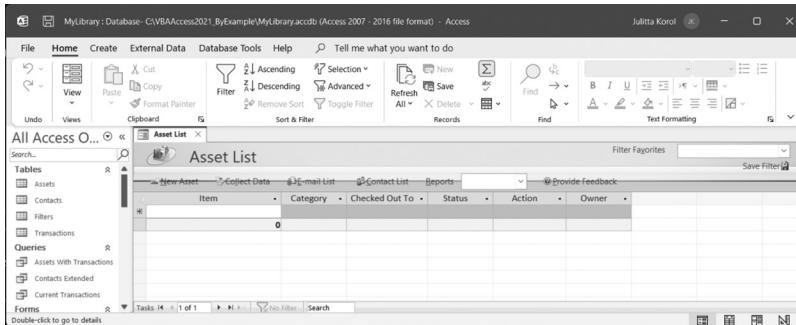
### **Understanding the .accdt File Format**

---

The .accdt file format that Access 2021 uses for its database templates is based on the Microsoft Office Open Packaging Convention (.opc) file format. This file format is based on the XML and ZIP archive technologies. The .opc file format is also used by the .docx, .xlsx, and .pptx file formats first introduced in Office 2007 for Word, Excel, and PowerPoint. The .opc format makes it possible to store a number of text, image, and .xml/.xsd files in a single compressed file. The .opc files can be easily opened and examined. Before you can open an Access

template file (.accdt) and examine its structure, you need to add the .zip extension at the end of the filename as shown in the following steps:

1. Launch Access 2021. On the startup screen, type **lending library** in the search box and press Enter to begin searching the online content. You must be connected to the Internet to make it work. When Access displays the templates that matched your search criteria, click the one named Lending library. Enter the name of the database as **MyLibrary.accdb** and change the folder to **C:\VBAAccess2021\_ByExample**, and then click the **Create** button. Access downloads the template file and creates the specified desktop database. The resulting database is shown in Figure 26.53.



**FIGURE 26.53** The MyLibrary database is based on the Lending library template downloaded from the Microsoft Access templates archive.

2. Close the MyLibrary.accdb database file and exit Access.
3. Locate the downloaded Lending library.accdt template file. By default, templates are stored in your `\Users\username\AppData\Roaming\Microsoft\Templates` folder.
4. Rename the file **Lending library.accdt.zip**, as shown in Figure 26.54.
5. When the Rename dialog box appears, click **Yes** to confirm that you want to change the filename extension. Click **Continue**, if prompted to provide administrator permission to rename this file. The file format should now change to the zip archive.

Name	Date modified	Type	Size
Lending library.accdt.zip	1/25/2022 7:48 PM	Compressed (zipped) Folder	879 KB
Charitable contributions.accdt	1/24/2022 9:48 PM	Microsoft Access Template	643 KB
Northwind.accdt	10/26/2021 8:05 PM	Microsoft Access Template	2,007 KB

**FIGURE 26.54** By adding the zip file extension to the accdt file format you can turn it into a zip archive that you can examine and modify depending on your needs.

6. To open the Lending library.accdt.zip file, right-click the filename and choose **Open With | Compressed (zipped) Folders** or **File Explorer**, or simply **Open**, if you're using Windows 11. The folders that make up the document are shown in Windows Explorer (see Figure 26.55).

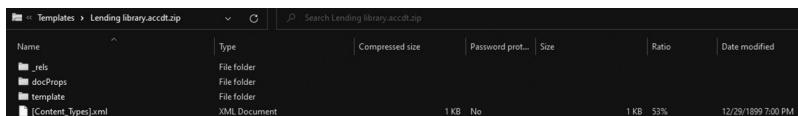


FIGURE 26.55 The directory structure of an Access template file.

Notice that the archive file contains the following three folders: `_rels`, `docProps`, and `template`. The `_rels` folder contains one `.xml` file with the extension `.rels` that defines the relationships between various files included in the file package (see Figure 26.56). Access uses this file to find out information about the template and the database.

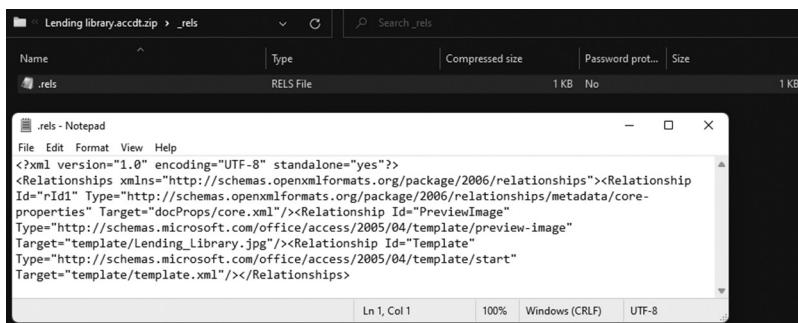


FIGURE 26.56 The contents of the `.rels` `.xml` file in the `_rels` folder.

The `docProps` folder contains the `core.xml` file that describes the core document properties such as creator name, identifier, title, description, keywords, category, version, and lastModifiedBy (see Figure 26.57).

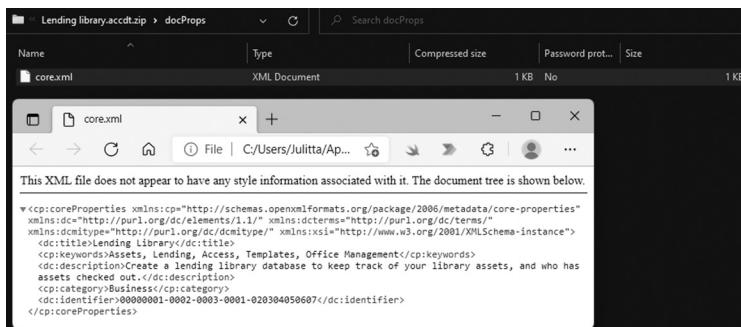


FIGURE 26.57 The contents of the `core.xml` file in the `docProps` folder.

When you double-click the template folder, you will see two subfolders named \_rels and database, as well as a template.xml file. The template.xml file contains information about the format of the template file (Figure 26.58).

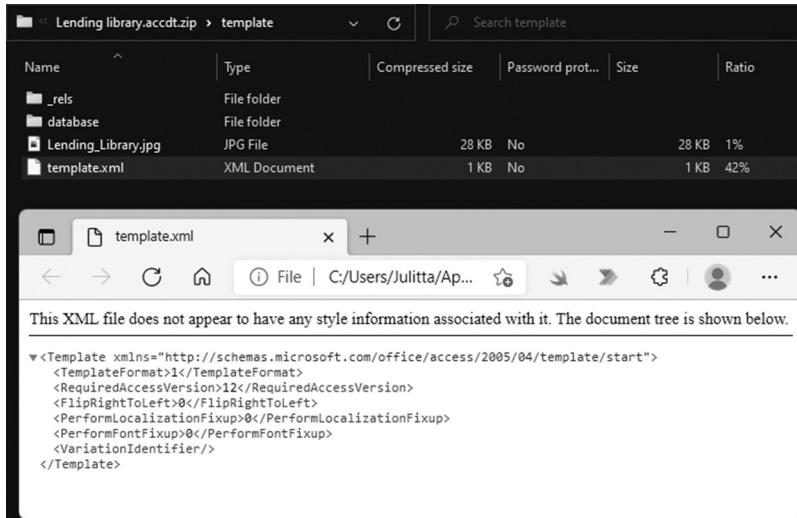


FIGURE 26.58 The contents of the template folder and the Template.xml file.

You can find out a lot of information about the contents and structure of the .accdt file by opening the database folder (Figure 26.59).

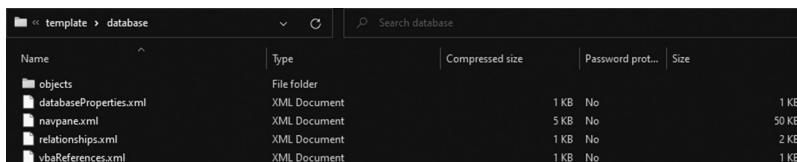


FIGURE 26.59 The contents of the database folder.

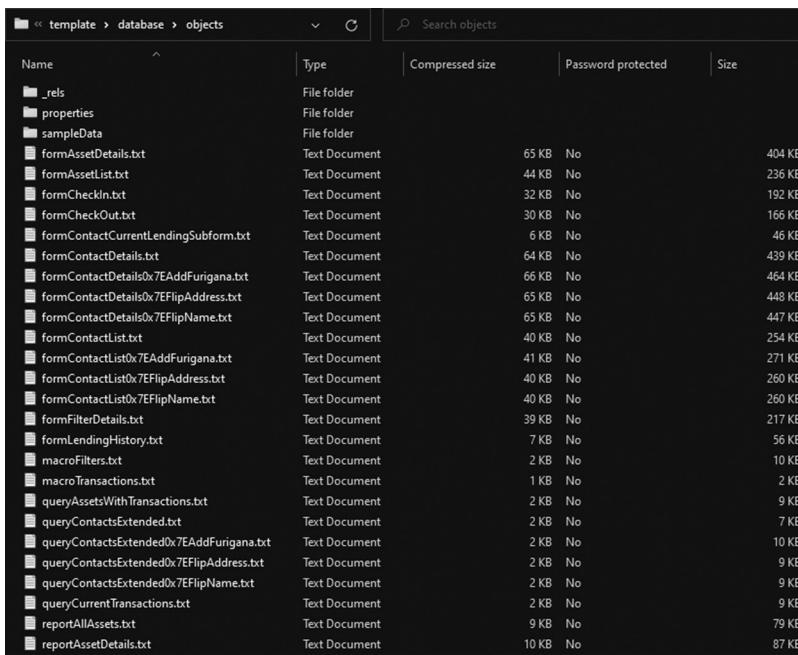
The databaseProperties.xml file in the database folder stores various database settings and properties. You can modify this file to include additional properties that need to be set by adding new nodes to the file.

The navpane.xml file contains information about the structure of the Navigation pane. It also contains the data for the Navigation pane system tables: MSysNavPaneGroupCategories, MSysNavPaneGroups, MSysNavPaneGroupToObjects, and MSysNavPaneObjectIDs.

The relationships.xml file contains the contents of the MSysRelationships system table.

The vbaReferences.xml file contains all VBA project references that Access needs to set.

In the objects folder you will find many other files that describe different database objects (Figure 26.60).



The screenshot shows the Microsoft Access 'Objects' browser window. The path in the top navigation bar is 'template > database > objects'. A search bar at the top right contains the placeholder 'Search objects'. The main area is a table listing files. The columns are 'Name', 'Type', 'Compressed size', 'Password protected', and 'Size'. The table lists numerous files, mostly 'Text Document' type, with sizes ranging from 2 KB to 448 KB. Some files have descriptive names like 'formAssetDetails.txt', 'macroFilters.txt', and 'queryAssetsWithTransactions.txt'.

Name	Type	Compressed size	Password protected	Size
_rels	File folder			
properties	File folder			
sampleData	File folder			
formAssetDetails.txt	Text Document	65 KB	No	404 KB
formAssetList.txt	Text Document	44 KB	No	236 KB
formCheckIn.txt	Text Document	32 KB	No	192 KB
formCheckOut.txt	Text Document	30 KB	No	166 KB
formContactCurrentLendingSubform.txt	Text Document	6 KB	No	46 KB
formContactDetails.txt	Text Document	64 KB	No	439 KB
formContactDetails0x7EAddFurigana.txt	Text Document	66 KB	No	464 KB
formContactDetails0x7EFlipAddress.txt	Text Document	65 KB	No	448 KB
formContactDetails0x7EFlipName.txt	Text Document	65 KB	No	447 KB
formContactList.txt	Text Document	40 KB	No	254 KB
formContactList0x7EAddFurigana.txt	Text Document	41 KB	No	271 KB
formContactList0x7EFlipAddress.txt	Text Document	40 KB	No	260 KB
formContactList0x7EFlipName.txt	Text Document	40 KB	No	260 KB
formFilterDetails.txt	Text Document	39 KB	No	217 KB
formLendingHistory.txt	Text Document	7 KB	No	56 KB
macroFilters.txt	Text Document	2 KB	No	10 KB
macroTransactions.txt	Text Document	1 KB	No	2 KB
queryAssetsWithTransactions.txt	Text Document	2 KB	No	9 KB
queryContactsExtended.txt	Text Document	2 KB	No	7 KB
queryContactsExtended0x7EAddFurigana.txt	Text Document	2 KB	No	10 KB
queryContactsExtended0x7EFlipAddress.txt	Text Document	2 KB	No	9 KB
queryContactsExtended0x7EFlipName.txt	Text Document	2 KB	No	9 KB
queryCurrentTransactions.txt	Text Document	2 KB	No	9 KB
reportAllAssets.txt	Text Document	9 KB	No	79 KB
reportAssetDetails.txt	Text Document	10 KB	No	87 KB

**FIGURE 26.60** Files in the objects folder contain information about different database objects in the template file as well as information about sample data and properties for each object included in the template.

You can open any of the files listed in Figure 26.60 and examine the type of information being stored. Because this chapter covered macros, look at the macroFilters.txt file to find out how Access stores the embedded macros.

#### NOTE

*When you are done reviewing the files, change the name of the Lending library.accdt.zip file back to its original name—Lending Library.accdt.*

## SUMMARY

This chapter introduced you to working with macros in Access 2021. You learned about macro security; created standalone and embedded macros; worked with data macros; saw examples of return variables (ReturnVars), local variables, and temporary variables (TempVars), and examined the error-handling actions in macros. You also learned how standalone and embedded macros can be converted to Visual Basic code. Because Access uses embedded macros extensively in its templates, we examined the structure and contents of the .accdt file format.

This chapter concludes Part VIII of the book, which focused on getting familiar with macro interface and template file structure in Access 2021. In Part IX, you learn how to use your Access VBA skills with the Extensible Markup Language (XML) and Web Services.

# Part IX

## *WORKING TOGETHER: VBA, XML, AND RESTAPI*

**E**xtensible Markup Language (XML) has long been the standard format for sharing data without regard for the originating application or the operating system. In this part of the book, you learn how XML is used in Access to bring external data to your database as well as provide your data to other applications. You also learn about the Rest APIs, the newest and the most flexible method of integrating applications. You will use your Access VBA and XML skills to make HTTP requests to a Web server to retrieve data and integrate it with Access. In this process, you are introduced to using JSON (JavaScript Object Notation), the most popular file format for storing and transporting data.

Chapter 27 XML Features in Access 2021

Chapter 28 Access and REST API



# Chapter 27 *XML FEATURES IN ACCESS 2021*

If you need to store, share, and exchange data between different applications regardless of the operating system or programming language used, you need to become familiar with Extensible Markup Language (XML). Imagine these two scenarios where your combined knowledge of Access and XML will come in handy:

- You have just received an XML file, and you need to merge its data with an existing Access table, or perhaps create a one or more Access tables based on the contents of that file.
- You have been asked to provide a data dump from your Access database in XML file format.

XML is a complex language that cannot be covered in details within the pages of one chapter; however, this chapter will get you started using XML with Access 2021.

## XML AND ACCESS

XML is not a new feature in Access 2021. Access 2002 was the first version to support XML files. In 2003 many new features were added to Access so it could easily export data into an XML document and create multiple tables from a single XML document. With XML features added throughout various versions, Access can easily share relational data with other applications.

XML is a platform-independent markup language used for creation of various structured documents. These documents are simply text files that use tags (such as `<order></order>`, `<item></item>`, `<unitPrice></unitPrice>` and so on) to describe data content placed within those tags. While HTML (Hypertext Markup Language) uses fixed, non-customizable tags to provide formatting instructions that should be applied to the data, XML is *extensible*, which means that it is not restricted to a set of predefined tags. XML allows you to invent your own tags in order to define and describe data stored in a wide range of documents. The XML parser does not care what tags you use; it only needs to be able to find the tags and confirm that the XML document is well formed. A document that follows the formatting rules for XML is considered a well-formed document. A *parser* is a software engine, usually a dynamic-link library (DLL), that can read and extract data from XML. All current browsers have built-in XML parsers that can read well-formed documents and detect those that are not.

In addition to being well formed, an XML document must also be valid. When a document is *valid*, it follows the predefined rules for valid data. These rules are defined in a Document Type Definition (DTD) or a schema file, which is written in XML. DTD is an older method of data validation. The XML Schema Definition files are easily recognized by an .XSD file extension. These files contain information about the structure of the XML tags, data types and constraints. You can have Access generate these files automatically for you when you export your database data into XML documents. Later in this chapter you will see how Access uses a schema to determine the types of elements and attributes an XML document should contain, how these elements and attributes should be named, whether they're optional or required, their data types and default values, and the relationship between the elements. Another file type you will be working with will have an .XSL extension (don't confuse it with the older Excel XLS file format). The XSL is an abbreviation for an Extensible Stylesheet Language that defines how the XML data is supposed to look. You will have a chance to apply a stylesheet to a raw XML document to make it look attractive to an end user. A part of the Extensible Stylesheet Language are transformations

(XSLT). You will use XSLT later in this chapter to define the structure of your XML data.

Because of its extensibility, XML makes it easy to describe any data structure and send it anywhere across the Web using common protocols such as HTTP (Hypertext Transfer Protocol) or FTP (File Transfer Protocol). Although XML was designed specifically for delivering information over the World Wide Web, it is being utilized in other areas, such as storing, sharing, and exchanging data. Since XML is stored in plain text files, it can be read by many types of applications, independent of the operating system or hardware.

### What Is a Well-Formed XML Document?

---

An XML document must have one root element. While in an HTML document the root element is always `<html>`, in an XML document you can name your root element anything you want. Element names must begin with a letter or underscore character. The root element must enclose all other elements, and elements must be properly nested. The XML data must be hierarchical; the beginning and ending tags cannot overlap. Let's look at the following XML document that describes a list of books:

```
<books>
  <book>
    <title>Microsoft Access 2021 Programming by Example</title>
    <ISBN>978-1-68392-841-6</ISBN>
    <author>Julitta Korol</author>
    <publisher>Mercury Learning and Information</publisher>
    <yearPublished>2021</yearPublished>
  </book>
  <book>
    <title>
      Microsoft Excel 2019 Programming by Example with VBA, XML and ASP
    </title>
    <ISBN>978-1-68392-400-5</ISBN>
    <author>Julitta Korol</author>
    <publisher> Mercury Learning and Information </publisher>
    <yearPublished>2021</yearPublished>
  </book>
</books>
```

Notice how the `<books>` tag describes the entire document. In the above example, `<books>` tag is a root element. It tells people that this document contains information about books. The name of the root tag can be anything you want but by choosing the name that makes sense, the XML document becomes easy to read even for people who are not familiar with its file structure. At the end of

the document, you will find an ending tag </books>. Make sure that all element tags are closed (a beginning tag must be followed by an ending tag). Let's look at another example below:

```
<Sessions>5</Sessions>
```

You can use shortcuts, such as a single slash (/), to end the tag so you don't have to type the full tag name. For example, if the current <Sessions> element is empty (does not have a value), you could use the following tag:

```
<Sessions />
```

Tag names are case-sensitive: The tags <Title> and </Title> aren't equivalent to <TITLE> and </TITLE>.

For example, the following line:

```
<title>Microsoft Access 2021 Programming by Example</title>
```

is not the same as:

```
<TITLE>Microsoft Access 2021 Programming by Example</TITLE>
```

XML documents can be element or attribute based. If you use attributes the books information can be presented as follows:

```
<books>
<book title="Microsoft Access 2021 Programming by Example"
      ISBN="978-1-68392-841-6" author="Julitta Korol" publisher = "Mer-
      cury Learning and Information" yearPublished=2021 />
<book title="Microsoft Excel 2019 Programming by Example with VBA,
      XML and ASP" ISBN="978-1-68392-400-5" author="Julitta Korol" pub-
      lisher = "Mercury Learning and Information" yearPublished=2021 />
</books>
```

Notice that all text attributes are inside quotation marks. You cannot have more than one attribute with the same name within the same element.

Two main goals of XML are the separation of content from presentation and data portability. It is important to understand that XML was designed to address the limitations of HTML and not to replace it. One of these limitations is the inability of HTML to identify data. By using XML tags you can give meaning to the data in the document and provide a consistent way of identifying each item of data. By separating content from presentation and structuring data based on its meaning, we are able to create documents that are easy to reuse, manipulate, and search.

## EXPORTING XML DATA

---

In Access you can export tables, queries, forms, and/or reports to XML files. There is no XML support for macros or modules. When you export a form or report, you actually export the data from the form or report's underlying table or query.

Access uses a special XML vocabulary known as *ReportML* for representing its objects as XML data. *ReportML* is an XML file that contains tags describing properties, methods, events, and attributes of the Access object being exported. This file is generated automatically by Access when you begin the export process and is used by Access to generate the final output files.

To allow XML data to be viewed in browsers in a user-friendly format, *ReportML* relies on a rather complicated stylesheet that contains formatting instructions. We examine stylesheets later in this chapter.

After the formatting instructions contained in the stylesheet have been applied to the XML file, the *ReportML* file is automatically deleted.

No matter what Access object you need to export to XML, you always follow the same procedure:

- Export all data
  - select the appropriate object (table, query, report, or form) in the database window
  - choose External Data | XML File or right-click the object name in the Navigation pane and select Export | XML File from the shortcut menu
- Export a single record or a filtered or sorted set of records
  - open the appropriate object and follow these steps:

You want to...	Step 1	Step 2
Export a single record	Select that record	Choose External Data   XML File, specify the name of the export file you want to create, and click OK. Click the More Options button and in the Records to Export area, select Current Record.
Export filtered records	Apply a filter to the records	Choose External Data   XML File and select the appropriate options.
Export records in a pre-defined order	Arrange records in the order you want	Choose External Data   XML File and select the appropriate options.

The following hands-on exercise demonstrates how to use the Export command to save the Shippers table in XML format.

All code files and figures for the hands-on projects may be found in the companion files.



### Hands-On 27.1 Exporting an Access Table to an XML File

1. Use File Explorer to create a new folder named C:\VBAAccess2021\_XML for this chapter's practice files.
2. Copy the sample **Northwind 2007.accdb** database the **VBAAccess2021\_XML** folder and open it. If you get a Security warning message that some active content has been disabled, click the **Enable Content** button.

<b>NOTE</b>	<i>As Microsoft continues to improve security in Office, the default behavior and banners displayed in Access and other Office applications may be different from those presented in the instructions and images included in this book. For the most recent guidelines, please see <a href="https://docs.microsoft.com/en-us/DeployOffice/security/internet-macros-blocked">https://docs.microsoft.com/en-us/DeployOffice/security/internet-macros-blocked</a>.</i>
-------------	---

3. Login as Andrew Cencini.
4. Add the **C:\VBAAccess2021\_XML** folder to your trusted locations. Refer to the section titled “Placing a database in a trusted location” in Chapter 1 for more details.
5. In the Access window’s Navigation pane, choose **Tables and Related Views**, and open the **Products** table. Click the **External Data** tab, and in the Export group of the Ribbon, click the More drop-down and choose **XML File**.
6. In the File name box, enter **C:\VBAAccess2021\_XML\Products.xml** and click **OK**.

In the Export XML dialog box that appears, there are three checkboxes (see Figure 27.1). The first one, will cause Access to generate an XML file containing the data from the Products table. The second one specifies that Access should create an XSD file with the data definition. The third checkbox tells Access to generate the stylesheet (XSL) file that will contain formatting specifications.

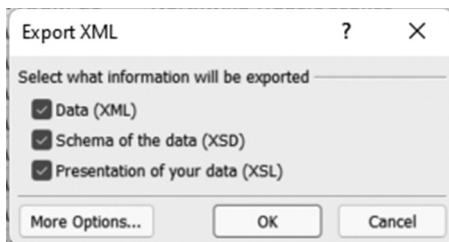


FIGURE 27.1. The Export XML dialog box displays three checkboxes; the first one is selected by default. The More Options button allows for more customizations.

7. Select all the checkboxes and click **OK** to proceed with the export.
8. When the export operation completes, Access displays the Export - XML File window where you are given a chance to save the export steps so that you can repeat them in the future without using the wizard.
9. Click **Close** to exit the Export - XML File window without saving the export steps.

Continue to the next Hands-On to examine the files generated by the XML export.

### **Understanding the XML Data File**

In Hands-On 27.1 you prepared the Products.xml file. Let's switch to File Explorer and examine the contents of your Access2021\_XML folder.



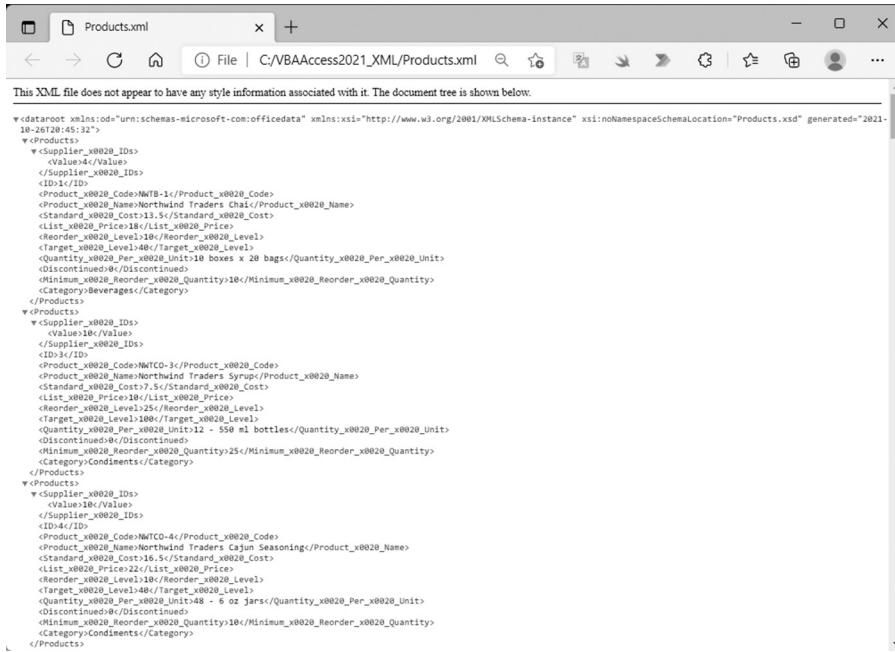
### **Hands-On 27.2 Examining the Contents of an XML Data File**

1. Open File Explorer and switch to the C:\VBAAccess2021\_XML folder. Figure 27.2 displays the contents of the Access2021\_XML folder after exporting the Products table to XML file.

Name	Date modified	Type	Size
Northwind 2007.accdb	10/26/2021 8:46 PM	Microsoft Access Database	3,968 KB
Northwind 2007.laccdb	10/26/2021 8:24 PM	Microsoft Access Record-Locking Information	1 KB
Products.htm	10/26/2021 8:45 PM	Microsoft Edge HTML Document	2 KB
Products.xml	10/26/2021 8:45 PM	XML Document	27 KB
Products.xsd	10/26/2021 8:45 PM	XML Schema File	27 KB
Products.xsl	10/26/2021 8:45 PM	XSLT Stylesheet	19 KB

FIGURE 27.2. After exporting the Shippers table to XML with all three checkboxes selected in the Export XML dialog box, Access creates four files and the Images folder.

2. Highlight the **Products.xml** and choose **Open With** from the File menu. Select **Internet Explorer** or another browser of your choice. Access displays the Products data in XML format as shown in Figure 27.3.



**FIGURE 27.3.** The tree-like structure of the XML document.

When you open an XML file in the browser, you can see the hierarchical layout of an XML document very clearly. The down arrows (or plus/minus signs in Windows 10 and earlier) make it possible to display the document as a collapsible tree. Scroll down to see the data for all the products.

The first line in the generated XML document is a dataroot element:

```
<dataroot xmlns:od="urn:schemas-microsoft-com:officedata"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:n
  oNamespaceSchemaLocation="Products.xsd" generated="2021-10-
  26T20:45:32">
```

The `dataroot` element tag defines two namespaces:

```
xmlns:od="urn:schemas-microsoft-com:officedata"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

A *namespace* is a collection of names in which each name is unique. The XML namespaces are used in XML documents to ensure that the tag used for

element names do not conflict with one another and are unique within a set of names (a namespace).

For example, the `<TITLE>` tag will certainly have a different meaning and content in an XML document generated from the Books table than the `<TITLE>` element used to describe the courtesy titles of your customers. If the two XML documents containing the `<TITLE>` tag were to be merged, there would be an element name conflict. Therefore, to distinguish between tags that have the same names but need to be processed differently, namespaces are used.

The attribute `xmlns` is an XML keyword used for declaring a namespace. The namespace is identified by a Uniform Resource Identifier (URI)—either a Uniform Resource Locator (URL) or a Uniform Resource Name (URN). The URI used as an XML namespace name is simply an identifier; it is not guaranteed to point to anything. Most namespaces use URIs for the namespace names because URIs are guaranteed to be unique. The use of a namespace is identified via a name prefix, which is mapped to a URI to select a namespace.

For example, in the context of the Products.xml document, the `od` prefix is associated with the `urn:schemas-microsoft-com:officedata` namespace and the `xsi` prefix identifies the `http://www.w3.org/2001/XMLSchema-instance namespace`. These prefixes may be associated with other namespaces outside of this XML document. Notice that the prefix is separated from the `xmlns` attribute with a colon and the URI is used as the value of the attribute.

In addition to namespaces, the dataroot element specifies where to find the schema file. This is done by using two attributes: the location of a schema file that defines the rules of an XML document and the date the file was generated.

```
xsi:noNamespaceSchemaLocation="Products.xsd"  
generated="2021-10-26T20:45:32"
```

An XML document's data is contained in elements. An element consists of the following three parts:

- Start tag—Contains the element's name such as `<ID>`
- Element data—Represents the actual data, for example: 1
- End tag—Contains the element's name preceded by a slash such as `</ID>`

If you expand the dataroot element, you will notice that the dataroot element encloses all the elements in the Access XML file. Each element in a tree structure is called a *node*.

The dataroot node contains child nodes for each row of the Products table. Notice that the table name is used for each element representing a row. You can expand or collapse any row element by clicking on the arrow or plus or minus sign (+/-) in front of the element tag name.

Within row elements, there is a separate element for each table column such as ID, Supplier IDs, Product Code, and so on. Notice that each XML element contains a start tag, the element data, and the end tag:

```
<Products>
<Supplier_x0020_IDs>
<Value>4</Value>
</Supplier_x0020_IDs>
<ID>1</ID>
<Product_x0020_Code>NWTB-1</Product_x0020_Code>
<Product_x0020_Name>Northwind Traders Chai</Product_x0020_Name>
<Standard_x0020_Cost>13.5</Standard_x0020_Cost>
<List_x0020_Price>18</List_x0020_Price>
<Reorder_x0020_Level>10</Reorder_x0020_Level>
<Target_x0020_Level>40</Target_x0020_Level>
<Quantity_x0020_Per_x0020_Unit>10 boxes x 20 bags</Quantity_x0020_Per_x0020_Unit>
<Discontinued>0</Discontinued>
<Minimum_x0020_Reorder_x0020_Quantity>10</Minimum_x0020_Reorder_x0020_Quantity>
<Category>Beverages</Category>
</Products>
```

The elements listed under Products are children of the Products element. In turn, each Products element is a child of the dataroot element. XML documents can be nested to any depth provided that each inner node is entirely contained within the outer node.

At the end of the XML document, you should see the ending dataroot element: </dataroot>.

### 3. Close the browser containing the Products.xml file.

## Understanding the XML Schema File

Now let's look at another type of XML file that was created by Access during the export to XML process—the XML schema file (XSD).

Schema files describe XML data using the XML Schema Definition (XSD) language and allow the XML parser to validate the XML document. An XML document that conforms to the structure of the schema is said to be *valid*.

Here are some examples of the types of information that can be found in an XML schema file:

- Elements that are allowed in each XML document
- Data types of allowed elements
- Number of allowed occurrences of a given element

- Attributes that can be associated with a given element
  - Default values for attributes
  - Child elements of other elements
  - The sequence and number of child elements

## Hands-On 27.3 Examining the Contents of an XML Schema File

Open File Explorer and switch to the C:\VBAAccess2021\_XML folder containing the files generated in Hands-On 27.1 and presented in Figure 27.2.

1. Use Notepad to open the **Products.xsd** file located in the Access2021\_XML folder. Access displays the contents of the Products.xsd file as shown in Figure 27.4.

**FIGURE 27.4.** The partial view of the schema file shown here defines the data in the Products.xml document.

If you examine the Products.xsd file currently open in Notepad, you will notice several xsd declarations and commands that begin with the `<xsd>` tag followed by a colon and the name of the command. You will also notice the names of the elements and attributes that are allowed in the Products.xml file as well as the data types for each element.

The names of the data types begin with the `od` prefix followed by a colon. The schema file also specifies the number of times an element can be used in a document based on the schema. This is done via the `minOccurs` and `maxOccurs` attributes.

2. Close Notepad and the Products.xsd file.

**NOTE**

*To learn more about XML schemas, check out the following links:*  
<http://www.w3.org/TR/xmlschema-0/>  
<http://www.w3.org/TR/xmlschema-1/>  
<http://www.w3.org/TR/xmlschema-2/>

### **Understanding the XSL Transformation Files**

---

When you examined the contents of the Products.xml document earlier in this chapter you may have noticed that the file did not contain any formatting instructions. Although it is easy to display the XML file in any browser, end users expect to see documents that are nicely formatted. To meet their expectations, the raw XML data is formatted with the Extensible Stylesheet Language (XSL).

When you exported the Products table to XML and selected the Presentation of your data (XSL) checkbox in the Export XML dialog box (see Hands-On 27.1), Access generated an XSL file. Extensible Stylesheet Language is a transformation style language that uses XSL Transformations (XSLT) to create templates that are applied to the source document data to create the target document. The target document can be another XML document, an HTML page, or even a text-based file.

XSL files include all the XSLT transforms that are needed to define how the data is to be presented. Transformations allow you to change the order of elements and selectively process elements. Later in this chapter you create XSL files with XSLT transforms to display only selected fields from the Access-generated XML documents. There is no limit to the number of stylesheets that can be used with a particular XML document. By creating more than one XSL file, you can present different styles of the same XML document to various users.



#### Hands-On 27.4 Examining the Contents of an XSL File

1. Use Notepad to open the **Products.xsl** file located in the **C:\VBAAccess2021\_XML** folder. Access displays the contents of the Products.xsl file as shown in Figure 27.5.

The screenshot shows a Windows Notepad window titled "Products.xsl - Notepad". The window contains the XML code for an XSLT stylesheet. The code includes declarations for XML version 1.0, namespaces for XML Transform and MSXSL, and an output method set to HTML version 4.0 with a yes indent. It defines a template for the dataroot that outputs an HTML page with a title "Products" and a style sheet. The body contains a table with several columns, each with a specific width defined in inches. The table is styled with a border of 1, a white background (#FFFFFF), and no cell spacing or padding. The entire table is enclosed in a colgroup element. The XSLT code uses various elements like `<xsl:template>`, `<xsl:for-each>`, and `<xsl:value-of>` to manipulate the XML data. The code also includes some HTML-like elements such as `<head>`, `<title>`, `<style>`, `<body>`, `<table>`, `<tr>`, `<td>`, and `<div>`.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:msxsl="urn:schemas-microsoft-com:xslt" xmlns:fx="#fx-functions" exclude-result-prefixes="msxsl fx">
    <xsl:output method="html" version="4.0" indent="yes" encoding="UTF-8" />
    <xsl:template match="/" data-root="" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <html>
            <head>
                <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
                <title>Products</title>
                <style type="text/css"></style>
            </head>
            <body link="#0c0000" vlink="#050000">
                <table border="1" bgcolor="#FFFFFF" cellspacing="0" cellpadding="0" id="CTRL1">
                    <colgroup>
                        <col style="width: 1.6875in;" />
                        <col style="text-align: right; width: 0.4062in;" />
                        <col style="width: 1.1354in;" />
                        <col style="width: 2.802in;" />
                        <col style="width: 1.0184in;" />
                        <col style="text-align: right; width: 1.1562in;" />
                        <col style="text-align: right; width: 0.8125in;" />
                        <col style="text-align: right; width: 0.927in;" />
                        <col style="text-align: right; width: 1.1458in;" />
                        <col style="width: 1.6145in;" />
                        <col style="text-align: right; width: 0.9375in;" />
                        <col style="text-align: right; width: 2.0625in;" />
                        <col style="width: 1.8437in;" />
                        <col style="width: 0.9375in;" />
                    </colgroup>
                    <tbody>
                        <tr>
                            <td>
                                <div align="center">
                                    <strong>Supplier IDs</strong>
                                </div>
                            </td>
                            <td>
                                <div align="center">
                                    <strong>ID</strong>
                                </div>
                            </td>
                        
                    
                
            </body>
        
    

```

FIGURE 27.5. The XSL stylesheet document is just another XML document that contains HTML formatting instructions and XSLT formatting elements for transforming raw XML data into HTML.

When you expand all the nodes and scroll through the contents of the Products.xsl file you will notice a number of XSLT formatting elements such as `<xsl:template>`, `<xsl:for-each>`, and `<xsl:value-of>`. You will also find many HTML formatting instructions such as `<head>`, `<title>`, `<style>`, `<body>`, `<tbody>`, `<table>`, `<colgroup>`, `<col>`, `<tr>`, `<td>`, `<div>`, and `<strong>`.

The first line of the stylesheet code declares that this is an XML document that follows the XML 1.0 standard (version). An XSL document is a type of

XML document. While XML documents store data, XSL documents specify how the data should be displayed.

The second line declares the namespace that will be used to identify the tags in the XSL document. (See the “Understanding the XML Data File” section earlier in this chapter for more information about namespaces.) The third line specifies that HTML should be used to display the data.

The next line is the beginning of the formatting section. Before we look at the XSLT tags, you need to know that XSL documents use templates to perform transformations of XML documents. The XSL stylesheet can contain one or more XSLT templates. You can think of templates as special blocks of code that apply to one or more XML tags. Templates contain rules for displaying a set of elements in the XML document. The use of templates is made possible via special formatting tags.

Notice that the Products.xsl file contains the `<xsl:template>` tag to define a template for the entire document. The `<xsl:template>` element has a `match` attribute. The value of this attribute indicates the nodes (elements) for which this template is appropriate.

For example, the special pattern “`//`” in the match attribute tells the XSL processor that this is the template for the document root:

```
<xsl:template match="//dataroot"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

The template ends with the `</xsl:template>` closing tag. Following the definition of the template, standard HTML tags are used to format the document. Next, the XSLT formatting instruction `<xsl:for-each>` (Figure 27.6) tells the XSL processor to do something every time it finds a pattern. The pattern follows the `select` attribute.

For example:

```
<xsl:for-each select="Products">
```

tells the XML processor to loop through the `<Products>` elements. The loop is closed with a closing loop tag:

```
</xsl:for-each>
```

The XSLT formatting instruction `<xsl:value-of>` tells the XSL processor to retrieve the value of the tag specified in the `select` attribute. For example:

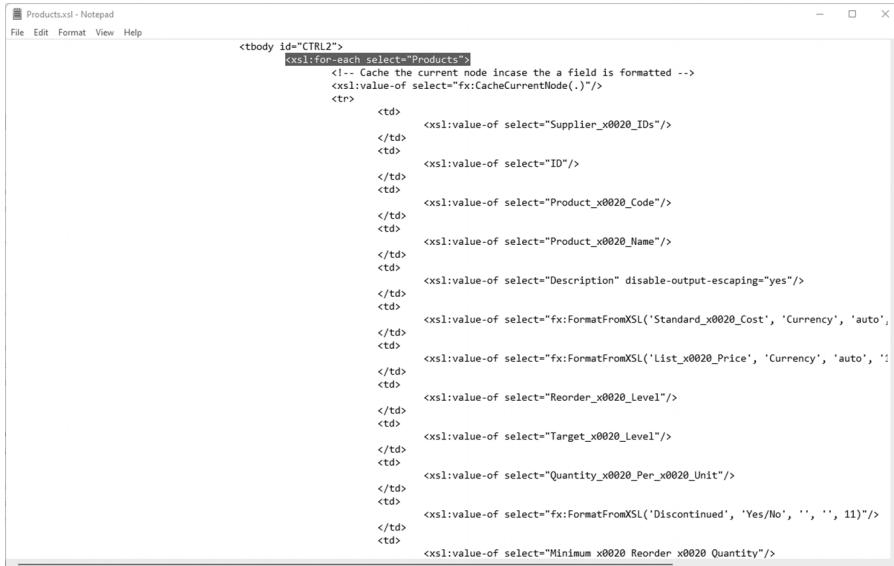
```
<xsl:value-of select="ID">
```

tells the XML processor to select the ID column. Because this formatting

instruction is located below the `<xsl:for-each>` tag, the XSL processor will retrieve the value of the ID column for each Products element. The select attribute uses the XML Path language (XPath) expression to locate the child elements to be processed.

If you scroll down the Products.xsl file, you will also notice that Access has generated several VBScript functions to evaluate expressions. To prevent the XSL processor from parsing these functions, the function section is placed within the CDATA directive (`<![CDATA[...]`)

## 2. Close the browser containing the Products.xsl file.



The screenshot shows a Windows Notepad window titled "Products.xsl - Notepad". The menu bar includes File, Edit, Format, View, and Help. The main content area displays an XSLT transformation script. It starts with an `<tbody id="CTRL2">` tag, followed by an `<xsl:for-each select="Products">` tag. Inside this loop, there are multiple `<td>` tags, each containing an `<xsl:value-of>` instruction with a XPath expression. The expressions include `Supplier_x0020_IDs`, `ID`, `Product_x0020_Code`, `Product_x0020_Name`, `Description` (with `disable-output-escaping="yes"`), `Standard_x0020_Cost`, `List_x0020_Price`, `Reorder_x0020_Level`, `Target_x0020_Level`, `Quantity_x0020_Per_x0020_Unit`, `Discontinued` (with `'Yes/No', '' , '' , 11'`), and `Minimum x0020 Reorder x0020 Quantity`.

FIGURE 27.6. The XSLT formatting instructions in the Products.xsl file.

### SIDE BAR *What Exactly Is XPath?*

XPath is a query language used to create expressions for finding data in the XML data file. These expressions can manipulate strings, numbers, and Boolean values. They can also be used to navigate an XML tree structure and process its elements with XSLT instructions. XPath is designed to be used by XSL Transformations (XSLT). With XPath expressions, you can easily identify and extract from the XML document specific elements (nodes) based on their type, name, values, or the relationship of a node to other nodes. When preparing stylesheets for transforming your XML documents into HTML, you will often use various XPath expressions in the select attribute.

**NOTE**

*For more information about Extensible Stylesheet Language (XSL), visit: <http://www.w3.org/TR/xsl/>*

## **Viewing XML Documents Formatted with Stylesheets**

When you exported the Products table to XML format, Access applied XSLT transforms to turn the XML data into an HTML file called Products.htm. Figure 27.7 displays the contents of this file. Let's spend few minutes reviewing the code statements in this file. Notice that when the HTML page loads, it executes the VBScript `ApplyTransform` function, that appears in the `<SCRIPT>` block just below the ending `</BODY>` tag.

The VBScript, which is a scripting language introduced by Microsoft in Internet Explorer in 1996, uses a software component called the XML Document Object Model (DOM) that provides methods and properties for working with XML programmatically, allowing you to output and transform the XML data.

The `DOMDocument` object is the top level of the XML DOM hierarchy and represents a tree structure composed of nodes. You can navigate through this tree structure and manipulate the data contained in the nodes by using various methods and properties. Because every XML object is created and accessed from `DOMDocument`, you must first create the `DOMDocument` object to work with an XML document.

The `ApplyTransform` function begins by setting an object variable (`objData`) to an instance of `DOMDocument` that's returned by a custom `CreateDOM` function:

```
Set objData = CreateDOM
```

The `CreateDOM` function creates a reference to the `DOMDocument` via the `CreateObject` method of the `Server` object. Because different versions of the MSXML parser may be installed on a client machine (`DOMDocument6`, `DOMDocument5`, `DOMDocument4`, etc.), the function attempts to instantiate the `DOMDocument` object using the most recent version. If such a version is not found, it looks for older versions of the MSXML parser that may exist. It is extremely important that only one version of the `DOMDocument` is used, since mixing `DOMDocument` objects from different versions of the MSXML parser can cause ugly errors.

Once the `DOMDocument` object has been instantiated, the `LoadDOM` function listed at the bottom of the page is called. This function expects two parameters: `objDOM`, which is the `objData` variable referencing the `DOMDocument`, and `strXMLFile`, which is the name of the file to load into the `DOMDocument` object. To

ensure that Internet Explorer waits until all the data is loaded before rendering the rest of the page, the `Async` property of the `DOMDocument` is set to False:

```
objDOM.Async = False  
objDOM.Load strXMLFile
```

The `Load` method is used to load the supplied file into the `objData` object variable. This method returns True if it successfully loaded the data and False otherwise. If there is a problem with loading, a description of the error is returned in a message box.

The `Document` object of XML DOM exposes a `parseError` object that allows you to check whether there was an error when loading the XML file or stylesheet. The `ParseError` object has the properties, one of them is the `reason` property that provides text description of the error.

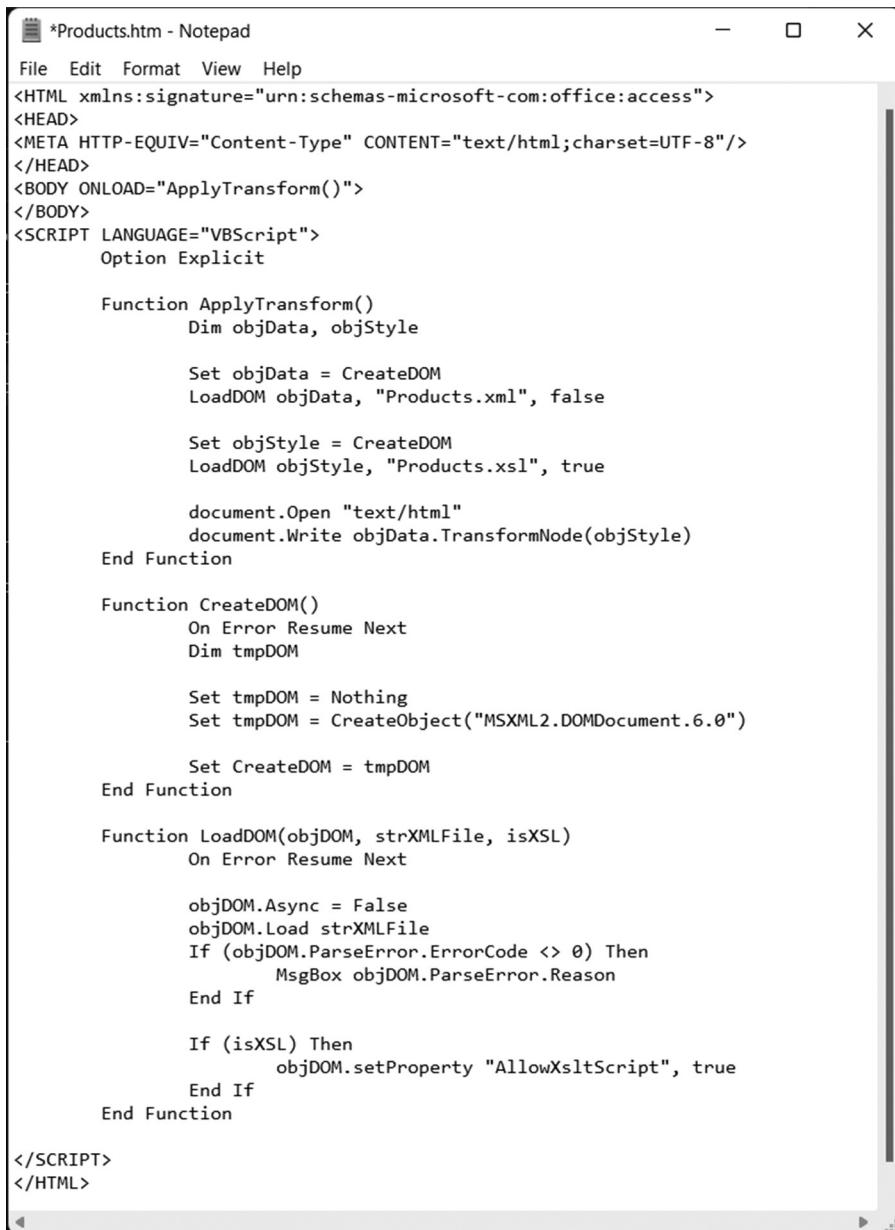
After loading the `Products.xml` data file into the DOM software component, the `ApplyTransform` function repeats the same process for the `Products.xsl` file. After both files are successfully loaded, the transform is applied to the data using the `TransformNode` method:

```
document.Write objData.TransformNode (objStyle)
```

The `TransformNode` method performs the transformation by applying the XSL stylesheet to the XML data file. The result should be a nicely formatted products table displayed in a browser. But wait, this is no longer true in 2021!

In August 2019, Microsoft officially disabled VBScript in Internet Explorer. This means that the VBScript code that Access produced will never run by default. The code is not running on my Windows 11 machine where Internet Explorer is not even present. It is not running on my Windows 10 machine after the most recent updates. Do I care? Not at all. It's 2021 and it's time to move on. When you try to open the `Products.htm` file in your browser all you get is a blank page. Save yourself time, and do not try in other browsers. VBScript never worked in non-Microsoft browsers such as Google Chrome, Mozilla Firefox, and Apple Safari. Those browsers adopted JavaScript for client-side scripts.

All you've learned in this section about the DOM Document object will come in handy later in this chapter when you yourself write code to format the XML data.



The screenshot shows a Windows Notepad window titled "Products.htm - Notepad". The window contains VBScript code for generating an HTML page from XML and XSL files. The code includes functions for applying transforms, creating DOM objects, and loading XML files.

```
<HTML xmlns:signature="urn:schemas-microsoft-com:office:access">
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=UTF-8"/>
</HEAD>
<BODY ONLOAD="ApplyTransform()">
</BODY>
<SCRIPT LANGUAGE="VBScript">
    Option Explicit

    Function ApplyTransform()
        Dim objData, objStyle

        Set objData = CreateDOM
        LoadDOM objData, "Products.xml", false

        Set objStyle = CreateDOM
        LoadDOM objStyle, "Products.xsl", true

        document.Open "text/html"
        document.Write objData.TransformNode(objStyle)
    End Function

    Function CreateDOM()
        On Error Resume Next
        Dim tmpDOM

        Set tmpDOM = Nothing
        Set tmpDOM = CreateObject("MSXML2.DOMDocument.6.0")

        Set CreateDOM = tmpDOM
    End Function

    Function LoadDOM(objDOM, strXMLFile, isXSL)
        On Error Resume Next

        objDOM.Async = False
        objDOM.Load strXMLFile
        If (objDOM.ParseError.ErrorCode <> 0) Then
            MsgBox objDOM.ParseError.Reason
        End If

        If (isXSL) Then
            objDOM.setProperty "AllowXsltScript", true
        End If
    End Function

</SCRIPT>
</HTML>
```

FIGURE 27.7. Access generated Products.htm file opened in Windows Notepad.

Now the good part. To see your Products table data nicely displayed in any browser, use the Export to HTML Document feature in Access. To do this,

- open the Products table,
- click the External Data tab and choose HTML Document from the More drop-down.
- Enter the C:\VBAAccess2021\_XML\Products.html in the File name box and check the box next to Export data with formatting and layout.
- In the HTML Output Options dialog box click OK while the Default encoding is selected.
- Click Close to exit the Save Export Step dialog.
- Open File Explorer and double-click the Products.html file in the C:/VBAAccess2021\_XML folder.

The result of the Export to HTML Document is shown in Figure 27.8.

Products													
Supplier ID	ID	Product Code	Product Name	Description	Standard Cost	List Price	Reorder Level	Target Level	Quantity Per Unit	Discontinued	Minimum Reorder Quantity	Category	Attachments
Supplier D	1	NW-TRD-1	Northwind Traders Chai		\$13.50	\$18.00	10	40	10 boxes x 20 bags	No	10	Beverages	0
Supplier J	2	NW-TRD-2	Northwind Traders Syrup		\$7.50	\$10.00	25	100	12 - 550 ml bottles	No	25	Condiments	0
Supplier J	3	NW-TRD-3	Northwind Traders Cajun Seasoning		\$16.50	\$22.00	10	40	48 - 6 oz jars	No	10	Condiments	0
Supplier J	4	NW-TRD-4	Northwind Traders Curry Seasoning		\$16.50	\$22.00	10	40	48 - 6 oz jars	No	10	Condiments	0
Supplier J	5	NW-TRD-5	Northwind Traders Dried Apricots		\$16.01	\$21.35	10	40	36 boxes	No	10	Condiments	0
Supplier J, Supplier F	6	NW-TRD-6	Northwind Traders Dried Cranberries		\$23.50	\$30.00	20	100	12 - 16 oz jars	No	20	Jams, Preserves	0
Supplier B	7	NW-TRD-7	Northwind Traders Dried Pears		\$23.50	\$30.00	10	40	12 - 1 lb pkgs.	No	10	Dried Fruit & Nuts	0
Supplier H	8	NW-TRD-8	Northwind Traders Curry Sauce		\$30.00	\$40.00	10	40	12 - 12 oz jars	No	10	Sauces	0
Supplier L, Supplier F	9	NW-TRD-9	Northwind Traders Walnuts		\$17.44	\$23.25	10	40	40 - 100 g pkgs.	No	10	Dried Fruit & Nuts	0
Supplier F	10	NW-TRD-10	Northwind Traders Fruit Cocktail		\$29.25	\$39.00	10	40	15 25 OZ	No	10	Canned Fruit & Vegetables	0
Supplier A	11	NW-TRD-11	Northwind Traders Chocolate Biscuits Mix		\$6.90	\$9.20	5	20	10 boxes x 12 pieces	No	5	Baked Goods & Mixes	0
Supplier G, Supplier F	12	NW-TRD-12	Northwind Traders Caramel Biscuits Mix		\$60.75	\$81.00	10	40	20 gift boxes	No	10	Jams, Preserves	0
Supplier G, Supplier F	13	NW-TRD-13	Northwind Traders Caramel Fudge		\$20.00	\$25.00	20	100	12 - 4 oz pieces	No	20	Baked Goods & Mixes	0
Supplier D	14	NW-TRD-14	Northwind Traders Beer		\$30.50	\$44.00	15	60	24 - 12 oz bottles	No	15	Beer	0
Supplier G	15	NW-TRD-15	Northwind Traders Crab Meat		\$13.80	\$16.40	30	120	24 - 4 oz tins	No	30	Canned Meat	0
Supplier F	16	NW-TRD-16	Northwind Traders Clam Chowder		\$7.24	\$9.65	10	40	12 - 12 oz cans	No	10	Soups	0
Supplier C, Supplier D	17	NW-TRD-17	Northwind Traders Coffee		\$34.50	\$46.00	25	100	16 - 500 g tins	No	25	Beverages	0
Supplier J	18	NW-TRD-18	Northwind Traders Chocolate		\$9.56	\$12.75	25	100	10 pkgs	No	25	Candy	0
Supplier B	19	NW-TRD-19	Northwind Traders Dried Apples		\$39.75	\$53.00	10	40	50 - 300 g pkgs.	No	10	Dried Fruit & Nuts	0
Supplier A	20	NW-TRD-20	Northwind Traders Long Grain Rice		\$5.25	\$7.00	25	100	16 - 2 kg boxes	No	25	Grains	0
Supplier A	21	NW-TRD-21	Northwind Traders Macaroni		\$6.50	\$8.00	30	120	16 - 100 g pkgs.	No	30	Pasta	0
Supplier A	22	NW-TRD-22	Northwind Traders Noodles		\$14.45	\$16.50	25	100	24 - 250 g pkgs.	No	25	Pasta	0
Supplier H	23	NW-TRD-23	Northwind Traders Olives		\$15.75	\$20.00	10	40	12 - 8 oz bottles	No	10	Sauces	0
Supplier H	24	NW-TRD-24	Northwind Traders Hot Pepper Sauce		\$15.79	\$21.05	10	40	12 - 8 oz jars	No	10	Sauces	0
Supplier H	25	NW-TRD-25	Northwind Traders Tomato Sauce		\$12.75	\$17.00	20	80	24 - 8 oz jars	No	20	Sauces	0
Supplier E	26	NW-TRD-26	Northwind Traders Mozzarella		\$26.10	\$34.80	10	40	24 - 200 g pkgs.	No	10	Dairy Products	0
Supplier J	27	NW-TRD-27	Northwind Traders										

FIGURE 27.8 Access generated Products.html file opened in Edge browser.

## Advanced XML Export Options

When you exported the Products table to XML format, you may have noticed the More Options button in the Export XML dialog box (see Figure 27.1 at the beginning of this Chapter). Pressing this button opens a window with three tabs as shown in Figure 27.9. Each tab groups options for the types of XML objects that you can export. The Data tab contains options for the XML document, the Schema tab lists options for the XSD document, and the Presentation tab provides options for generating the XSL document.

### Data Export Options

The options shown on the Data tab (see Figure 27.9) control the data that is exported to the XML documents. These options are grouped into three main areas.

The Data to Export section displays data that you may want to export. In this particular scenario the Customers table has been chosen for export. Because this table is directly related to the Orders table in the Northwind 2007 database, the Orders table is displayed as a child node of Customers. The Orders table is related to the Order Details table and so on. Clicking on the plus sign in front of the [Lookup Data] node will display the names of tables that provide lookup information for the main tables. By clicking on the checkbox you may export just the table that you originally requested or you can export the customers' data along with all the orders, and perhaps include lookup information.

Below the Data to Export section is the Export Location area that shows the filename for the XML document that will be created when you click the OK button. You can change the location of this document by using the Browse button. Simply navigate to the folder where you want to save the XML file. You can also change the name of the document by replacing the name shown in the text box with another name.

The area to the right of the Data to Export section allows you to specify which records you want to export. This area contains three option buttons that allow you to export all records, filtered records, or the current record. Notice that only one option is enabled in Figure 27.9.

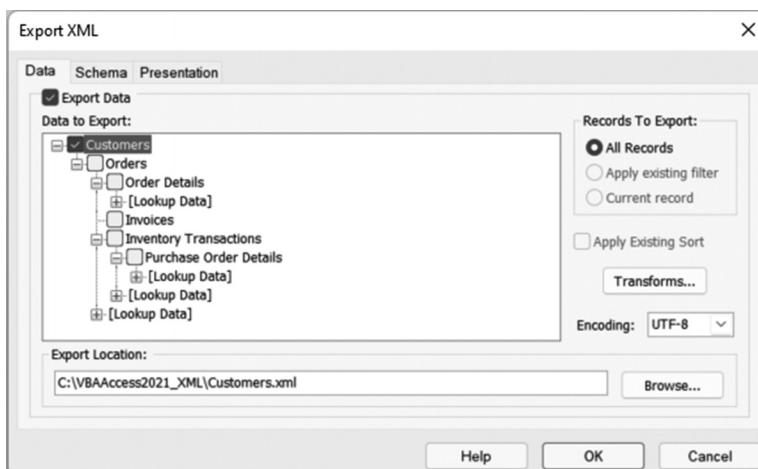


FIGURE 27.9. Use the Data tab in the Export XML window to set advanced data options.

When you highlight the table to export in the database window and then choose the Export command from the File menu, only the All Records option button will be enabled in the Records To Export section. Opening the table prior to choosing the Export command tells Access to enable the All Records and Current record option buttons. And if you open the table and apply a filter to the data, then select the Export command, Access will enable the Apply existing filter option button in addition to the other two buttons.

The other options on the Data tab are Apply Existing Sort, Transforms, and Encoding. The Apply Existing Sort checkbox is enabled if the exported object is open and a sort is applied. Access will export the data in the specified sort order. Clicking the Transforms button allows you to select a custom XSL transform file to apply to the data during export. You can choose from the transforms you have written or received with the XML data. Use the Encoding drop-down list to select UTF-8 or UTF-16 encoding for the exported XML. The default is UTF-8.

When you export an object from an Access database file, Access exports static data. This means that the exported object is not automatically updated when the data changes. If the data in the Access database has changed since you exported an Access object to an XML data file, you will need to re-export the object so the new data is available to the client application.

<b>NOTE</b>	<i>Exporting live data is supported by Access data projects (.adp file format) in Access 2010. Support for ADP was removed in Access 2013, therefore additional options related to Access data projects are not discussed here.</i>
-------------	---

### ***Schema Export Options***

The options shown on the Schema tab (see Figure 27.10) control the way the schema file for the object is exported. Advanced schema options are presented in two sections: Export Schema and Export Location.

The Export Schema section has two checkboxes. By selecting the Export Schema checkbox you indicate that you want to export the object's schema as an XSD file. This selection is the same as choosing the Schema of the data (XSD) option in the first Export XML dialog box (see Figure 27.1). The checkboxes under Export Schema allow you to specify whether you want to include primary key and index information in the XSD schema file, and whether to export all table and field properties.

The Export Location section has two option buttons that allow you to specify whether you want the schema information to be embedded in the exported XML data document or stored in a separate schema file. You can enter the

filename in the provided text box and specify the location of the schema file by clicking the Browse button.

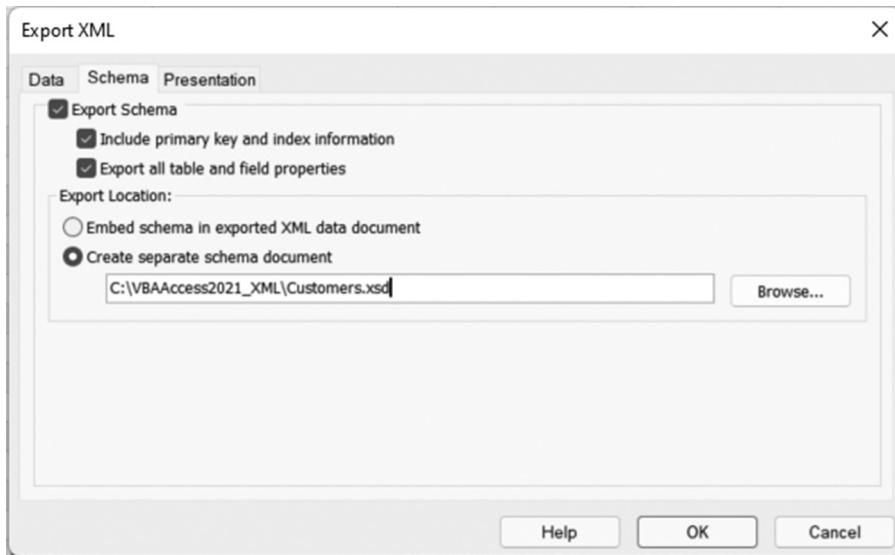


FIGURE 27.10. Use the Schema tab in the Export XML dialog box to set advanced schema options.

### ***Presentation Export Options***

The selections on the Presentation tab (see Figure 27.11) specify available options for the XSL files. The Export Presentation (HTML 4.0 Sample XSL) checkbox allows you to indicate whether you want to export the object's presentation. Choose the Client (HTML) option in the Run from section if you want the presentation to run on the client. Access will create an HTML file with the script necessary to perform the transform. The script will be executed on the client (user) machine. While this selection reduces the load on the server, a client application will need to download a few files (HTML document, XML data file, and XSD schema file) to present the data in the browser. If the XSL file is going to be placed on the Web server and called from a classic ASP page, choose the Server (ASP) option. By choosing this option, only the final HTML is downloaded to the client.

If the exported presentation includes pictures, you can indicate whether to include them in the output by clicking the appropriate option button in the Include report images section. If you choose to include the images, Access will create separate image files and link them with the HTML file. By default, the

image files are stored in the Images folder of the main export folder. To place them in another location, click the Browse button to specify the folder name.

The Export Location section allows you to specify the name and location of the export files. When you export a presentation file, Access creates two files: an XSL file that includes all the XSLT transforms needed to define how the data is presented, and a simple HTML file that contains properly formatted data from the exported object and not the raw data with XML tags. The HTML file contains a snapshot of the data as it existed during the export process.

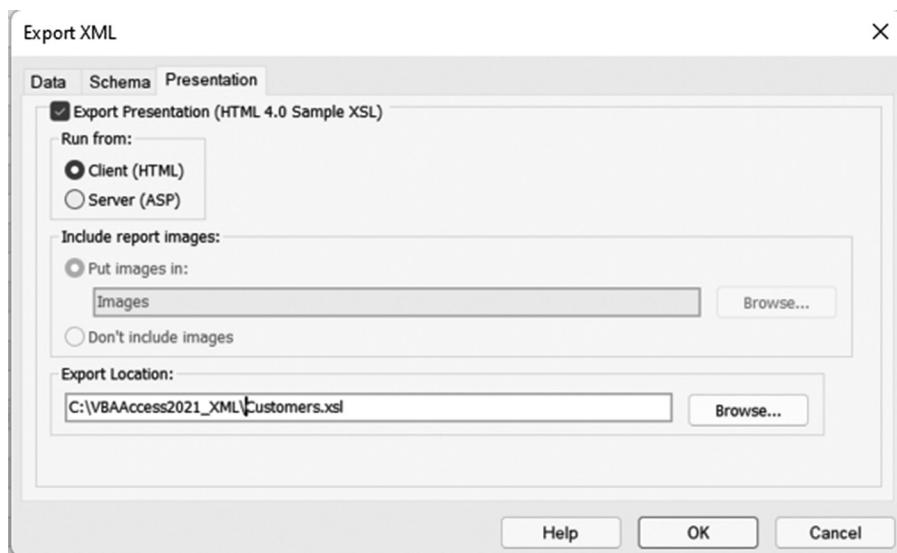


FIGURE 27.11. Use the Presentation tab in the Export XML dialog box to set advanced presentation options.



### Hands-On 27.5 Advanced Export of the Customer table to XML and ASP

1. Open the Customers table in the Northwind 2007.accdb database.
2. Choose External Data | More | XML File.
3. Enter C:\VBAAccess2021\_XML\Customers\_Server.xml in the File name box and click OK.
4. In the Export XML dialog box make sure the first 2 checkboxes are selected and click the More Options button.
5. In the Data tab of the Export XML dialog box ensure that Customers table is selected for Export of All Records and the export location is pointed to C:\VBAAccess2021\_XML\Customers\_Server.xml. Do not click OK.

6. Click the Schema tab in the same Export to XML dialog box. In the Schema tab, make sure that a separate schema document option button is selected, and both check boxes are checked under the Export Schema section. Enter the full path and name of the schema file as C:\VBAAccess2021\_XML\Customers\_Server.xsd. Do not click OK.
7. Click the Presentation tab in the same Export to XML dialog box. Select Export Presentation (HTML 4.0 Sample XSL and run it from Server (ASP). Enter C:\VBAAccess2021\_XML\Customers\_Server.xsl. Click OK to finish setting up the Export Options.
8. Click Close to exit the Save Export Steps screen.
9. Open the File Explorer and ensure that Access has generated four new files in the C:\VBAAccess2021\_XML folder. You should see the following files:

Customers_Server.asp	Active Server Page
Customers_Server.xml	XML Document
Customers_Server.xsd	XML Schema File
Customers_Server.xsl	XSLT Stylesheet

Now that the Customers data has been generated, how can you view this data? The Active Server page (ASP) is a text file with an .ASP extension. Because this file is located on your computer, you can open it with Windows Notepad. However, once this file is placed on a Web server, only authorized people will be able to access it.

10. Use Windows Notepad to open the Customers\_Server.asp file to view its source code.

The contents of the file are as follows:

```
<%
Set objData = CreateDOM
objData.async = false

if (false) then
    Set objDataXMLHTTP = Server.CreateObject("Microsoft.XMLHTTP")
    objDataXMLHTTP.open "GET", "", false
    objDataXMLHTTP.setRequestHeader "Content-Type", "text/xml"
    objDataXMLHTTP.send
    objData.load(objDataXMLHTTP.responseText)
else
    objData.load(Server.MapPath("Customers_Server.xml"))
end if

Set objStyle = CreateDOM
```

```
objStyle.async = false
objStyle.load(Server.MapPath("Customers_Server.xsl"))
Session.CodePage = 65001

Response.ContentType = "text/html"
Response.Write objData.transformNode(objStyle)

Function CreateDOM()
    On Error Resume Next
    Dim tmpDOM
    Set tmpDOM = Nothing
    Set tmpDOM = Server.CreateObject("MSXML2.DOMDocument.6.0")

    Set CreateDOM = tmpDOM
End Function
%>
```

The Active Server (ASP) files allow you to create dynamic web pages where information is updated dynamically from the connected data source. In this case, the data source is the Access generated XML file. ASP files can include standard HTML formatting tags, embedded scripting statements, and references to other files. The VBScript, which is a subset of VBA, is the scripting language for ASP. As you review the file, you will notice that it begins with the <% and ends with the %> delimiters. The statements that appear between these delimiters constitute the VBScript code that will be executed on the Web server once the connection with it is established.

The VBScript code uses a software component called the XML Document Object Model (DOM). The DOM offers methods and properties for working with XML programmatically, allowing you to output and transform the XML data.

The `DOMDocument` object is the top level of the XML DOM hierarchy and represents a tree structure composed of nodes. You can navigate through this tree structure and manipulate the data contained in the nodes by using various methods and properties. Because every XML object is created and accessed from `DOMDocument`, you must first create the `DOMDocument` object to work with an XML document. Therefore, the first statement in the `Customers_Server.asp` file sets an object variable (`objData`) to an instance of `DOMDocument` that's returned by a custom `CreateDOM` function:

```
Set objData = CreateDOM
```

The `CreateDOM` function sets a reference to the `DOMDocument` via the `CreateObject` method of the `Server` object.

Once the `DOMDocument` object has been instantiated, the second statement in the file ensures that the browser waits until all the data is loaded before rendering the rest of the page. This is done by setting the `Async` property of the `DOMDocument` to false:

```
objData.async = false
```

The next step is to make sure that we can send and receive information to and from a Web server. This is done by creating an instance of `Microsoft.XMLHTTP` object. We store the reference to this object in the object variable `objDataXMLHTTP`. The `open` method is then used to open the connection to a Web server. The `open` method takes several arguments. The first argument specifies the type of HTTP Protocol being called. In this case, we use the simple “GET” protocol. The second argument contains the `URL` that is being targeted. In this case, we use the empty string (“”) to indicate that we are calling the current ASP page. The third argument, which is set to `false`, tells the browser to send the request and wait for the return reply. The next statement sends some extra information to the server to inform it about the type of data being sent to it. This done via the `setRequestHeader` property of the `XMLHTTP` object. The `Content-Type` header informs the server that the document being sent is the XML document. Notice that the `Content-Type` is set to “`text/xml`.”

The code then uses the `send` method to send the request via the HTTP GET protocol specified in the `open` method. `Send` opens a connection to the Web server and sends the header information that was specified earlier. At this point, the Web server will send back a set of headers with the status information so the browser can display error messages, if necessary. If the connection succeeded, the statement in the Else clause will load the specified XML file for processing by the Web server. The `Load` method is used to load the supplied file into the `objData` object variable. This method returns `True` if it successfully loaded the data and `False` otherwise. After loading the `Customer_Server.xml` data file into the DOM, the transform is applied to the data using the `transformNode` method:

```
document.Write objData.transformNode(objStyle)
```

The `TransformNode` method performs the transformation by applying the XSL stylesheet to the XML data file. After both XML and XSL files have been processed, the following statement is executed:

```
objData.load(objDataXMLHTTP.responseBody)
```

The `responseBody` property of the `XMLHTTP` object returns the data from the server as an HTML document, as depicted in Figure 27.12.

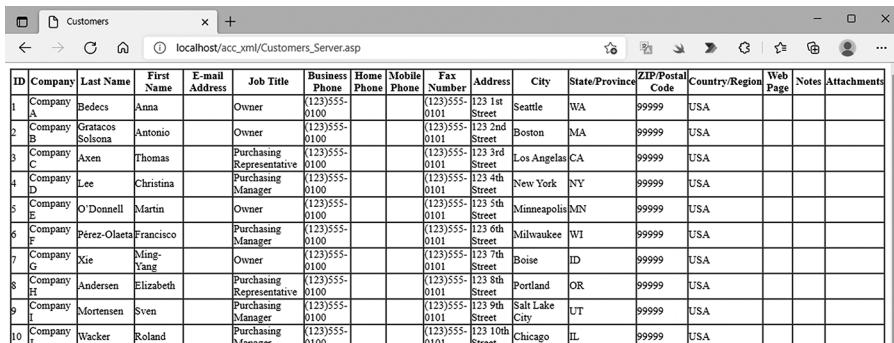
**11. Close the Customers\_Server.asp file.**

Now that you know how the VBScript code looks like, let's prepare your environment for running this code. In the next step you are referred to Appendix A where you will find step by step instructions of how to set up the Internet Information Services (IIS) on your computer. IIS is a built-in Web server provided by the Microsoft Windows Operating System. Because this server is turned off by default, you will need to enable and configure it before it can be accessed.

**12. Go to Appendix A and follow the steps of setting up and configuring the IIS. Once completed return here and continue with Step 13.**

**13. Open your favorite browser and, in the address bar, type the following URL and press Enter. `http://localhost/acc_xml/Customers_Server.asp`**

`localhost` is the name of the Web server that you enabled on your computer. The `acc_xml` is the name of the virtual folder where the ASP script and other supporting files are located. Your browser sends the request to the Web server to process the `Customers_Server.asp` file and once the request is processed, your browser receives the response, and the resulting data is shown in Figure 27.12. If you got an error instead, read on to find out how to correct it.



A screenshot of a web browser window titled "Customers". The address bar shows the URL "localhost/acc\_xml/Customers\_Server.asp". The main content area displays a table with 10 rows of customer data. The columns are labeled: ID, Company, Last Name, First Name, E-mail Address, Job Title, Business Phone, Home Phone, Mobile Phone, Fax Number, Address, City, State/Province, ZIP/Postal Code, Country/Region, Web Page, Notes, and Attachments. The data is as follows:

ID	Company	Last Name	First Name	E-mail Address	Job Title	Business Phone	Home Phone	Mobile Phone	Fax Number	Address	City	State/Province	ZIP/Postal Code	Country/Region	Web Page	Notes	Attachments
1	Company A	Bedecs	Anna		Owner	(123)555-0100			(123)555-0101	123 1st Street	Seattle	WA	99999	USA			
2	Company B	Gratacos	Antonio		Owner	(123)555-0100			(123)555-0101	123 2nd Street	Boston	MA	99999	USA			
3	Company C	Axen	Thomas		Purchasing Representative	(123)555-0100			(123)555-0101	123 3rd Street	Los Angeles	CA	99999	USA			
4	Company D	Lee	Christina		Purchasing Manager	(123)555-0100			(123)555-0101	123 4th Street	New York	NY	99999	USA			
5	Company E	O'Donnell	Martin		Owner	(123)555-0100			(123)555-0101	123 5th Street	Minneapolis	MN	99999	USA			
6	Company F	Pérez-Olaeta	Francisco		Purchasing Manager	(123)555-0100			(123)555-0101	123 6th Street	Milwaukee	WI	99999	USA			
7	Company G	Xie	Ming-Yang		Owner	(123)555-0100			(123)555-0101	123 7th Street	Boise	ID	99999	USA			
8	Company H	Andersen	Elizabeth		Purchasing Representative	(123)555-0100			(123)555-0101	123 8th Street	Portland	OR	99999	USA			
9	Company I	Mortensen	Sven		Purchasing Manager	(123)555-0100			(123)555-0101	123 9th Street	Salt Lake City	UT	99999	USA			
10	Company J	Wacker	Roland		Purchasing Manager	(123)555-0100			(123)555-0101	123 10th Street	Chicago	IL	99999	USA			

**FIGURE 27.12.** Customers data is generated from Access by running an Active Server Page that queries the data contained in XML file and formats it with the stylesheet.

<b>NOTE</b>	<p>Errors happen everywhere, there are no perfect programs, thus programmers need to acquire skills to troubleshoot various types of errors, whether they are their own, or created by others. Here, is a file created entirely by Access, without us messing up with its code, that may produce the following error:</p> <p><i>msxml6.dll error '80004005'</i></p> <p><i>Security settings do not allow the execution of script code within this stylesheet.</i></p> <p style="padding-left: 40px;"><i>/acc_xml/Customers_Server.asp, line 23</i></p> <p><i>What seems to be a problem? How do you fix this security issue? The error references a specific line. Open the file in Notepad and see which line it is. If you turn the Status bar from the View menu, you will be able to easily identify the line number as you click around. It looks like the problem is on the line with the following code:</i></p> <pre style="padding-left: 40px;">Response.Write objData.transformNode(objStyle)</pre> <p><i>Here the server is trying to work with the stylesheet, and it needs some permissions. In programming, the rights are often given by setting properties of objects. The following statement placed above the problem line should provide the needed correction:</i></p> <pre style="padding-left: 40px;">objStyle.setProperty "AllowXsltScript", true Response.Write objData.transformNode(objStyle)</pre> <p><i>To find out the solution to an issue you encounter while executing code written by yourself or others, Google the error code and error message to see if a solution was already posted somewhere. If not, try to look at other code that Access generated to see if you can spot any differences. In this case, I compared the Products.htm file generated by Access that contains the client-side VBScript code, and found that it contained the following line, which was not included in the server-side VBScript code:</i></p> <pre style="padding-left: 40px;">objDOM.setProperty "AllowXsltScript", true</pre> <p><i>The problem was fixed by adding the same statement to the problem file, just making sure that it references the correct object.</i></p> <p><i>If you encountered other types of errors, for example “A provider cannot be found”, you may need to download the 2007 Office System Driver Data Connectivity Components. It’s hard to foresee what kind of errors you may be faced with while working with this book, as your machine is configured differently than mine, and may be missing some older components that are still required to run on newer systems and which are never installed until they are needed. Each system is unique and the recommended troubleshooting steps may or may not work. Solving your own problems is a skill that takes time to acquire and the more troubleshooting you perform by yourself the more skillful you become.</i></p>
-------------	---

14. Repeat Step 13 if you needed to correct the code in the ASP file.
15. Close the browser and proceed to the next section.

## **APPLYING XSLT TRANSFORMS TO EXPORTED DATA**

---

When exporting Access data to XML format, you can use custom transformation files (XSL) to modify the data after you export it. Hands-On 27.6 demonstrates how to create a custom stylesheet for use after export. This stylesheet assumes that for each customer in the Customers table we want to display only selected columns from the Orders table. You learn how to apply this custom stylesheet in Hands-On 27.7.



### **Hands-On 27.6 Creating a Custom Transformation File**

1. Open Notepad and enter the following statements:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/
XSL/Transform">
<xsl:output method="html" version="4.0" indent="yes"/>

<xsl:template match="dataroot">
  <html>
    <body>
      <h2 style="font-family:Verdana">Customer Orders</h2>
      <p/>
      <xsl:apply-templates select="Customers"/>
    </body>
  </html>
</xsl:template>

<xsl:template match="Customers">
<table>
  <tr>
    <td style="background-color:#FFCC33; color:#000000;">
      <xsl:value-of select="ID"/>
    </td>
    <td><b>
      <xsl:value-of select="Company"/>
    </b></td>
  </tr>
</table>
<table cellpadding="5" cellspacing="5">
```

```
<tr style="background-color:black; color:white;">
<td style="background-color:black; width:10px;">
<td>Order ID</td>
<td>Order Date</td>
<td>Shipped Date</td>
<td>Shipping Fee</td>
</tr>
<xsl:apply-templates select="Orders"/>
</table>
</xsl:template>

<xsl:template match="Orders">
<tr>
<td style="background-color:black; width:10px;">
<td><xsl:value-of select="Order_x0020_ID"/></td>
<td><xsl:value-of select="substring(Order_x0020_Date, 1, 10)"/></td>
<td><xsl:value-of select="substring(Shipped_x0020_Date, 1, 10)"/></td>
<td>$<xsl:value-of select="format-number(Shipping_x0020_Fee,'####0.00')"/></td>
</tr>
</xsl:template>

</xsl:stylesheet>
```

2. Save the file as C:\VBAAccess2021\_XML\ListCustOrders.xsl. You must include the file extension to ensure that the file is not saved as text.
3. Close **Notepad**.

Let's now proceed to analyze the contents of the ListCustOrders.xsl file that will be used to transform XML to HTML in our next Hands-On exercise. Because the XSLT stylesheet is an XML document, we need to start out with a standard XML declaration like this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Next, we define the namespace for the stylesheet and declare its prefix like this:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

On the third line we use the `<xsl:output>` tag to indicate that XSLT should transform the XML into HTML:

```
<xsl:output method="html" version="4.0" indent="yes"/>
```

The `<xsl:output>` tag has three attributes: method, version, and indent. The method attribute specifies the format of the output. This can be XML, HTML, or text. The version attribute sets the version number for the output format. The indent attribute, which is set to “yes” in this example, indicates that the XML should be indented. This will make the final XML document more readable when viewed in the browser.

The remaining part of the XSL file contains transformation instructions for the XML document element nodes. We begin by creating the root template. The `<xsl:template>` tag initiates a template within a stylesheet. Because a template must indicate which nodes you want to use, we use the tag’s match attribute to supply the node information:

```
<xsl:template match="dataroot">
```

This tells the XSLT processor to extract the XML document’s root node. The root node provides a base node upon which we will build the HTML web page. Notice that in the root template we need to include the `<html>` and `<body>` tags to create the structure of the final document. The HTML tags such as `<h2>`, `<font>`, and `<p>` are used to add the required formatting. In the root template we are also telling the XSLT processor that it should apply the template rules found in the Customers template (defined further down in the file):

```
<xsl:apply-templates select="Customers"/>
```

When the XSLT processor encounters the `<xsl:apply-templates>` instruction, it will proceed to the following line:

```
<xsl:template match="Customers">
```

This line marks the beginning of the Customers template rule. Within it there are HTML tags as well as other XSLT processing instructions. For example, to output the ID we use the `<xsl:value-of>` tag with the select attribute like this:

```
<xsl:value-of select="ID"/>
```

Because the `<xsl:value-of>` tag does not have any content, you must end it with the forward slash (/). Notice that we placed the value of the ID field in a table cell. Using the same approach, we can output the Company column like this:

```
<xsl:value-of select="Company"/>
```

Next, we define the column headings for the Orders table. For a special effect, we add to the output a 10-pixel-wide dummy column with a black background:

```
<td style="background-color:black; width:10px;" />
```

We also tell the XSLT processor to apply the Orders template:

```
<xsl:apply-templates select="Orders"/>
```

The Orders template rules indicate how to extract values for each of the defined column headings. This is done by using the `<xsl:value-of>` tag with the `select` attribute, like this:

```
<td><xsl:value-of select="Order_x0020_ID"/></td>
<td><xsl:value-of select="substring(Order_x0020_Date,
1, 10)"/></td>
<td><xsl:value-of select="substring(Shipped_x0020_Date,
1, 10)"/></td>
<td>$<xsl:value-of select="format-number(Shipping_x0020_
Fee, '###0.00')"/></td>
```

Notice that the space found in the column name must be replaced with `_x0020_`.

To obtain only the date portion from the Order Date and Shipped Date columns, we use the XPath `substring` function in the `select` attribute. This function has the same syntax as the VBA `Mid` function, allowing you to extract a specified number of characters from a string starting at a specific position. The expression

```
<xsl:value-of select="substring(Order_x0020_Date, 1, 10)"/>
```

tells the XSLT processor to retrieve only the first 10 characters from the value found in the Order Date column. To correctly format the Shipping Fee column, the `format-number` XPath expression is used like this:

```
$<xsl:value-of select="format-number(Shipping_x0020_
Fee, '###0.00')"/>
```

This tells the XSLT processor to format the value found in the Shipping Fee column as a number using two decimal places.

Each of the defined template rules ends with the `</xsl:template>` ending tag and the stylesheet itself ends with the `</xsl:stylesheet>` tag.

This concludes your first custom stylesheets. While this is a basic stylesheet to get you started, in real life you will probably want to create stylesheets that can use the following:

- Batch-processing nodes (`<xsl:for-each>` tag with the `select` attribute)
- Conditional processing of nodes (`<xsl:if>` tag with the `test` attribute)
- Decisions based on conditions (`<xsl:choose>` tag and `<xsl:when>` tag with the `test` attribute)
- Sorting nodes before processing (`<xsl:sort>` tag with the `select` attribute)

**NOTE**

*For more information about XSL Transformations (XSLT), visit the following link: <http://www.w3.org/TR/xslt#section-Applying-Template-Rules>*

Now that you have a custom stylesheet, what can you do with it? Hands-On 27.7 demonstrates how to export data from an Access table directly to an HTML file and apply a custom transform so that only specified columns are displayed.



### Hands-On 27.7 Exporting Data and Applying a Custom XSL File

1. Make sure that you open the **C:\VBAAccess2021\_XML\Northwind 2007.accdB** database.
2. In the Navigation pane, right-click the **Customers** table and choose **Export | XML File**.
3. In the File name box, enter **C:\VBAAccess2021\_XML\ListCustOrders.xml**, and click **OK**.
4. In the Export XML dialog box, make sure that the first two checkboxes are selected. Click the **More Options** button.
5. In the Data to Export area, the **Customers** table is automatically selected. Click the checkbox next to the **Orders** table to include it in the export.
6. Click the **Transforms** button.
7. In the Export Transforms window that appears, click the **Add** button, switch to the **C:\VBAAccess2021\_XML** folder and select the **ListCustOrders.xsl** file that you created in the previous hands-on exercise. Click the **Add** button to add this file to the list of transforms. The transformation file appears in the list as shown in Figure 27.13.
8. In the Export Transforms window, click **OK**.

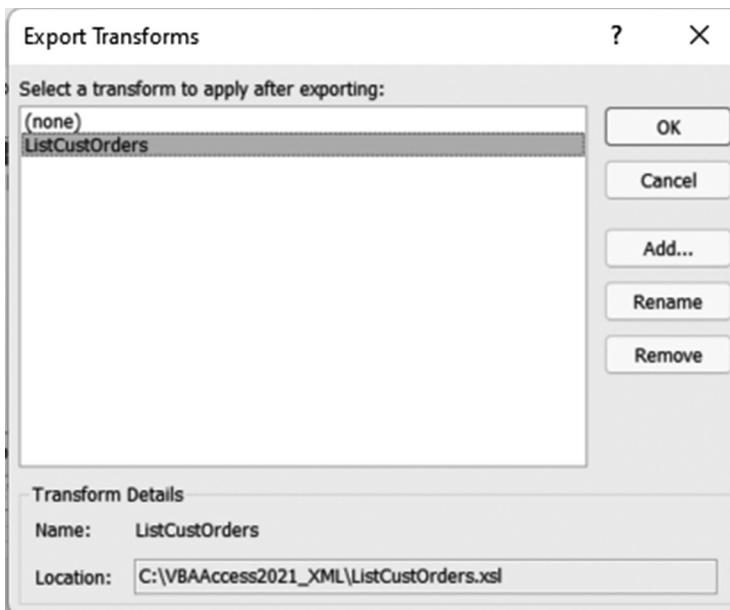


FIGURE 27.13. Use this window to indicate a transformation file (stylesheet) to be used after export.

9. Back in the Export XML dialog box, change the file extension from xml to html as shown in Figure 27.14.

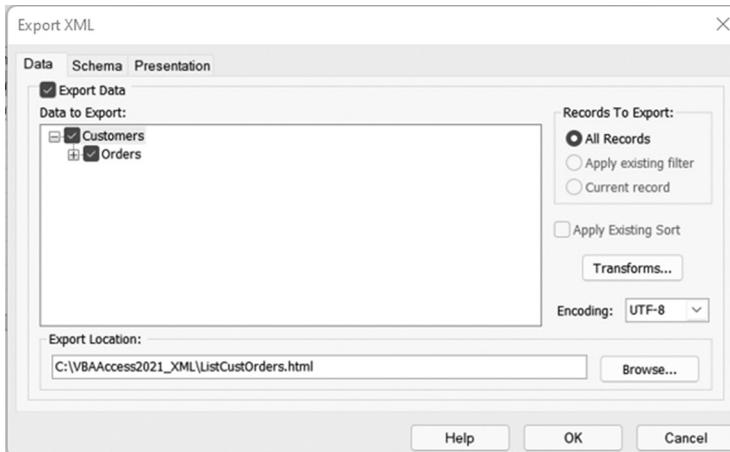


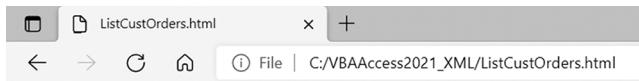
FIGURE 27.14 To export XML data directly to the HTML file, you must choose the transformation file using the Transforms button and change the file extension from xml to html.

10. Click the **OK** button to begin the export.
11. Upon successful export operation, click **Close**.

**NOTE**

If the selected transformation file is invalid, you will see an error message. Access will prompt you to save the data for troubleshooting and will bring up the Export XML dialog box. At this time you may want to open the transformation file in Notepad and make appropriate corrections. Once you save the corrected XSL file, you should return to the Export XML dialog box to try the export again. Before you click the OK button in the Export XML dialog box, ensure that the appropriate tables are selected.

12. Close the Northwind 2007 database and exit Access.
13. In the File Explorer, double-click the ListCustOrders.html file. Your output should match Figure 27.15.
14. Close the browser window.

**Customer Orders****1 Company A**

	Order ID	Order Date	Shipped Date	Shipping Fee
	44	2006-03-24		\$0.00
	71	2006-05-24		\$0.00

**2 Company B**

	Order ID	Order Date	Shipped Date	Shipping Fee
	36	2006-02-23	2006-02-25	\$7.00
	63	2006-04-25	2006-04-25	\$7.00
	81	2006-04-25		\$0.00

**3 Company C**

	Order ID	Order Date	Shipped Date	Shipping Fee
	31	2006-01-20	2006-01-22	\$5.00
	34	2006-02-06	2006-02-07	\$4.00
	58	2006-04-22	2006-04-22	\$5.00

**4 Company D**

	Order ID	Order Date	Shipped Date	Shipping Fee
	31	2006-01-20	2006-01-22	\$5.00
	34	2006-02-06	2006-02-07	\$4.00
	58	2006-04-22	2006-04-22	\$5.00
	61	2006-04-07	2006-04-07	\$4.00
	80	2006-04-25		\$0.00

**5 Company E**

	Order ID	Order Date	Shipped Date	Shipping Fee
	37	2006-03-06	2006-03-09	\$12.00

	Order ID	Order Date	Shipped Date	Shipping Fee
	37	2006-03-06	2006-03-09	\$12.00

**FIGURE 27.15.** XML data can be formatted any way you like by applying a custom transformation (see Figures 27.13 and 27.14).

## IMPORT XML DATA

You can use Access built-in Import command to import an XML data or XML schema document to a database. When you import structure or data from an XML file, Access assigns the Text data type to all the fields in a table. However, when you import structure from an XSD schema file, each field is assigned a data type that closely matches the data type specified in the schema. You can change the data types after importing data or a table structure as long as the fields' data allows such a change.

### Importing a Schema File

When you import a schema, Access creates a new empty table with the structure of the imported schema. Earlier in this chapter, when you exported the Products table to XML format, Access also created the schema of that table. Hands-On 27.8 shows how to import this schema document to a new Access database.

#### ● Hands-On 27.8 Importing a Schema File (XSD) to an Access Database

1. Create a new Access database named C:\VBAAccess2021\_XML\Chap27.accdb.
2. In the Access window, choose **External Data**, and then select **New Data Source | From File | XML File**.
3. In the Get External Data - XML File window, type C:\VBAAccess2021\_XML\Products.xsd in the File name box and click **OK**.
4. Access displays the Import XML dialog box as shown in Figure 27.16.

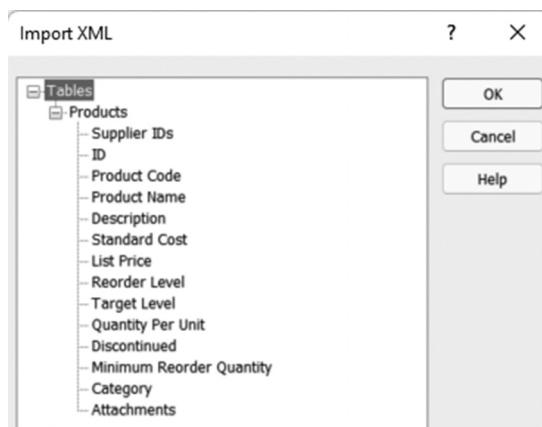


FIGURE 27.16. When importing a schema file to an Access database, the Import XML dialog box displays the table name and its columns as defined in the schema.

Notice that you cannot indicate which columns you would like to import. Access always imports the entire XSD file.

5. Click **OK** to perform the import. When the import operation is completed, click **Close**. The Products table appears in the Navigation pane of the Access window. Figure 27.17 shows this table opened in Design view.
6. Close the Chap27.accdb database.

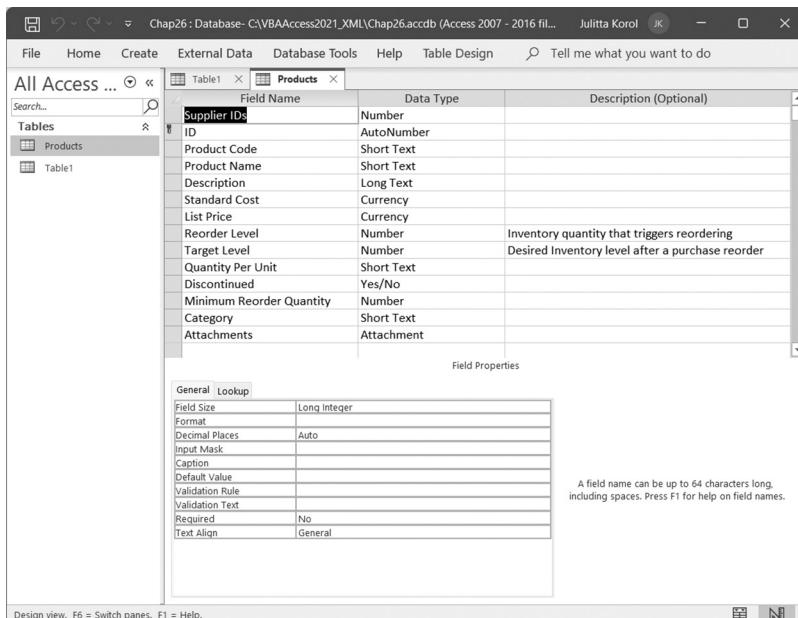


FIGURE 27.17. The Shippers table was created by importing the Shippers.xsd schema file.

## Importing an XML File

When importing an XML data file to an Access database, you can use the Import Options section to specify whether you want to import structure only, import structure and data, or append data to an existing table (see Figure 27.18). When you append data to an existing table, Access compares the structure of the imported table with the table structures that are already in the database. If Access cannot find a table structure matching the imported table, the data is placed in a new table; otherwise, it is appended to the existing table. You can also click the Transform button in the Import XML dialog box to specify a transformation file that you want to apply when the XML data is imported.

It is important to point out that when XML data is imported to an Access database, it is not linked with the original XML file. This means that to refresh the data in the table, you need to repeat the import process.

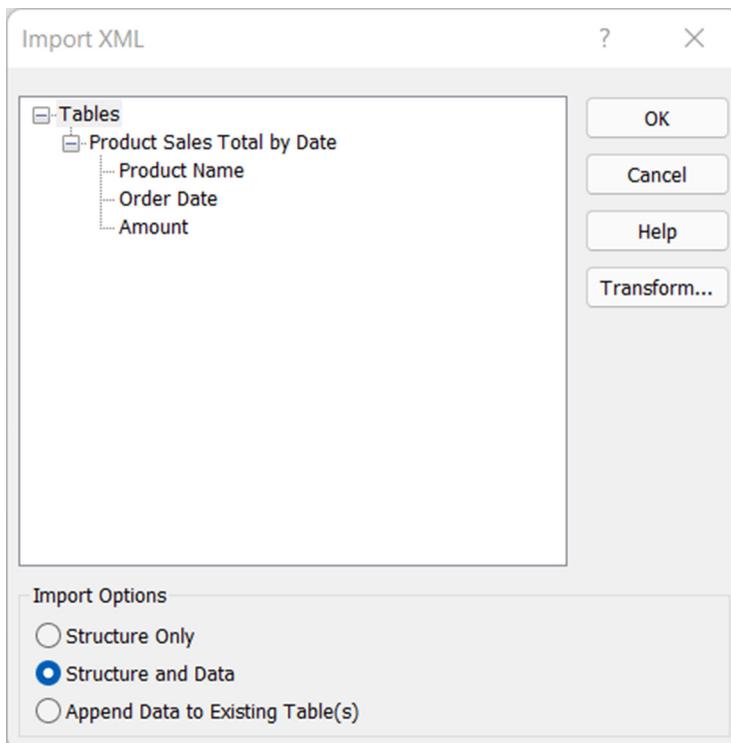


FIGURE 27.18 Importing XML data (Products Sales Total by Date.xml) to the Chap27.accdb database.

The following project demonstrates how to import XML data to an Access database and modify the data before import using a transformation file. We will perform the tasks outlined here:

- Create a custom transformation file to be used after the XML data import
- Export the Customers table and the related Orders table to an XML file
- Import to an Access database only two columns from the Customers table and five columns from the Orders table



## Custom Project 27.1 Importing XML Data to an Access Database and Applying a Transform

### ***Part 1: Creating a Custom Transformation File to be Used After the XML Data Import***

---

1. Open Notepad and enter the following statements:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" version="4.0" indent="yes"/>

<xsl:template match="dataroot">
  <html>
    <body>
      <table>
        <xsl:apply-templates select="Customers"/>
      </table>
      <table>
        <xsl:apply-templates select="Customers/Orders"/>
      </table>
    </body>
  </html>
</xsl:template>

<xsl:template match="Customers">
  <Customer>
    <ID>
      <xsl:value-of select="ID"/>
    </ID>
    <Company>
      <xsl:value-of select="Company"/>
    </Company>
  </Customer>
</xsl:template>

<xsl:template match="Customers/Orders">
  <Order>
    <OrderID>
      <xsl:value-of select="Order_x0020_ID"/>
    </OrderID>
    <CustomerID>
      <xsl:value-of select="Customer_x0020_ID"/>
    </CustomerID>
```

```
<OrderDate>
<xsl:value-of select="substring(Order_x0020_Date, 1, 10)"/>
</OrderDate>
<ShippedDate>
<xsl:value-of select="substring(Shipped_x0020_Date, 1, 10)"/>
</ShippedDate>
<ShippingFee>
<xsl:value-of select="format-number(Shipping_x0020_Fee, '####0.00')"/>
</ShippingFee>
</Order>
</xsl:template>

</xsl:stylesheet>
```

2. Save the file as **C:\VBAAccess2021\_XML\CustomerOrders.xsl**. You must include the file extension to ensure that the file is not saved as text.
3. Close Notepad.

Since you've already created a similar stylesheet in Hands-On 27.6, you should be familiar with the contents of the CustomerOrders.xsl file. All that's different here are the `<Customer>` and `<Order>` tags that specify the names of Access tables where we want to place our XML data. When importing data, tables are named according to the name of the XML element being imported. If the Access database already has a table with the specified name, a number is appended to the name.

---

### ***Part 2: Exporting the Customers and Related Orders Tables to an XML File***

---

1. Open the **C:\VBAAccess2021\_XML\Northwind 2007.accdb** database and log in as Andrew Cencini.
2. In the Navigation pane, right-click the **Customers** table and choose **Export | XML File**.
3. In the Export - XML File window, type **C:\VBAAccess2021\_XML\CustomerOrders.xml** in the File name box and click **OK**.
4. Access displays the Export XML dialog box with three checkboxes; the first two checkboxes should be selected. Click the **More Options** button.
5. In the Data to Export area of the Export XML dialog box, select the checkbox next to the **Orders** table. The Customers and Orders tables should both be selected.
6. Click **OK** to perform the export of all the records in the selected tables. When the export operation is completed, click **Close**.
7. Close the Northwind.accdb database file.

***Part 3: Importing to an Access Database Only Two Columns from the Customers Table and Five Columns from the Orders Table***

---

1. Open the C:\VBAAccess2021\_XML\Chap27.accdb database file that you created in Hands-On 27.8.
2. In the Access window, choose **External Data**, and then select **New Data Source | From File | XML File**.
3. In the Get External Data - XML File window, type **C:\VBAAccess2021\_XML\CustomerOrders.xml** in the File name box and click **OK**.  
Access displays the Import XML window with the file's Customers and Orders tables listed. By expanding nodes in the tree structure, you can see the columns in each table, but you cannot indicate which columns to import, as Access always imports the entire file by default. You can, however, tell Access to perform a custom XSLT transform to import only the columns needed.
4. In the Import XML window, click the **Transform** button.
5. In the Import Transforms window that appears, click the **Add** button to apply a transform before importing.  
Access displays the Add New Transform window. Switch to the **VBAAccess2021\_XML** folder and select the **CustomerOrders.xsl** stylesheet file that you created in Part 1 of this project. Click the **Add** button to add this file to the list of transforms.
6. In the Import Transforms window, click **OK**.
7. Back in the Import XML window, make sure that the **Structure and Data** option button is selected under Import Options and click **OK**. When Access finishes importing the C:\VBAAccess2021\_XML\CustomerOrders.xml document, click **Close**.
8. In the Navigation pane of the Access window, notice the appearance of two new tables: Customer and Order. Open both tables and check their contents. As you can see, Access has applied the custom stylesheet before importing the data and only the columns specified in the stylesheet were imported (see Figure 27.19).
9. Open the **Order** table in Design view. Notice that all the fields in this table have been assigned the **Text** data type.
10. After importing data or table structure you can change the fields' data types.
11. Change the data type of the **OrderID** and **CustomerID** to **Number**, **OrderDate** and **ShippedDate** columns to **Date/Time** and the **ShippingFee** column's data type to **Currency** to match the original Orders table.

12. Save the modified Order table and click Yes when Access notifies that that some data may be lost.
13. Close the Chap27.accdb database.

The screenshot shows the Microsoft Access 2021 interface. The title bar reads "Chap26 : Database- C:\VBAAccess2021\_XML\Chap26.accdb... Julitta Korol JK". The ribbon menu includes File, Home, Create, External Data, Database Tools, Help, Table Fields, Table, and Tell me. On the left, the "All Access Objects" pane shows a list of tables: Customer, Order, Product Sales Total by Date, and Products. The "Customer" table is currently selected. In the main workspace, the "Customer" table is displayed in Datasheet View. It has two columns: "ID" and "Company". The "ID" column contains integers from 1 to 12. The "Company" column contains strings: Company A, Company B, Company C, Company D, Company E, Company F, Company G, Company H, Company I, Company J, Company K, and Company L. At the bottom of the datasheet, there are navigation buttons for records 14, 1 of 29, 21, 22, and 23, along with a "No Filter" button and a "Search" input field. The status bar at the bottom right shows "Datasheet View".

**FIGURE 27.19** Applying a custom transform prior to XML data import will allow you to limit the number of imported columns.

## PROGRAMMATICALLY EXPORTING TO AND IMPORTING FROM XML

---

Now that you've mastered the use of Access 2021 built-in commands for exporting and importing XML data, let's look at what tools are available for programmers who want to perform these XML operations via code. In the following sections of this chapter, you will learn how to work with XML using:

- The `ExportXML` and `ImportXML` methods from the Microsoft Access 16.0 Object Library
- The `TransformXML` method

### Exporting to XML Using the ExportXML Method

---

Use the Microsoft Access 16.0 Object Library `ExportXML` method of the Application object to export XML data, schemas (XSD), and presentation information (XSL) from a Microsoft Access database, Microsoft SQL Server 2000 Desktop Engine (MSDE 2000), or Microsoft SQL Server 6.5 or later.

The `ExportXML` method takes a number of arguments, which are shown in Table 27.1.

In its simplest form, the `ExportXML` method looks like this:

```
Application.ExportXML ObjectType:=acExportTable, _
DataSource:="Customers", _
DataTarget:= "C:\VBAAccess2021_XML\North_Customers.xml"
```

The preceding statement, when typed on a single line (without the underscore characters) in the Visual Basic Editor's Immediate window or inside a VBA procedure stub in a Visual Basic module, will render the Customers table in the `North_Customers.xml` file.

Using the arguments described in Table 27.1, you can easily write the command to export the XML Products table with its schema and presentation information placed in separate files:

```
Application.ExportXML ObjectType:=acExportTable, _
DataSource:="Products", _
DataTarget:= "C:\VBAAccess2021_XML\North_Products.xml", _
SchemaTarget:= "C:\VBAAccess2021_XML\North_ProdSchema.xsd", _
PresentationTarget:= "C:\VBAAccess2021_XML\North_ProdReport.xsl"
```

To export a specific customer's data to an XML data file, use the following statement:

```
Application.ExportXML ObjectType:=acExportTable, _
DataSource:="Customers", _
DataTarget:= "C:\VBAAccess2021_XML\OneCustomer.xml", _
WhereCondition:="ID = 4"
```

<b>NOTE</b>	<p><i>To try out the preceding statements,</i></p> <ul style="list-style-type: none"> <li>● <i>Open the Northwind 2007.accdb database as Andrew Cencini</i></li> <li>● <i>Switch to the Visual Basic Editor window (Choose Database Tools   Visual Basic)</i></li> <li>● <i>Insert a new standard module (Insert   Module)</i></li> <li>● <i>Create sub procedure named Test_ExportToXML (choose Insert   Procedure, enter the procedure name and click OK)</i></li> <li>● <i>Type each of the preceding statements inside the procedure stub</i></li> <li>● <i>Execute the procedure (Run   Run Sub/UserForm)</i></li> <li>● <i>Locate and check out the newly created XML files in your C:\VBAAccess2021_XML folder</i></li> </ul>
-------------	--

TABLE 27.1 Arguments of the `ExportXML` method (in order of appearance)

Argument Type	Data Type / Description																
ObjectType (required)	<p>AcExportXMLObjectType</p> <p>Use one of the following constants:</p> <table border="1" data-bbox="475 369 942 675"> <thead> <tr> <th data-bbox="475 369 857 399">Constant</th><th data-bbox="857 369 942 399">Value</th></tr> </thead> <tbody> <tr> <td data-bbox="475 399 857 433">acExportForm</td><td data-bbox="857 399 942 433">2</td></tr> <tr> <td data-bbox="475 433 857 468">acExportFunction</td><td data-bbox="857 433 942 468">10</td></tr> <tr> <td data-bbox="475 468 857 502">acExportQuery</td><td data-bbox="857 468 942 502">1</td></tr> <tr> <td data-bbox="475 502 857 537">acExportReport</td><td data-bbox="857 502 942 537">3</td></tr> <tr> <td data-bbox="475 537 857 571">acExportServerView</td><td data-bbox="857 537 942 571">7</td></tr> <tr> <td data-bbox="475 571 857 606">acExportStoredProcedure</td><td data-bbox="857 571 942 606">9</td></tr> <tr> <td data-bbox="475 606 857 641">acExportTable</td><td data-bbox="857 606 942 641">0</td></tr> </tbody> </table> <p>Specifies the type of Access object to export. The constant values 10, 7, and 9 are used only with Microsoft Access projects.</p>	Constant	Value	acExportForm	2	acExportFunction	10	acExportQuery	1	acExportReport	3	acExportServerView	7	acExportStoredProcedure	9	acExportTable	0
Constant	Value																
acExportForm	2																
acExportFunction	10																
acExportQuery	1																
acExportReport	3																
acExportServerView	7																
acExportStoredProcedure	9																
acExportTable	0																
DataSource (required)	<p>String</p> <p>Indicates the name of the Access object specified in the Object-Type argument.</p>																
DataTarget (optional)	<p>String</p> <p>Specifies the path and filename for the exported data. Omit this argument only if you don't want the data to be exported.</p>																
SchemaTarget (optional)	<p>String</p> <p>Specifies the path and filename for the exported schema information. Omit this argument only if you don't want the schema to be exported to a separate file.</p>																
PresentationTarget (optional)	<p>String</p> <p>Specifies the path and filename for the exported presentation information. Omit this argument only if you don't want the presentation information to be exported.</p>																
ImageTarget (optional)	<p>String</p> <p>Specifies the path for the exported images. Omit this argument if you don't want to export images.</p>																
Encoding (optional)	<p>AcExportXMLEncoding</p> <p>Use one of the following constants:</p> <table border="1" data-bbox="475 1401 753 1517"> <thead> <tr> <th data-bbox="475 1401 723 1430">Constant</th><th data-bbox="723 1401 753 1430">Value</th></tr> </thead> <tbody> <tr> <td data-bbox="475 1430 723 1465">acUTF16</td><td data-bbox="723 1430 753 1465">1</td></tr> <tr> <td data-bbox="475 1465 723 1499">acUTF8</td><td data-bbox="723 1465 753 1499">0</td></tr> </tbody> </table> <p>The default is acUTF8. Specifies the text encoding for the exported data.</p>	Constant	Value	acUTF16	1	acUTF8	0										
Constant	Value																
acUTF16	1																
acUTF8	0																

Argument Type	Data Type / Description														
OtherFlags (optional)	<p>AcExportXMLOtherFlags Use one or more of the following constants:</p> <table border="1"> <thead> <tr> <th>Constant</th><th>Value</th></tr> </thead> <tbody> <tr> <td>acEmbedSchema</td><td>1</td></tr> <tr> <td>acExcludePrimaryKeyAndIndexes</td><td>2</td></tr> <tr> <td>acExportAllTableAndFieldProperties</td><td>32</td></tr> <tr> <td>acLiveReportSource</td><td>8</td></tr> <tr> <td>acPersistReportML</td><td>16</td></tr> <tr> <td>acRunFromServer</td><td>4</td></tr> </tbody> </table> <p>Specifies behaviors associated with exporting to XML. Values can be added to specify a combination of behaviors. Here are the meanings of the constants:</p> <ul style="list-style-type: none"> <li>(1) Write schema information into a separate document specified by the DataTarget argument. This value takes precedence over the SchemaTarget argument.</li> <li>(2) Does not export primary key and index schema properties.</li> <li>(32) The exported schema contains properties of the table and its fields.</li> <li>(8) Used only when exporting reports bound to SQL Server 2000. Will create a live link to a Microsoft SQL Server database.</li> <li>(16) Persists the exported object's ReportML file.</li> <li>(4) Used only when exporting reports. Creates an Active Server Pages (ASP) or HTML wrapper. The default is HTML.</li> </ul>	Constant	Value	acEmbedSchema	1	acExcludePrimaryKeyAndIndexes	2	acExportAllTableAndFieldProperties	32	acLiveReportSource	8	acPersistReportML	16	acRunFromServer	4
Constant	Value														
acEmbedSchema	1														
acExcludePrimaryKeyAndIndexes	2														
acExportAllTableAndFieldProperties	32														
acLiveReportSource	8														
acPersistReportML	16														
acRunFromServer	4														
WhereCondition (optional)	<p>String Specifies a subset of records to export.</p>														
AdditionalData (optional)	<p>AdditionalData AdditionalData is an Access object that represents the collection of tables and queries that will be included with the parent table that is exported by the ExportXML method (see Hands-On 27.9). Specifies additional tables to export. This argument is ignored if the OtherFlags argument is set to acLiveReportSource (8).</p>														

Hands-On 27.9 demonstrates how to export to XML three tables: Customers, Orders, and Order Details.



### Hands-On 27.9 Exporting Multiple Tables to an XML Data File

- In the C:\VBAAccess2021\_XML\Chap27.accdb database, switch to the Visual Basic Editor window.

2. Choose **Insert | Module** to add a standard module to the current VBA project.
3. In the module's Code window, enter the following **Export\_CustomerOrderDetails** procedure:

```
Sub Export_CustomerOrderDetails()
    Dim objAppl As New Access.Application
    Dim objOtherTbls As AdditionalData
    Dim strPath As String
    Dim strDBName As String

    strPath = "C:\VBAAccess2021_XML\""
    strDBName = "Northwind.mdb"

    On Error GoTo ErrorHandler
    objAppl.OpenCurrentDatabase (strPath & strDBName)
    objAppl.Visible = False

    Set objOtherTbls = objAppl.CreateAdditionalData

    ' include the Orders and OrderDetails tables
    ' in export
    objOtherTbls.Add "Orders"
    objOtherTbls.Add "Order Details"

    ' export Customers, Orders, and Order
    ' Details table into one XML data file

    objAppl.ExportXML ObjectType:=acExportTable, _
        DataSource:="Customers", _
        DataTarget:=strPath & "CustomerOrdersDetails.xml", _
        AdditionalData:=objOtherTbls

    MsgBox "Export operation completed."

    Exit_Here:
    On Error Resume Next
    objAppl.CloseCurrentDatabase
    Set objAppl = Nothing
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ": " & Err.Description
    Resume Exit_Here
End Sub
```

4. Place the insertion point anywhere within the **Export\_CustomerOrderDetails** procedure code and choose **Run | Run Sub/UserForm**.

Access executes the procedure code and displays a message.

5. Click **OK** to clear the informational message.
6. Switch to File Explorer and locate and open the **C:\VBAAccess2021\_XML\CustomerOrdersDetails.xml** file.

Notice that all the requested data was placed into one file. If the expected xml file was not created, double-check the spelling of variables in your procedure. Make sure that the Option Explicit statement appears at the top of the module, above your procedure code.

7. Exit the File Explorer.

In the above procedure code, Application object refers to the active Access application, which in this case, is the Chap27.accdb database where you wrote the procedure code shown here. Because this database does not contain the tables you want to export, you used the `New` keyword to create a new instance of the Access Application object and then opened another Access database (Northwind 2007.accdb) using the `OpenCurrentDatabase` method. You can use the `OpenCurrentDatabase` method to open an existing Access database as the current database.

Using the AdditionalData object, you can export any set of Access tables to an XML data file. To use this object, follow these steps:

- Declare an object variable as AdditionalData:

```
Dim objOtherTbls As AdditionalData
```

- Create the AdditionalData object using the `CreateAdditionalData` method of the Application object and set the object variable to the newly created object:

```
Set objOtherTbls = objAppl.CreateAdditionalData
```

- Use the AdditionalData object's `Add` method to add table names to the object:

```
objOtherTbls.Add "Orders"  
objOtherTbls.Add "Order Details"
```

- Pass the AdditionalData object to the `ExportXML` method:

```
objAppl.ExportXML ObjectType:=acExportTable, _  
DataSource:="Customers", _  
DataTarget:=strPath & "CustomerOrdersDetails.xml", _  
AdditionalData:=objOtherTbls
```

## Transforming XML Data with the TransformXML Method

---

While stylesheets are often used to render XML files into HTML for display in a Web browser, they can also be used to transform XML files into other XML files.

In this section, you will learn how the `TransformXML` method is used to apply an XSL stylesheet to an XML data file to transform it into another XML file.

The `TransformXML` method takes a number of arguments, which are presented in Table 27.2. In its simplest form, the `TransformXML` method looks like this:

```
Application.TransformXML _ DataSource:="C:\VBAAccess2021_XML\  
InternalContacts.xml", _  
TransformSource:="C:\VBAAccess2021_XML\Extensions.xsl", _  
OutputTarget:="C:\VBAAccess2021_XML\EmpExtensions.xml"
```

The preceding statement can be used inside a VBA procedure stub to programmatically apply the specified stylesheet.

**TABLE 27.2** Arguments of the `TransformXML` method (in order of appearance)

Argument Type	Data Type	Description								
DataSource (required)	String	Specifies the full path of the XML data file that will be transformed.								
TransformSource (required)	String	Specifies the full path of the XSL stylesheet to apply to the XML data file specified in the <code>DataSource</code> argument.								
OutputTarget (required)	String	Specifies the full path of the resulting XML data file after applying the XSL stylesheet.								
WellFormedXMLOutput (optional)	Boolean	Set this argument to True to create a well-formed XML document. Set this argument to False to encode the resulting XML file in UTF-16 format. The default is False.								
ScriptOption (optional)	<p>AcTransformXMLScriptOption</p> <p>Use one of the following constants:</p> <table border="1"> <tr> <td>Constant</td> <td>Value</td> </tr> <tr> <td>acDisableScript</td> <td>2</td> </tr> <tr> <td>acEnableScript</td> <td>0</td> </tr> <tr> <td>acPromptScript</td> <td>1</td> </tr> </table>	Constant	Value	acDisableScript	2	acEnableScript	0	acPromptScript	1	Use this argument to specify the action that should be taken if the XSL file contains scripting code. <code>acPromptScript</code> is the default.
Constant	Value									
acDisableScript	2									
acEnableScript	0									
acPromptScript	1									

Custom Project 27.2 demonstrates how to transform an XML data file into another XML file. We will start by creating a custom stylesheet named Extensions.xsl that will transform the InternalContacts.xml file (generated from the Northwind.mdb database Employees table) into an XML file named EmpExtensions.xml. Next, we will write a VBA procedure to export the XML source file and perform the transformation. Finally, we will import the resulting XML data file into Access.



### Custom Project 27.2 Applying a Stylesheet to an XML Data File with the TransformXML Method

---

#### *Part 1: Creating a Custom Stylesheet for Transforming an XML Source File into Another XML Data File*

---

1. Open Notepad and enter the following statements:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/
XSL/Transform">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="/">
<dataroot>
<xsl:for-each select="//Employees">
<Extensions>
    <LastName>
        <xsl:value-of select="LastName" />
    </LastName>
    <FirstName>
        <xsl:value-of select="FirstName" />
    </FirstName>
    <Extension>
        <xsl:value-of select="Extension" />
    </Extension>
</Extensions>
</xsl:for-each>
</dataroot>
</xsl:template>
</xsl:stylesheet>
```

Look at the preceding stylesheet and notice that we have asked the XSL processor to produce the output in XML format:

```
<xsl:output method="xml" indent="yes"/>
```

Next, we used the following instruction:

```
<xsl:template match="/">
```

This instruction defines a template for the entire document. The special pattern “/” in the match attribute tells the XSL processor that this is a template for the document root. Because each XML document must have a root node, we proceeded to define `<dataroot>` as the document root. You can use any name you want for this purpose. Next, we told the XSL processor to get all the Employees nodes from the source XML data file:

```
<xsl:for-each select="//Employees">
```

The first forward slash in the preceding instruction represents the XML document root. This is the same as:

```
<xsl:for-each select="dataroot/Employees">
```

Next, we proceed to extract data from the required nodes. We are only interested in three columns from the source XML data file: FirstName, LastName, and Extension. We create the necessary elements using the `<xsl:value-of>` tag with the select attribute specifying the element name:

```
<LastName>
  <xsl:value-of select="LastName" />
</LastName>
<FirstName>
  <xsl:value-of select="FirstName" />
</FirstName>
<Extension>
  <xsl:value-of select="Extension" />
</Extension>
```

We tell the XSL processor to place the defined elements under the `<Extensions>` node. When importing the resulting XML file to Access, Access will create an Extensions table with three columns: LastName, FirstName, and Extension. You can use any name you want when specifying the container node for your elements.

To finish the stylesheet, we must write the necessary closing tags:

```
</xsl:for-each>
</dataroot>
</xsl:template>
</xsl:stylesheet>
```

2. Save the file as C:\VBAAccess2021\_XML\Extensions.xsl. You must include the .xsl file extension to ensure that the file is not saved as text.

3. Close Notepad.

Now that we've got the stylesheet ready, let's write a VBA procedure to actually export the source data and perform the transformation.

***Part 2: Writing a VBA Procedure to Export and Transform Data***

---

1. Copy the Northwind.mdb database from the companion files to your C:\VBAAccess2021\_XML folder.
2. In the Chap27.accdb database, choose **External Data** and click **New Data Source | From Database | Access**.
3. In the File name box, enter C:\VBAAccess2021\_XML\Northwind.mdb and click **OK**.
4. In the Import Object dialog box, click the Tables tab, select **Employees**, and click **OK**.
5. Click **Close** to exit the Get External Data – Access Database dialog box.  
You should see the Employees table in the Navigation pane.
6. Choose **Database Tools | Visual Basic**.
7. In the Visual Basic Editor window, choose **Insert | Module** to add a standard module to the current VBA project.
8. In the module's Code window, enter the following **Transform\_Employees** procedure:

```
Sub Transform_Employees()
Dim strPath As String
    ' use the ExportXML method to
    ' create a source XML data file

strPath = "C:\VBAAccess2021_XML\
Application.ExportXML _
    ObjectType:=acExportTable, _
    DataSource:="Employees", _
    DataTarget:=strPath & "InternalContacts.XML"

MsgBox "The export operation completed.

    ' use the TransformXML method
    ' to apply the stylesheet
    ' that transforms the source
    ' XML data file into
    ' another XML data file
Application.TransformXML _
    DataSource:=strPath & "InternalContacts.xml", _
```

```

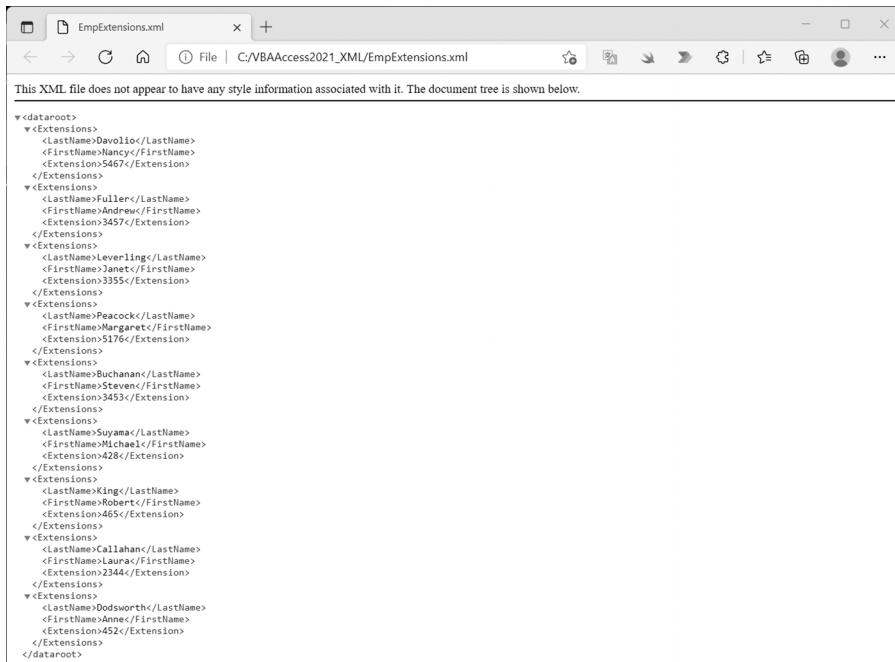
TransformSource:=strPath & "Extensions.xsl", _
OutputTarget:=strPath & "EmpExtensions.xml", _
WellFormedXMLOutput:=False

MsgBox "The transform operation completed."
End Sub

```

The first part of this procedure exports the Employees table to an XML file named InternalContacts.xml. The second part of this procedure applies the Extensions.xsl stylesheet prepared in Part 1 of this custom project to the InternalContacts.xml data file. The resulting XML document after the transformation is named EmpExtensions.xml. A portion of this file is shown in Figure 27.20.

9. Choose **Run | Run Sub/UserForm** or press **F5** to run the **Transform\_Employees** procedure.
10. Save changes in the module and exit the Visual Basic Editor window.



```

<dataroot>
  <Extensions>
    <Extension>
      <LastName>Davolio</LastName>
      <FirstName>Nancy</FirstName>
      <Extension>5467</Extension>
    </Extension>
    <Extension>
      <LastName>Fuller</LastName>
      <FirstName>Andrew</FirstName>
      <Extension>3457</Extension>
    </Extension>
    <Extension>
      <LastName>Leverling</LastName>
      <FirstName>Janet</FirstName>
      <Extension>3355</Extension>
    </Extension>
    <Extension>
      <LastName>Peacock</LastName>
      <FirstName>Margaret</FirstName>
      <Extension>1276</Extension>
    </Extension>
    <Extension>
      <LastName>Buchanan</LastName>
      <FirstName>Steven</FirstName>
      <Extension>3453</Extension>
    </Extension>
    <Extension>
      <LastName>Suyama</LastName>
      <FirstName>Michael</FirstName>
      <Extension>428</Extension>
    </Extension>
    <Extension>
      <LastName>King</LastName>
      <FirstName>Robert</FirstName>
      <Extension>465</Extension>
    </Extension>
    <Extension>
      <LastName>Callahan</LastName>
      <FirstName>Laura</FirstName>
      <Extension>2344</Extension>
    </Extension>
    <Extension>
      <LastName>Dundsworth</LastName>
      <FirstName>Anne</FirstName>
      <Extension>452</Extension>
    </Extension>
  </Extensions>
</dataroot>

```

**FIGURE 27.20.** Partial contents of the EmpExtensions.xml file.

After transforming the source XML data file into another XML document, it's time to bring it into Access.

---

***Part 3: Importing the Transformed XML Data File to Access***

---

1. In the Chap27.accdb, choose **External Data | New Data Source | From File | XML File**.
2. In the File name box, type **C:\VBAAccess2021\_XML\EmpExtensions.xml** and click **OK**. Access displays the Import XML dialog box listing the Extensions table with three columns: LastName, FirstName, and Extension.
3. In the Import XML dialog box, click **OK** to perform the import.
4. Click **Close** to exit the Import XML window.
5. In the Navigation pane of the Access window, notice the appearance of the Extensions table. Open the **Extensions** table to examine its contents.
6. Close the Extensions table.

A nice thing about XSLT transformations is that you can apply different stylesheets to the same XML data file to create and view the resulting document in different formats.

For example, let's assume that in the Extensions table you'd like to combine the LastName and FirstName columns into one column and sort the data by LastName. To do this, you could create the following **Extensions\_SortByEmp.xsl** stylesheet and apply it to the InternalContacts.xml.

---

***Part 4: Creating another transformation***

---

1. Open Notepad and prepare the following stylesheet.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
    <dataroot>
      <xsl:apply-templates select="Employees" />
      <xsl:sort select="LastName" order="ascending" />
    </xsl:apply-templates>
  </dataroot>
  </xsl:template>

  <xsl:template match="//Employees">
    <Extensions>
      <FullName>
        <xsl:value-of select="LastName" />
        <xsl:text>, </xsl:text>
        <xsl:value-of select="FirstName" />
      </FullName>
```

```

<Extension>
<xsl:value-of select="Extension" />
</Extension>
</Extensions>
</xsl:template>
</xsl:stylesheet>

```

**2.** Save the stylesheet as **C:\VBAAccess2021\_XML\Extensions\_SortByEmp.xsl**.

The preceding stylesheet uses the `<xsl:apply-templates>` tag to tell the XSL processor to select the child elements of the `dataroot/Employees` node. For each child element, it will find in the stylesheet the matching template rule and process it:

```

<xsl:apply-templates select="dataroot/Employees">
    <xsl:sort select="LastName" order="ascending" />
</xsl:apply-templates>

```

The `<xsl:sort>` tag specifies how the resulting XML document should be sorted. The `select` attribute of this tag is set to `LastName`, indicating that the file should be sorted by the `LastName` element. The `order` attribute defines the sort order as ascending.

Next, the stylesheet uses the template rule that begins with the `<xsl:template>` tag. Its `match` attribute specifies which nodes in the document tree the template rule should process:

```
<xsl:template match="//Employees">
```

The `//Employees` expression in the `match` attribute is equivalent to `dataroot/Employees`.

Next, you need to define the document node in the output file as `Extensions`, and proceed to define its child elements as `FullName` and `Extension`:

```

<Extensions>
    <FullName>
        <xsl:value-of select="LastName" />
        <xsl:text>, </xsl:text>
        <xsl:value-of select="FirstName" />
    </FullName>
    <Extension>
        <xsl:value-of select="Extension" />
    </Extension>
</Extensions>

```

The `FullName` element should contain the last name of the employee followed by a space and the first name. You can obtain the values of these fields with the

<xsl:value-of> tag and use the <xsl:text></xsl:text> tag pair to output a comma followed by a space between the last name and first name. Since there is nothing special about the Extension element, you can simply use the <xsl:value-of> tag to obtain this element's value.

Finally, you must complete the template and the stylesheet with the required closing tags:

```
</xsl:template>
</xsl:stylesheet>
```

3. To apply the preceding stylesheet to the source XML file, switch to the Visual Basic Editor window and enter the following VBA procedure.

```
Sub Transform_ContactsSort()
    Dim strPath As String

    strPath = "C:\VBAAccess2021_XML\"

    ' use the ExportXML method to create
    ' a source XML data file
    ' objAppl.ExportXML ObjectType:=acExportTable, _
    Application.ExportXML ObjectType:=acExportTable, _
    DataSource:="Employees", _
    DataTarget:=strPath & "InternalContacts.xml"

    ' use the TransformXML method
    ' to apply the stylesheet that
    ' transforms the source XML data
    ' file into another XML data file

    Application.TransformXML
    DataSource:=strPath & "InternalContacts.xml", _
    TransformSource:=strPath & "Extensions_SortByEmp.xsl", _
    OutputTarget:=strPath & "EmpExtensions.xml", _
    WellFormedXMLOutput:=False

End Sub
```

4. Choose **Run | Run Sub/UserForm** or press **F5** to run the **Transform\_ContactsSort** procedure.
5. Save changes in the module and exit the Visual Basic Editor window.
6. Follow the steps from Part 3 above to import the **EmpExtensions.xml** file to Access.

You should see the Extensions1 table in the database window. When opened, this table displays a sorted list of employees with their extensions (see Figure 27.21).

	FullName	Extension
Buchanan, Steven	3453	
Callahan, Laura	2344	
Davolio, Nancy	5467	
Dodsworth, Anne	452	
Fuller, Andrew	3457	
King, Robert	465	
Leverling, Janet	3355	
Peacock, Margaret	5176	
Suyama, Michael	428	

FIGURE 27.21. The Extensions table after it was reformatted with another stylesheet.

## Importing to XML Using the ImportXML Method

Use the `ImportXML` method to programmatically import an XML data file and/or schema file. The `ImportXML` method takes two arguments, as shown in Table 27.3.

TABLE 27.3 Arguments of the `ImportXML` method (in order of appearance)

Argument Type	Data Type	Description								
DataSource (required)	String	Specifies the full path of the XML file to import.								
ImportOptions (optional)	<p>acImportXMLOption Use one of the following constants:</p> <table border="1"> <tr> <td>Constant</td> <td>Value</td> </tr> <tr> <td>acAppendData</td> <td>2</td> </tr> <tr> <td>acStructureAnd- Data</td> <td>1</td> </tr> <tr> <td>acStructureOnly</td> <td>0</td> </tr> </table>	Constant	Value	acAppendData	2	acStructureAnd- Data	1	acStructureOnly	0	Specifies whether to import structure only (0), import structure and data (1) (default), or append data (2).
Constant	Value									
acAppendData	2									
acStructureAnd- Data	1									
acStructureOnly	0									

The following procedure will import the structure of the Extensions table from the EmpExtensions.xml file:

```
Sub Import_XMLFile()
    Application.ImportXML
        DataSource:="C:\VBAAccess2021_XML\EmpExtensions.xml", _
```

```
ImportOptions:=acStructureOnly  
End Sub
```

The result should be an empty Extensions2 table in your Chap27.accdb database.

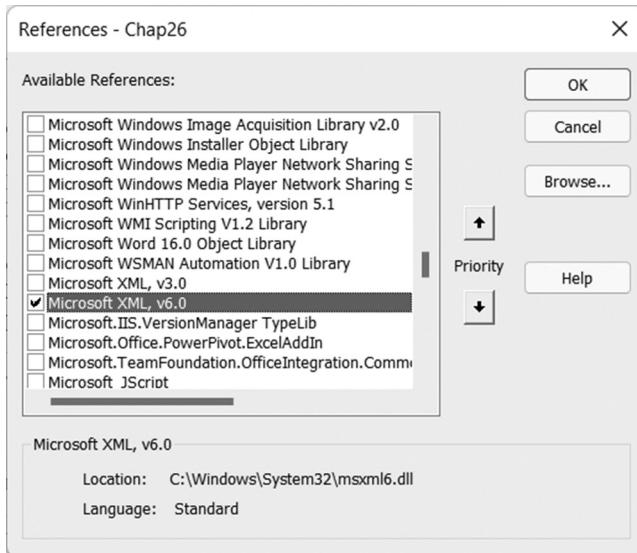
## **MANIPULATING XML DOCUMENTS PROGRAMMATICALLY**

You can create, access, and manipulate XML documents programmatically using the XML Document Object Model (DOM). The DOM has objects, properties, and methods for interacting with XML documents.

To use the XML DOM from your VBA procedures, take a few minutes now to set up a reference to the MSXML Object Library:

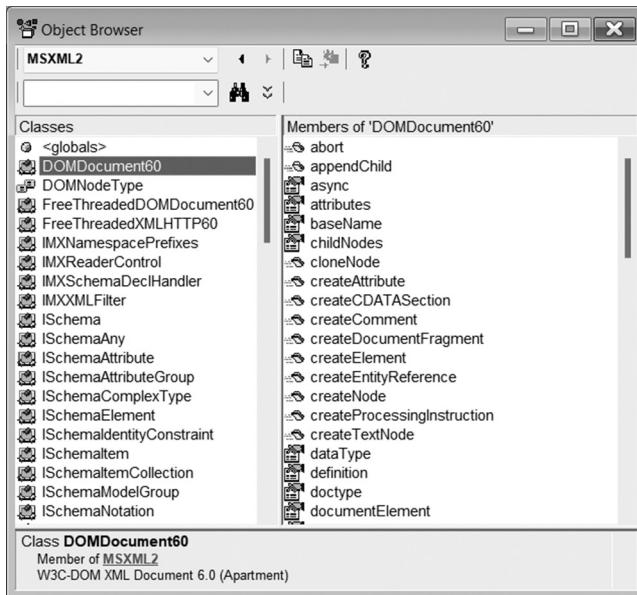
1. Switch to the Visual Basic Editor window in Chap27.accdb and choose **Tools | References**.
2. In the References window, select **Microsoft XML, v6.0** (see Figure 27.22) and click **OK**.

If you don't have version 6.0 installed, select the lower version of this object type library, or upgrade your browser to the higher version so that the most recent library is available.



**FIGURE 27.22.** To work with XML documents programmatically, you need to reference the Microsoft XML object type library.

3. With the reference set, open the Object Browser (press F2) and examine XML DOM's objects, methods, and properties (see Figure 27.23).



**FIGURE 27.23.** To view objects, properties, and methods exposed by the XML DOM, open the Object Browser after setting up a reference to the Microsoft XML object type library (see Figure 27.22).

4. Close the Object Browser window.

The `DOMDocument` object is the top level of the XML DOM object hierarchy. This object represents a tree structure composed of nodes. You can navigate through this tree structure and manipulate the data contained in the nodes by using various methods and properties. The following sections demonstrate how to read and manipulate XML documents by using VBA procedures.

### **Loading and Retrieving the Contents of an XML File**

Hands-On 27.10 shows how to open an XML data file and retrieve both the raw data and the actual text stored in nodes.



#### **Hands-On 27.10 Loading and Retrieving the Contents of an XML File**

1. In the Visual Basic Editor window of the `Chap27.accdb` database, choose **Insert | Module** to add a new standard module to the current VBA project.

2. In the module's Code window, enter the following **ReadXMLDoc** procedure:

**NOTE**

*For this procedure to work correctly, you must set up the reference to the Microsoft XML object type library as instructed at the beginning of this section.*

```
Sub ReadXMLDoc()
    Dim xmldoc As MSXML2.DOMDocument60
    Dim strPath As String

    strPath = "C:\VBAAccess2021_XML\
    Set xmldoc = New MSXML2.DOMDocument60

    xmldoc.Async = False
    If xmldoc.Load(strPath & "InternalContacts.xml") Then
        Debug.Print xmldoc.XML
        ' Debug.Print xmldoc.Text
    End If
End Sub
```

To work with an XML document, we begin by creating an instance of the `DOMDocument` object as follows:

```
Dim xmldoc As MSXML2.DOMDocument60
Set xmldoc = New MSXML2.DOMDocument60
```

MSXML uses an asynchronous loading mechanism by default for working with documents. Asynchronous loading allows you to perform other tasks during long database operations, such as providing feedback to the user as MSXML parses the XML file or giving the user the chance to cancel the operation. Before calling the `Load` method, however, it's a good idea to set the `Async` property of the `DOMDocument` object to `False` to ensure that the XML file is fully loaded before other statements are executed. The `Load` method returns `True` if it successfully loaded the data and `False` otherwise. Having loaded the XML data into a `DOMDocument` object, you can use the `XML` property to retrieve the raw data or use the `Text` property to obtain the text stored in document nodes.

3. Position the insertion point anywhere within the code of the `ReadXMLDoc` procedure and choose **Run | Run Sub/UserForm**. The procedure executes and writes the contents of the XML file into the Immediate window as shown in Figure 27.24.
4. In the code of the `ReadXMLDoc` procedure, comment the first `Debug.Print` statement and uncomment the second statement that reads `Debug.Print xmldoc.Text`.

5. Run the ReadXMLDoc procedure again. This time the Immediate window should show the entry as one long line of text.



The screenshot shows the Microsoft Visual Studio Immediate window. The title bar says "Immediate". The window contains the raw XML data from the file "Employees.xml". The XML structure includes root elements like <Employees> and <Employee>, and nested elements such as <EmployeeID>, <LastName>, <FirstName>, <Title>, <TitleOfCourtesy>, <BirthDate>, <HireDate>, <Address>, <City>, <Region>, <PostalCode>, <Country>, <HomePhone>, <Extension>, <Photo>, <Notes>, and <ReportsTo>. The data is presented as a single continuous string of XML code.

```
<?xml version="1.0"?>
<dataroot xmlns:od="urn:schemas-microsoft-com:officedata" generated="2021-11-02T23:
<Employees>
    <EmployeeID>1</EmployeeID>
    <LastName>Davolio</LastName>
    <FirstName>Nancy</FirstName>
    <Title>Sales Representative</Title>
    <TitleOfCourtesy>Ms.</TitleOfCourtesy>
    <BirthDate>1968-12-08T00:00:00</BirthDate>
    <HireDate>1992-05-01T00:00:00</HireDate>
    <Address>507 - 20th Ave. E.
    Apt. 2A</Address>
    <City>Seattle</City>
    <Region>WA</Region>
    <PostalCode>98122</PostalCode>
    <Country>USA</Country>
    <HomePhone>(206) 555-9857</HomePhone>
    <Extension>5467</Extension>
    <Photo>EmpID1.bmp</Photo>
    <Notes>Education includes a BA in psychology from Colorado State University
    <ReportsTo>2</ReportsTo>
</Employees>
<Employees>
    <EmployeeID>2</EmployeeID>
    <LastName>Fuller</LastName>
    <FirstName>Andrew</FirstName>
    <Title>Vice President, Sales</Title>
    <TitleOfCourtesy>Dr.</TitleOfCourtesy>
    <BirthDate>1952-02-19T00:00:00</BirthDate>
```

FIGURE 27.24 By using the `XML` property of the `DOMDocument` object you can retrieve the raw data from an XML file.

## Working with XML Document Nodes

An XML document can contain nodes of different types. It can have a node that provides access to the entire XML document or one or more element nodes representing individual elements. Some nodes represent comments and processing instructions and others hold the text content of a tag. To determine the type of node, use the `nodeType` property of the `IXMLDOMNode` object. Node types are identified by either a text string or a constant.

For example, the node representing an element can be referred to as `NODE_ELEMENT` or 1, while the node representing the comment is named `NODE_COMMENT` or 8. See the MSXML2 Library in the Object Browser for the names of other node types.

In addition to node types, nodes can have parent, child, and sibling nodes. The `hasChildNodes` method lets you determine if a `DOMDocument` object has child nodes. There's also a `childNodes` property, which simplifies retrieving a collection of child nodes. Before you start looping through the collection of child nodes, it's a good idea to use the `length` property of the `IXMLDOMNode` ob-

ject to determine how many elements the collection contains. The following hands-on uses the InternalContacts.xml file to demonstrate how to work with XML document nodes.



### Hands-On 27.11 Working with XML Document Nodes

1. In the same module where you entered the ReadXMLDoc procedure in the previous hands-on, enter the following **LearnAboutNodes** procedure:

```
Sub LearnAboutNodes()
    Dim xmldoc As MSXML2.DOMDocument60
    Dim xmlNode As MSXML2.IXMLDOMNode
    Dim strPath As String

    strPath = "C:\VBAAccess2021_XML\
    Set xmldoc = New MSXML2.DOMDocument60
    xmldoc.Async = False

    xmldoc.Load (strPath & "InternalContacts.xml")
    If xmldoc.hasChildNodes Then
        Debug.Print "Number of child Nodes: " & _
            xmldoc.childNodes.length
        For Each xmlNode In xmldoc.childNodes
            Debug.Print "Node name:" & xmlNode.nodeName
            Debug.Print vbTab & "Type:" & _
                xmlNode.nodeTypeString _
                & "(" & xmlNode.nodeType & ")"
            Debug.Print vbTab & "Text: " & xmlNode.Text
        Next xmlNode
    End If
    Set xmldoc = Nothing
End Sub
```

Notice that this procedure uses the `hasChildNodes` property of the `DOMDocument` object to check whether there are any child nodes in the loaded XML file. If child nodes are found, the `length` property of the `childNodes` collection returns the total number of child nodes found. Next, the procedure loops through the `childNodes` collection and retrieves the node name using the `nodeName` property of the `IXMLDOMNode` object.

The `nodeTypeString` property returns the string version of the node type (for example, processing instruction, element, text, etc.) and the `nodeType` property is used to return the enumeration value. Finally, the `Text` property of the `IXMLDOMNode` object retrieves the node text.

2. Position the insertion point anywhere within the code of the LearnAboutNodes procedure and choose **Run | Run Sub/UserForm**. Running the LearnAboutNodes procedure produces the output like this:

```
Number of child Nodes: 2
Node name:xml
    Type:processinginstruction(7)
    Text: version="1.0" encoding="UTF-8"
Node name:dataroot
    Type:element(1)
        Text: 1 Davolio Nancy Sales Representative Ms. 1968-12-
08T00:00:00 1992-05-01T00:00:00 507 - 20th Ave. E.
... (and so on)
```

### Retrieving Information from Element Nodes

---

Let's assume that you want to read the information from only the text element nodes. Use the `getElementsByTagName` method of the `DOMDocument` object to retrieve an `IXMLDOMNodeList` object containing all the element nodes. This method takes one argument specifying the tag name to search for. To search for all the element nodes, use "\*" as the tag to search for.

The following hands-on exercise demonstrates how to obtain data from XML document element nodes.



### Hands-On 27.12 Retrieving Information from Element Nodes

1. In the Visual Basic Editor window, enter the following **IterateThruElements** procedure below the last procedure code you entered in Hands-On 27.11:

```
Sub IterateThruElements()
Dim xmldoc As MSXML2.DOMDocument60
Dim xmlNode As MSXML2.IXMLDOMNode
Dim xmlNodeList As MSXML2.IXMLDOMNodeList
Dim myNode As MSXML2.IXMLDOMNode
Dim strPath As String

strPath = "C:\VBAAccess2021_XML\"

Set xmldoc = New MSXML2.DOMDocument60
xmldoc.Async = False
xmldoc.Load (strPath & "InternalContacts.xml")
Set xmlNodeList = xmldoc.getElementsByTagName("*")
For Each xmlNode In xmlNodeList
    For Each myNode In xmlNode.childNodes
```

```
If myNode.nodeType = NODE_TEXT Then
    Debug.Print xmlNode.nodeName & _
    "=" & xmlNode.Text
End If
Next myNode
Next xmlNode
Set xmldoc = Nothing
End Sub
```

The IterateThruElements procedure retrieves the XML document name and the corresponding text for all the text elements in the InternalContacts.xml file. Notice that this procedure uses two `For Each...Next` loops. The first one (the outer loop) iterates through the entire collection of element nodes. The second one (the inner loop) uses the `nodeType` property to find only those element nodes that contain a single text node.

2. Position the insertion point anywhere within the code of the `IterateThruElements` procedure and choose **Run | Run Sub/UserForm**. Running the `IterateThruElements` procedure produces the following results:

```
EmployeeID=1
LastName=Davolio
FirstName=Nancy
Title=Sales Representative
TitleOfCourtesy=Ms.
BirthDate=1968-12-08T00:00:00
HireDate=1992-05-01T00:00:00
Address=507 - 20th Ave. E.
Apt. 2A
City=Seattle
Region=WA
PostalCode=98122
Country=USA
HomePhone=(206) 555-9857
Extension=5467
Photo=EmpID1.bmp
Notes=Education includes a BA in psychology from Colorado State University. She also completed "The Art of the Cold Call." Nancy is a member of Toastmasters International.
ReportsTo=2
EmployeeID=2
LastName=Fuller
FirstName=Andrew
...(and so on)
```

## Retrieving Specific Information from Element Nodes

You can list all the nodes that match a specified criterion by using the `selectNodes` method. The following hands-on exercise prints to the Immediate window the text for all Title nodes that exist in the InternalContacts.xml file. The `//Title` criterion of the `selectNodes` method looks for the element named Title at any level within the tree structure of the nodes.

### Hands-On 27.13 Retrieving Specific Information from Element Nodes

1. In the Visual Basic Editor Code window, in the same module where you entered previous procedures, enter the following **SelectNodesByCriteria** procedure:

```
Sub SelectNodesByCriteria()
    Dim xmldoc As MSXML2.DOMDocument60
    Dim xmlNodeList As MSXML2.IXMLDOMNodeList
    Dim myNode As MSXML2.IXMLDOMNode
    Dim strPath As String

    strPath = "C:\VBAAccess2021_XML\
    Set xmldoc = New MSXML2.DOMDocument60
    xmldoc.async = False
    xmldoc.Load (strPath & "InternalContacts.xml")
    Set xmlNodeList = xmldoc.selectNodes("//Title")
    If Not (xmlNodeList Is Nothing) Then
        For Each myNode In xmlNodeList
            Debug.Print myNode.Text
            If myNode.Text = "Sales Representative" Then
                myNode.Text = "Representative"
                xmldoc.Save strPath & "InternalContacts.xml"
            End If
        Next myNode
    End If
    Set xmldoc = Nothing
End Sub
```

The `SelectNodesByCriteria` procedure creates the `IXMLDOMNodeList` object that represents a collection of child nodes. The `selectNodes` method applies the specified pattern to this node's context and returns the list of matching nodes as `IXMLDOMNodeList`. The expression used by the `selectNodes` method specifies that all the Title element nodes should be included in the node list.

---

You can use the `Is Nothing` conditional expression to find out whether a matching element was found in the loaded XML file. If the matching elements were found in the `IXMLDOMNodeList`, the procedure iterates through the node list and prints each element node text to the Immediate window. In addition, if the node element's text value is Sales Representative, the procedure replaces this value with Representative. The `Save` method of the `DOMDocument` is used to save the changes in the `InternalContacts.xml` file.

---

2. Position the insertion point anywhere within the code of the `SelectNodesByCriteria` procedure and choose **Run | Run Sub/UserForm**. Running the `SelectNodesByCriteria` procedure produces the following results:

```
Sales Representative  
Vice President, Sales  
Sales Representative  
Sales Representative  
Sales Manager  
Sales Representative  
Sales Representative  
Inside Sales Coordinator  
Sales Representative
```

<b>NOTE</b>	<p><i>When you run this procedure again, you should see the following output:</i></p> <pre>Representative Vice President, Sales Representative Representative Sales Manager Representative Representative Inside Sales Coordinator Representative</pre>
-------------	---

### Retrieving the First Matching Node

If all you want to do is retrieve the first node that meets the specified criterion, use the `SelectSingleNode` method of the `DOMDocument` object. For this method's argument specify the string representing the node you'd like to find. For example, the following procedure finds the first node that matches the criterion `//City` in the `InternalContacts.xml` file:

```
Sub SelectSingleNode ()
```

```
Dim xmldoc As MSXML2.DOMDocument60
Dim xmlSingleNode As MSXML2.IXMLDOMNode
Dim strPath As String

strPath = "C:\VBAAccess2021_XML\
Set xmldoc = New MSXML2.DOMDocument60
xmldoc.async = False
xmldoc.Load (strPath & "InternalContacts.xml")
Set xmlSingleNode =
    xmldoc.SelectSingleNode("//City")
If xmlSingleNode Is Nothing Then
    Debug.Print "No nodes selected."
Else
    Debug.Print xmlSingleNode.Text
End If
Set xmldoc = Nothing
End Sub
```

The XML DOM provides a number of other methods that make it possible to programmatically add or delete elements in the XML document tree structure. Covering all of the details of the XML DOM Object Model is beyond the scope of this chapter. When you are ready for more information on this subject, visit: <http://www.w3.org/DOM/>

## USING ACTIVEX DATA OBJECTS WITH XML

---

Earlier in this book, you learned how to save ADO Recordsets to disk using the Advanced Data TableGram (adPersistADTG) format. This section expands on what you already know about ADO Recordsets by showing you how to use the ADO Recordset's adPersistXML constant to save all types of recordsets to disk as XML.

### Saving an ADO Recordset as XML to Disk

---

To save an ADO Recordset to a disk file as XML, use the `Save` method of the Recordset object with the `adPersistXML` constant. Hands-On 27.14 demonstrates how to create an XML file from ADO.



#### Hands-On 27.14 Creating an XML Document from ADO

1. In the Visual Basic Editor window of the Chap27.accdb database, choose **Insert | Module** to add a new standard module to the current VBA project.

2. Choose **Tools | References** to open the References dialog box. Check the box next to **Microsoft ActiveX Object Library 6.1** (or a lower version, if this one is not available) and click **OK**.
3. In the module's Code window, enter the following **SaveRst\_ToXMLwithADO** procedure:

```
Sub SaveRst_ToXMLwithADO()
    Dim conn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strPath As String
    Dim strDBName As String
    Dim strConn As String
    Dim strSQL As String

    strPath = "C:\VBAAccess2021_XML\
    strDBName = "Northwind 2007.accdb"
    strConn = "Provider = Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & strPath & strDBName & ";"

    strSQL = "SELECT "
    strSQL = strSQL & "Nz([Supplier IDs], "" "") as SupplierIDs,"
    strSQL = strSQL & "Nz([ID], "" "") as ID, Nz([Product Code],"" "")"
    as ProductCode, "
    strSQL = strSQL & "Nz([Product Name], "" "") as ProductName, "
    strSQL = strSQL & "Nz([Standard Cost], "" "") as StandardCost, "
    strSQL = strSQL & "Nz([List Price], "" "") as ListPrice, "
    strSQL = strSQL & "Nz([Reorder Level], "" "") as ReorderLevel, "
    strSQL = strSQL & "Nz([Target Level], "" "") as TargetLevel, "
    strSQL = strSQL & "Nz([Quantity Per Unit], "" "") as QuantityPerUnit, "
    strSQL = strSQL & "Nz([Discontinued], "" "") as Discontinued, "
    strSQL = strSQL & "Nz([Minimum Reorder Quantity],"" "")"
    as MinReorderQuantity, "
    strSQL = strSQL & "Nz([Category], "" "") as Category "
    strSQL = strSQL & "FROM Products;"

    Debug.Print strSQL
    ' open a connection to the database
    ' execute an SQL SELECT statement
    ' against the database

    Set conn = New ADODB.Connection
    With conn
        .Open strConn

        Set rst = .Execute(strSQL)
```

```
End With

' delete the file if it exists
On Error Resume Next

Kill strPath & "Products_AttribCentric.xml"

' save the recordset as an XML file
rst.Save strPath & "Products_AttribCentric.xml", _
adPersistXML

Set rst = Nothing
Set conn = Nothing

End Sub
```

This procedure begins by defining the necessary variables and setting their values. After that, a connection to the Northwind 2007.accdb database is established using the ADO Connection object and its Microsoft.ACE.OLEDB.12.0 provider. Next, the procedure executes an SQL SELECT statement against the database to retrieve data from the specified fields in the Products table. To avoid problems later with rendering of an HTML page, we use the Nz function on each table field to ensure that if there is no value in that field, we get a blank space.

```
strSQL = strSQL & "Nz([Product Name], "" "") as ProductName, "
```

Notice that the blank space is surrounded by double quotation marks. The prior code says “take what is already in the variable strSQL and add to it the data from the [Product Name] field, replacing it with a blank space if there is no data and return the result to the strSQL variable using ProductName (with no spaces) as the new field name.” When building a complex select statement in your code, make sure to output it with the Debug.Print statement to the Immediate window so you can verify that it is correct.

Once the records are placed in a recordset,

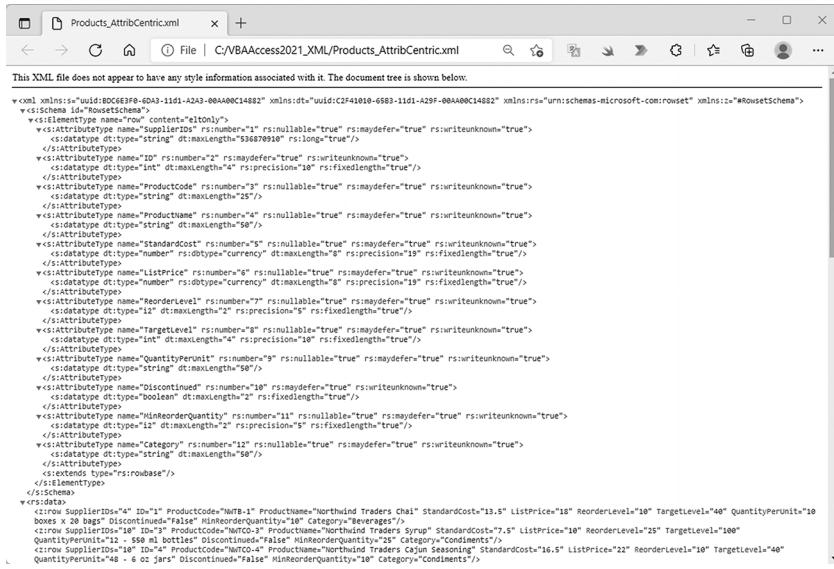
```
Set rst = .Execute(strSQL)
```

the Save method is called to store the recordset to a disk file using the adPersistXML format:

```
rst.Save strPath & "Products_AttribCentric.xml", adPersistXML
```

If the disk file already exists, the procedure deletes the existing file using the VBA Kill statement. The On Error Resume Next statement bypasses the Kill statement if the file you are going to create does not yet exist.

4. Position the insertion point anywhere within the code of the procedure and choose **Run | Run Sub/UserForm**.
  5. Open the **C:\VBAAccess2021\_XML\Products\_AttribCentric.xml** file created by the SaveRst\_ToXMLwithADO procedure and examine its content. The browser displays the raw XML as shown in Figure 27.25. Notice that the content of this file looks different from other XML files you generated in this chapter. The reason for this is that XML that is persisted from ADO Recordsets is created in a so-called attribute-centric XML. Access supports only element-centric XML. Therefore, to import to Access an XML file created from ADO, you must first create and apply an XSLT transformation to the source document. The stylesheet you create should convert the attribute-centric XML to element-centric XML that Access can handle (see Hands-On 27.15).



**FIGURE 27.25.** Saving a recordset to an XML file with ADO produces an attribute-centric XML file.

## Attribute-Centric and Element-Centric XML

In the XML file generated in Hands-On 27.14 and shown in Figure 27.25, you can see two child nodes: `<s:Schema>` and `<rs:data>`.

The schema node describes the structure of the recordset, while the data node holds the actual data. Inside the `<s:Schema id="RowsetSchema">` and `</s:Schema>` tags, ADO places information about each column: field name, position, data type and length, nullability, and whether the column is writable. Each field is represented by the `<s:AttributeType>` element. Notice that the value of the name attribute is the field name. The `<s:AttributeType>` element also has a child element, `<s:datatype>`, which holds information about its data type (integer, number, string, etc.) and the maximum field length.

Below the schema definition is the actual data. The ADO schema represents each record using the `<z:row>` tag. The fields in a record are expressed as attributes of the `<z:row>` element. Every XML attribute is assigned a value that is enclosed in a pair of single or double quotation marks; however, if the value of a field in a record is Null, the attribute on the `<z:row>` is not created. Notice that each record is written out in the following format:

```
<z:row SupplierIDs="4" ID="1" ProductCode="NWTB-1"
ProductName="Northwind Traders Chai" StandardCost="13.5" List-
Price="18" ReorderLevel="10" TargetLevel="40" QuantityPer-
Unit="10 boxes x 20 bags" Discontinued="False" MinReorderQuan-
tity="10" Category="Beverages"/>
```

The preceding code fragment is referred to as attribute-centric XML and Access will not be able to import it. To make the XML file compatible with Access, each record must be written out as follows:

```
<Product>
<SupplierIDs>4</SupplierIDs>
<ID>1</ID>
<ProductCode>NWTB-1</ProductCode>
<ProductName>Northwind Traders Chai</ProductName>
<StandardCost>13.5</StandardCost>
<ListPrice>18</ListPrice>
<ReorderLevel>10</ReorderLevel>
<TargetLevel>40</TargetLevel>
<QuantityPerUnit>10 boxes x 20 bags</QuantityPerUnit>
<Discontinued>False</Discontinued>
<MinReorderQuantity>10</MinReorderQuantity>
<Category>Beverages</Category>
</Product>
```

The code fragment shown above represents element-centric XML. Notice that each record is wrapped in a `<Product>` tag, and each field is an element under the `<Product>` tag.

## Changing the Type of an XML File

Because it is much easier to work with element-centric XML files (and Access does not support attribute-centric XML), you must write an XSL stylesheet to transform an attribute-centric XML file to an element-centric XML file before you can import to Access an XML file created from an ADO recordset.

The following hands-on exercise demonstrates how to write a stylesheet to perform a conversion.



### Hands-On 27.15 Creating a Stylesheet to Convert Attribute-Centric XML to Element-Centric XML

1. Open Notepad and type the following stylesheet code:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:rs="urn:schemas-microsoft-com:rowset">
<xsl:output method="xml" encoding="UTF-8" />

<xsl:template match="/">
    <!-- root element for the XML output -->
    <Products xmlns:z="#RowsetSchema">

        <xsl:for-each select="/xml/rs:data/z:row">
            <Product>
                <xsl:for-each select="@*"/>
                <xsl:element name="{name()}">
                    <xsl:value-of select="."/>
                </xsl:element>
            </xsl:for-each>
            </Product>
        </xsl:for-each>
    </Products>

</xsl:template>
</xsl:stylesheet>
```

2. Save this stylesheet as **C:\VBAAccess2021\_XML\AttribToElem.xsl**. Be sure to include the .xsl extension so the file is not saved as text. You will use this stylesheet for the transformation in the next hands-on exercise.

Notice in the preceding stylesheet that the “@\*” wildcard matches all attribute nodes. Each time the `<z:row>` tag is encountered, an element named `<Product>` will be created. And for each attribute, the attribute name will be converted to

the element name using the built-in XPath `name()` function. Expressions in curly braces are evaluated and converted to strings. The `select=". "` returns the current value of the attribute being read.

See the next section on how to apply this stylesheet to the XML document.

### Applying an XSL Stylesheet

Now that you've created the stylesheet to transform an attribute-centric XML file into an element-centric file, you can use the `transformNodeToObject` method of the `DOMDocument` object to apply the stylesheet to the `Products_AttribCentric.xml` file created in Hands-On 27.14. The hands-on exercise that follows demonstrates how to do this. In addition, the procedure in this exercise will import the converted ADO XML file to Access.



#### **Hands-On 27.16 Applying a Stylesheet to an ADO XML Document and Importing It to Access**

1. Enter the following procedure below the procedure code you created in Hands-On 27.14:

```
Sub ApplyStyleSheetAndImport()
    Dim myXMLDoc As New MSXML2.DOMDocument60
    Dim myXSLDoc As New MSXML2.DOMDocument60
    Dim newXMLDoc As New MSXML2.DOMDocument60
    Dim strXMLFile As String
    Dim strPath As String

    strPath = "C:\VBAAccess2021_XML\"

    strXMLFile = "Products_AttribCentric.xml"
    myXMLDoc.Async = False
    If myXMLDoc.Load(strPath & strXMLFile) Then
        myXSLDoc.Load strPath & "AttribToElem.xsl"

        ' apply the transformation
        If Not myXSLDoc Is Nothing Then
            myXMLDoc.transformNodeToObject _
                myXSLDoc, newXMLDoc

            ' save the output in a new file
            newXMLDoc.Save strPath & _
                "Products_Converted.xml"

            ' import to Access
            Application.ImportXML _
                strPath & "Products_Converted.xml"
```

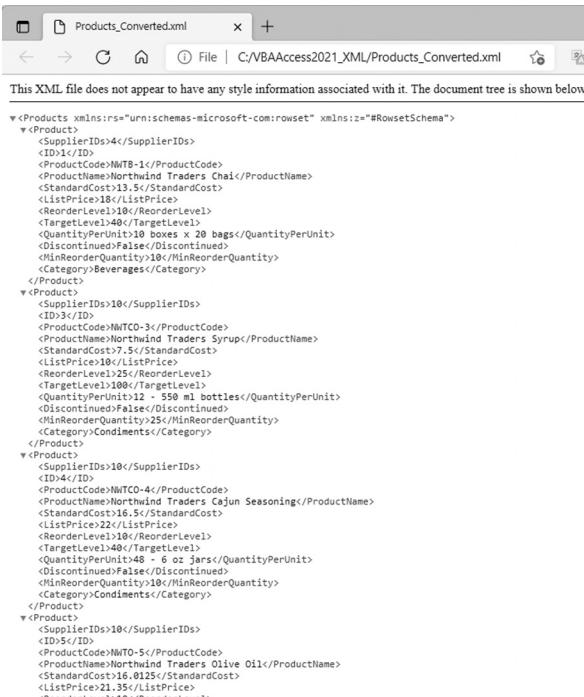
```

    End If
End If
End Sub

```

This procedure begins by loading both the Products\_AttribCentric.xml file (created in Hands-On 27.14) and the AttribToElem.xsl stylesheet (created in Hands-On 27.15) into the `DOMDocument` object. Next, the stylesheet is applied to the source file by using the `transformNodeToObject` method. This method is applied to a node in the source XML document's tree and takes two arguments. The first argument is a stylesheet in the form of a `DOMDocument` node. The second argument is another `DOMDocument` node that will hold the result of the transformation. Next, the result of the transformation is saved to a file (Products\_Converted.xml) and the file is imported to Access using the `ImportXML` method, which was introduced earlier in this chapter.

2. Run the `ApplyStyleSheetAndImport` procedure.
3. Open the `C:\VBAAccess2021_XML\Products_Converted.xml` file. Notice that the `Products_Converted.xml` file content is now element-centric XML (see Figure 27.26).



**FIGURE 27.26.** This element-centric XML file is a result of applying a stylesheet to the attribute-centric ADO Recordset that was saved to an XML file.

**4. In the Access window, locate and open the table named **Product**.**

The Product table shown in Figure 27.27 was created by the `ImportXML` method in the `ApplyStyleSheetAndImport` procedure.

SupplierIDs	ID	ProductCode	ProductName	StandardCost	ListPrice	ReorderLevel	TargetLevel	QuantityPerUnit	Discontinued	MinReorderQuantity	Category	UnitPrice
10	19	NWITD-3	Northwind Traders Syrup	7.5	18	10	25	100	1	10	Condiments	18.00
10	4	NWITD-4	Northwind Traders Cajun Seasoning	16.5	22	10	40	48	0	20	Condiments	12 - 250 ml bottles
10	5	NWITD-5	Northwind Traders Olive Oil	16.025	21.35	10	40	36 boxes	0	10	Condiments	48 - 6 oz jars
2	7	NWITD-7	Northwind Traders Boysenberry Spread	18.75	25	100	25	100	0	10	Jams, Preserves	12 - 8 oz jars
8	8	NWITD-8	Northwind Traders Curry Pears	30	30	10	40	12	0	10	Dried Fruit & Nuts	12 - 12 oz jars
2,6	18	NWITD-14	Northwind Traders Walnuts	17.4375	23.25	10	40	40 - 100 g pkgs	0	10	Sauces	15.25 OZ
6	17	NWITD-17	Northwind Traders Fruit Cocktail	6.9	9.9	10	20	10 boxes x 12 fl	0	5	Baked Goods & Mixes	5
1	19	NWITD-19	Northwind Traders Chocolate Biscuits Mix	6.9	9.2	5	20	10 boxes x 12 fl	0	10	Jams, Preserves	10
2,6	20	NWITD-20	Northwind Traders Apple Pie	7.75	11	10	20	30	0	10	Baked Goods & Mixes	10
1	21	NWITD-21	Northwind Traders Scones	7.5	10	5	20	24 pkgs x 4 pkg	0	5	Beverages	5
4	31	NWITD-34	Northwind Traders Beer	10.5	14	15	60	24 - 12 oz bottle	0	15	Canned Meat	15
7	40	NWITD-40	Northwind Traders Crab Meat	13.8	18.4	30	120	24 - 4 oz tins	0	30	Canned Meat	30
6	41	NWITD-41	Northwind Traders Tomato Chowder	7.375	9.65	10	40	12	0	10	Soups	10
3,4	43	NWITD-43	Northwind Traders Coffee	34.5	46	25	100	16 - 500 g tins	0	25	Beverages	10
10	48	NWITD-48	Northwind Traders Chocolate	9.5625	12.75	25	100	10 pkgs	0	25	Candy	12.75 OZ
2	51	NWITD-51	Northwind Traders Dried Apples	39.75	53	10	40	50 - 300 g pkgs	0	10	Dried Fruit & Nuts	50 - 300 g pkgs
1	52	NWITD-52	Northwind Traders Long Grain Rice	3.25	7	25	100	16 - 2 kg bags	0	25	Pasta	16 - 2 kg bags
1	57	NWITD-57	Northwind Traders Ravioli	28.5	38	30	120	24 - 250 g pkgs	0	30	Pasta	24 - 250 g pkgs
8	63	NWITD-63	Northwind Traders Hot Pepper Sauce	14.625	19.5	20	80	24 - 250 g pkgs	0	20	Sauces	20
8	66	NWITD-66	Northwind Traders Tomato Sauce	12.75	17	20	80	40	0	10	Sauces	10
5	72	NWITD-72	Northwind Traders Chocolates	26.1	34.8	10	40	24 - 8 oz bottles	0	10	Dairy Products	24 - 8 oz bottles
2,6	74	NWITD-74	Northwind Traders Almonds	7.5	10	5	20	5 kg pkg	0	5	Dried Fruit & Nuts	5 kg pkg
10	77	NWITD-77	Northwind Traders Mustard	9.75	13	15	60	12 boxes	0	15	Condiments	13 OZ
2	80	NWITD-80	Northwind Traders Dried Plums	3	3.5	50	75	1 lb bag	0	25	Dried Fruit & Nuts	1 lb bag
3	81	NWITD-81	Northwind Traders Gunpowder Tea	2	2.99	100	125	20 bags per box	0	25	Beverages	20 bags per box
1	82	NWITD-82	Northwind Traders Granola	2	4	20	100	False	0	10	Cereal	10
9	83	NWITD-83	Northwind Traders Potato Chips	0.5	1.8	30	200	False	0	10	Chips, Snacks	10

**FIGURE 27.27.** This table was imported to Access after conversion of attribute-centric ADO recordset into the element-centric XML file.

## Transforming Attribute-Centric XML Data into an HTML Table

As you've seen in earlier examples, creating an XML file from an ADO Recordset results in output that contains attribute-centric XML. To import this type of output to Access you had to create a special stylesheet and apply the transformation to convert the attribute-centric XML to the element-centric XML that Access supports. But what if you simply want to display the XML file created from an ADO Recordset in a Web browser? To do this you must create a generic XSL stylesheet that draws a simple HTML table for the users when they open the XML attribute-centric file in their browser.

Hands-On 27.17 demonstrates how to create a stylesheet to transform the attribute-centric XML file that we created in Hands-On 27.14 into HTML. Hands-On 27.18 performs the transformation by inserting a reference to the XSL stylesheet into the XML document.

### Hands-On 27.17 Creating a Generic Stylesheet to Transform an Attribute-Centric XML File into HTML

1. Open Notepad and type the following stylesheet code:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0">
```

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:s='uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882'
xmlns:dt='uuid:C2F41010-65B3-11d1-A29F-00AA00C14882'
xmlns:rs='urn:schemas-microsoft-com:rowset'
xmlns:z='#RowsetSchema'
xmlns:html="http://www.w3.org/TR/REC-html40">

<xsl:template match="/">

<html>
<head>
<title>XML to HTML</title>

<style type="text/css">

table {
    font-family: arial, sans-serif;
    font-size:9px;
    border-collapse: collapse;
    width: 100%;
}

td, th {
    border: 1px solid #dddddd;
    text-align: left;
    padding: 8px;
}

th {background-color:#9acd32; color:black}

tr:nth-child(even) {
    background-color: #dddddd;
}
</style>

</head>
<body>
<table width="100%" border="1">

<!-- generate table headings -->
<xsl:for-each
select="xml/s:s:Schema/s:s:ElementType/s:s:AttributeType">
<th>
<xsl:value-of select="@name" />
</th>
</xsl:for-each>
```

```
<!-- loop through all data rows and get values for each column
-->
<xsl:for-each select="xml/rs:data/z:row">
<tr>
<xsl:for-each select="@*">
<td>

<xsl:value-of select=". "/>

</td>
</xsl:for-each>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

2. Save the stylesheet as **AttribToHTML.xsl**. Be sure to include the .xsl extension so the file is not saved as text.
3. Close Notepad.

The stylesheet uses the feature known as Cascading Stylesheets (CSS) to format the HTML table. A style comprises different properties—bold, italic, font size and font weight, color, etc.—that you want to apply to text (titles, headers, body, etc.). In this stylesheet, we applied simple styles to the following HTML table elements: table, th (table heading), td (table data), and tr (table row). Using styles is very convenient. If you don't like the formatting, you can simply change the style definition and get a new look instantly. Each style definition is contained between curly braces { }.

```
td, th {
    border: 1px solid #dddddd;
    text-align: left;
    padding: 8px;
}
```

For more information about creating HTML styles and an opportunity to try it yourself, check out the following link: [https://www.w3schools.com/html/html\\_styles.asp](https://www.w3schools.com/html/html_styles.asp).

The example stylesheet uses template-based processing. The following instruction defines a template for the entire document:

```
<xsl:template match="/">
```

The code between the opening and closing tags will be processed for all tags whose names match the value of the match attribute. In other words, we want the pattern matching to be applied to the entire document (/).

Next, a loop is used to write out the table headings. To do this, you must move through all the AttributeType elements of the root element, outputting the name attribute's value like this:

```
<xsl:for-each  
select="xml/s:Schema/s:ElementType/s:AttributeType">  
<th>  
<xsl:value-of select="@name" />  
</th>  
</xsl:for-each>
```

An attribute's name is always preceded by @.

Next, another loop runs through all the <z:row> elements representing actual records:

```
<xsl:for-each select="xml/rs:data/z:row">
```

All the attributes of any <z:row> element are enumerated:

```
<xsl:for-each select="xml/rs:data/z:row">  
<tr>  
<xsl:for-each select="@*>  
<td>  
<xsl:value-of select="." />  
</td>
```

The string "@\*" denotes any attribute. For each attribute found under the <z:row> element, you need to match the attribute name with its corresponding value. Notice the period in the <xsl:value-of> tag. The period represents the node that XSLT is currently working with. In summary, the preceding code fragment tells the XSLT processor to display the value of the current node during the iteration of the <z:row> attributes.

Now that you are finished with the stylesheet, the next step is to link the XML and XSL files. You can do this by adding a reference to a stylesheet in your XML document as shown in Hands-On 27.18.



### Hands-On 27.18 Linking the Attribute-Centric XML File with the Generic Stylesheet and Displaying the Transformed File in a Web Browser

1. Save the Products\_AttribCentric.xml file as Products\_AttribCentric\_2.xml.

2. Open the **Products\_AttribCentric\_2.xml** file with Notepad.
3. Type the following definition in the first line of this file:

```
<?xml-stylesheet type="text/xsl" href="AttribToHTML.xsl"?>
```

This instruction establishes a reference to the XSL file.

4. Save the changes made to the **Products\_AttribCentric\_2.xml** file and close Notepad.

Now is the time to check the result. Unfortunately, double-clicking the **Products\_AttribCentric\_2.xml** file produces a blank page. To view the data, you need to access it from the Web server.

5. Open your favorite browser and enter the following in the URL: *http://localhost/acc\_xml/Products\_AttribCentric\_2.xml*

You should see the data formatted in a table (see Figure 27.28).

SupplierID	ID	ProductCode	ProductName	StandardCost	ListPrice	ReorderLevel	TargetLevel	QuantityPerUnit	Discontinued	MinReorderQuantity	Category
1	1	NWTB-1	Northwind Traders Chai	13.5	18	10	40	10 boxes x 20 bags	0	10	Beverages
2	3	NWTCO-3	Northwind Traders Syrup	7.5	10	25	100	12 - 500 ml bottles	0	25	Condiments
3	4	NWTCO-4	Northwind Traders Cajun Seasoning	18.5	22	10	40	48 - 6 oz jars	0	10	Condiments
4	5	NWTOP-5	Northwind Traders Olive Oil	16.0125	21.35	10	40	36 boxes	0	10	Oil
5	6	NWTJP-6	Northwind Traders Boysenberry Spread	18.75	25	25	100	12 - 8 oz jars	0	25	Jams, Preserves
6	7	NWTDN-7	Northwind Traders Dried Pears	22.5	30	10	40	12 - 1 lb pkgs.	0	10	Dried Fruit & Nuts
7	8	NWTS-8	Northwind Traders Curry Sauce	30	40	10	40	12 - 12 oz jars	0	10	Sauces
8	14	NWTDPN-14	Northwind Traders Wanuts	17.4375	23.25	10	40	40 - 100 g pkgs.	0	10	Dried Fruit & Nuts
9	17	NWTCPV-17	Northwind Traders Fruit Cocktail	29.25	39	10	40	15.25 OZ	0	10	Canned Fruit & Vegetables
10	19	NWTBGM-19	Northwind Traders Biscuits Mix	6.9	9.2	5	20	10 boxes x 12 pieces	0	5	Baked Goods & Mixes
11	20	NWTUP-20	Northwind Traders Marmalade	60.75	91	10	40	20 gift boxes	0	10	Jams, Preserves
12	21	NWTBGM-21	Northwind Traders Scones	7.5	10	5	20	24 pkgs x 4 pieces	0	5	Baked Goods & Mixes
13	34	NWTB-34	Northwind Traders Beer	10.5	14	15	80	24 - 12 oz bottles	0	15	Beverages
14	40	NWTCM-40	Northwind Traders Crab Meat	13.8	18.4	30	120	24 - 4 oz pkgs	0	30	Canned Meat
15	41	NWTSO-41	Northwind Traders Clam Chowder	7.2375	9.65	10	40	12 - 12 oz cans	0	10	Soups
16	43	NWTB-43	Northwind Traders Coffee	34.5	46	25	100	16 - 500 g lins	0	25	Beverages
17	48	NWTCAG-48	Northwind Traders Chocolate	9.6625	12.75	25	100	10 pkgs	0	25	Candy
18	51	NWTDPN-51	Northwind Traders Dried Apples	39.75	53	10	40	50 - 300 g pkgs.	0	10	Dried Fruit & Nuts
19	55	NWTQ-52	Northwind Traders Long Grain Rice	5.25	7	25	100	10 - 2 kg boxes	0	25	Grains

**FIGURE 27.28.** You can apply a stylesheet to an XML document generated by the ADO to display the data in a nicely formatted HTML table.

## Loading an XML Document in Excel

After saving an ADO Recordset to an XML file on disk, you can load it into a desired application and read it as if it were a database. To gain access to the records saved in the XML file, use the `Open` method of the Recordset object and specify the filename, including its path and the persisted recordset service provider as `Provider=MSPersist`. The following hands-on exercise demonstrates how to programmatically open in Excel a persisted recordset that was saved in XML files (`Products_AttribCentric.xml` and `Products_AttribCentric_2.xml`).



### Hands-On 27.19 From Access to Excel: Loading an XML File into an Excel Workbook

1. In the Visual Basic Editor window, choose **Insert | Module** to add a new standard module to the current VBA project.
2. Choose **Tools | References** and click the checkbox next to the **Microsoft Excel 16.0 Object Library (or its earlier version)**. Click **OK** to exit the References dialog box.
3. In the module's Code window, enter the following **OpenAdoFile** procedure:

```
Sub OpenAdoFile()
    Dim rst As ADODB.Recordset
    Dim objExcel As Excel.Application
    Dim wkb As Excel.Workbook
    Dim wks As Excel.Worksheet
    Dim StartRange As Excel.Range
    Dim h As Integer
    Dim strPath As String

    strPath = "C:\VBAAccess2021_XML\" 
    Set rst = New ADODB.Recordset

    ' open your XML file and load it
    rst.Open strPath & "Products_AttribCentric.xml", _
    "Provider=MSPersist"

    ' display the number of records
    MsgBox "There are " & rst.RecordCount & _
    " records in this file."

    Set objExcel = New Excel.Application

    ' create a new Excel workbook
    Set wkb = objExcel.Workbooks.Add

    ' set a reference to the ActiveSheet
    Set wks = wkb.ActiveSheet

    ' make Excel application window visible
    objExcel.Visible = True

    ' copy field names as headings
    ' to the 1st row of the worksheet
    For h = 1 To rst.Fields.Count
        wks.Cells(1, h).Value = rst.Fields(h - 1).Name
    Next h
End Sub
```

Next

```
' specify the cell range to
' receive the data (A2)
Set StartRange = wks.Cells(2, 1)

' copy the records from the
' recordset beginning in cell A2
StartRange.CopyFromRecordset rst

' autofit the columns to make the data fit
wks.Range("A1").CurrentRegion.Select
wks.Columns.AutoFit

' save the workbook
wkb.SaveAs strPath & "ExcelReport.xls"

Set objExcel = Nothing
Set rst = Nothing
End Sub
```

This procedure is well commented, so we will skip its analysis and proceed to the next step.

#### 4. Run the OpenAdoFile procedure.

When the procedure is complete, the Excel application window should be visible with the ExcelReport.xls workbook file displaying products retrieved from the XML file (see Figure 27.29).

#### 5. Close the Excel workbook and exit Excel.

	A	B	C	D	E	F	G	H	I	J	K	L
	SupplierIDs	ID	ProductCode	ProductName	StandardCost	ListPrice	ReorderLevel	TargetLevel	QuantityPerUnit	Discontinued	MinReorderQuantity	Category
1	1	NWTRT-1	Northwind Traders Chai	13.5	18	10	40	10 boxes x 20 bags	0	10		Beverages
2	2	NWTRT-3	Northwind Traders Syrup	7.5	10	25	100	12 - 550 ml bottles	0	25		Condiments
3	3	NWTRT-4	Northwind Traders Cajun Seasoning	16.5	22	10	40	48 - 6 oz jars	0	10		Condiments
4	4	NWTRT-5	Northwind Traders Olive Oil	16.0125	21.35	10	40	36 boxes	0	10		Oil
5	5	NWTRT-6	Northwind Traders Boysenberry Spread	18.75	25	25	100	12 - 8 oz jars	0	25		Jams, Preserves
6	6	NWTRP-6	Northwind Traders Dried Pears	2.25	30	10	40	12 - 1 lb pkgs.	0	10		Dried Fruit & Nuts
7	7	NWTRP-7	Northwind Traders Curry Sauce	30	40	10	40	12 - 12 oz jars	0	10		Sauces
8	8	NWTRP-8	Northwind Traders Walnuts	17.4375	23.25	10	40	40 - 100 g pkgs.	0	10		Dried Fruit & Nuts
9	9	NWTRP-17	Northwind Traders Fruit Cocktail	29.25	39	10	40	15.25 OZ	0	10		Canned Fruit & Vegetables
10	10	NWTRGM-10	Northwind Traders Chocolate Biscuits Mix	6.9	9.2	5	20	10 boxes x 12 pieces	0	5		Baked Goods & Mixes
11	11	NWTRP-10	Northwind Traders Marmalade	60.75	81	10	40	30 gift boxes	0	10		Jams, Preserves
12	12	NWTRGM-21	Northwind Traders Scones	7.35	10	5	20	24 pkgs. x 4 pieces	0	5		Baked Goods & Mixes
13	13	NWTRP-21	Northwind Traders Beer	10.5	14	15	50	24 - 12 oz bottles	0	15		Beverages
14	14	NWTRCM-40	Northwind Traders Smoked Meat	13.8	18.4	10	120	24 - 2 oz tins	0	30		Canned Meat
15	15	NWTRD-41	Northwind Traders Clam Chowder	2.2375	6.65	10	40	12 - 12 oz cans	0	10		Soups
16	16	NWTRD-43	Northwind Traders Coffee	34.5	46	25	100	16 - 500 g tins	0	25		Beverages
17	17	NWTRCA-48	Northwind Traders Chocolate	9.5625	12.75	25	100	10 pkgs.	0	25		Candy
18	18	NWTRPN-51	Northwind Traders Dried Apples	39.75	63	10	40	50 - 300 g pkgs.	0	20		Dried Fruit & Nuts
19	19	NWTRG-52	Northwind Traders Long Grain Rice	5.25	7	25	100	16 - 2 kg boxes	0	25		Grains

FIGURE 27.29 An ADO Recordset persisted to an XML file is now opened in Excel.

## SUMMARY

---

This chapter has shown you what you can do with Access 2021 and XML. Using a combination of Access built-in commands and VBA programming code, you can export Access data to an XML file and import an XML file and display it as an Access table.

You learned what XML is and how it is structured. After working through the examples in this chapter, it's easy to see that XML supplies you with numerous ways to accomplish a specific task. Because XML is stored in plain text files, it can be read by many types of applications, independent of the operating system or hardware. You learned how to transform data from XML to HTML and from one XML format to another. You explored the ADO Recordset methods suitable for working with XML programmatically and were introduced to XSL stylesheets and XSLT transformations.

All of the methods and techniques you've studied here will take time to sink in. XML is not like VBA. It is not very independent; it needs many supporting technologies to assist it in its work. So don't despair if you don't understand something right away. Learning XML requires learning many other new concepts, like XSLT, XPath, and schemas. Take XML step by step by experimenting with it. The time that you invest in studying this technology will not be wasted. XML has been around for quite a while and is here to stay. The three main reasons why you should consider using XML are :

- XML separates content from presentation.

If you are planning to design Web pages, you do not need to make changes to your HTML files when the data changes. Because the data is kept in separate files, it's easy to make modifications.

- XML is perfect for sharing and exchanging data.

You no longer must worry about whether your data needs to be processed by a system that's not compatible with yours. Because all systems can work with text files (and XML documents are simply text files), you can share and exchange your data without a headache.

- XML can be used as a database.

You no longer need a database system to have a database.



# Chapter 28 ACCESS AND REST API

We've almost reached the end of this book. In this last chapter, we will focus on expanding your VBA skillset by covering topics such as working with a VBA Dictionary Object, using regular expressions, and calling a new type of a Web service, known as REST API.

## INTRODUCTION TO A VBA DICTIONARY OBJECT

---

After you've learned how arrays and collections can help you store values while your program is running, you may be surprised to learn that there is still another way to manipulate your data. An object known as VBA dictionary operates like a collection object, but offers more flexibility and speed than one. It works in a similar way to a normal dictionary, allowing you to look up values based on a key you provide. A dictionary is a collection of key-value pairs, and each key must be unique. Unlike an array, its size does not need to be predefined. Its data type is a Variant so you can enter any type of data you want to keep track of (text, numbers, dates, arrays, or any other objects).

The dictionary object is not part of standard VBA. It is a part of the Microsoft Scripting Runtime library (scrrun.dll).

## Accessing the VBA Dictionary

---

There are two ways in which you can access the VBA dictionary in your VBA code. One is referred to as *early binding* and the other is called *late binding*.

- Early Binding

To use early binding, you need to add a reference to the *Microsoft Scripting Runtime* library as described in the next section. With early binding, your code is compiled before it runs, so your procedures can run much faster. Once the reference is added to the required library, you can write the following code to define your dictionary object:

```
Dim myDict As New Scripting.Dictionary
```

With early binding, you can count on the *Intellisense* to help you with syntax and programming assistance.

- Late Binding

With late binding, your object will be compiled while your code runs, therefore it will be slower. You are not required to set up a library reference. To define the dictionary object, specify the `Scripting.Dictionary` using the `CreateObject` method:

```
Dim myDict As Object  
Set myDict = CreateObject("Scripting.Dictionary")
```

## Adding a Reference to the Microsoft Scripting Runtime Library

---

You add the reference to the Microsoft Scripting Runtime library in the same way you added references to other libraries that were introduced in this book. Simply select **Tools | References** in the Visual Basic Editor (VBE) screen and scroll down in the pop-up window until you locate the library. Select it and click OK. Figure 28.1 shows this selection. After you've added the library reference, it's a good idea to examine the content of this library by using the Object Browser. In the VBE screen, press **F2** or choose **View | Object Browser**. Choose **Scripting** in the first drop-down as shown in Figure 28.2. Notice the available types of objects, one of them being the `Dictionary`. By clicking on each `Dictionary` member, you can find out the type of the member (method, property) and its function. For example, the `Item` property is the default member of `Scripting.Dictionary` and is used to set or get the item for a given key, while `Add` is a method used to add a new key and item to the dictionary.

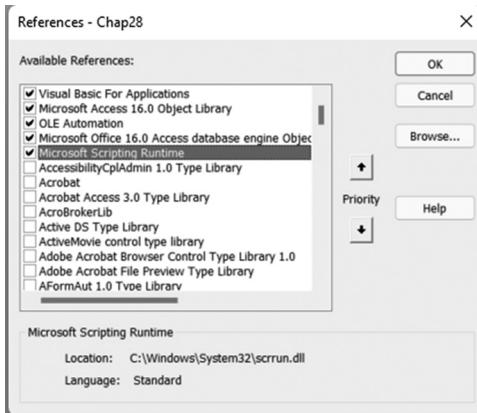


FIGURE 28.1 Adding a reference to the Microsoft Scripting Runtime library.



FIGURE 28.2 Examining the content of the Microsoft Scripting Runtime library.

## **Working with the Dictionary Object's Properties and Methods**

The Scripting Dictionary has the following methods and Properties:

<b>Methods</b>	
Add	Adds a new key/item pair to a Dictionary object.
Remove	Removes a specified key/item pair from the Dictionary object.
RemoveAll	Removes all the key/item pairs in the Dictionary object.
Items	Returns an array of all the items in a Dictionary object.
Keys	Returns an array of all the keys in a Dictionary object.
Exists	Returns a Boolean value (true/false) that indicates whether a specified key exists in a Dictionary object.
<b>Properties</b>	
Key	Sets a new key value for an existing key value in the Dictionary object.
Item	Sets or returns the value of an item in a Dictionary object.
Count	Returns the number of key/value pair in a Dictionary object.
CompareMode	Sets or returns the comparison mode for comparing keys in a Dictionary object.

Let's assume that you need to create a dictionary of world capital cities. Using the Add method of the Scripting.Dictionary, you can fill your dictionary like this:

```
Sub FillDictionary()
'declare a dictionary object
'uses early binding
'requires the reference to the Microsoft Scripting Runtime Library
Dim objDict As New Scripting.Dictionary

'add items to the dictionary
objDict("USA") = "Washington D.C."
objDict("Canada") = "Ottawa"
objDict("France") = "Paris"
objDict("England") = "London"
objDict("Hungary") = "Budapest"
objDict("Italy") = "Rome"
objDict("Japan") = "Tokyo"

objDict.Add "Germany", "Berlin"
objDict.Add key:="China", item:="Beijing"
```

In the code snippet above, you will notice that you can use three different ways to add an item to a dictionary. Use whatever method feels more comfortable to you.

After filling in the dictionary, you will want to read its keys / values. You can use the `Keys` method to return an array of all the keys like this:

```
'iterate through the dictionary to read its keys
Dim i As Long
For i = LBound(objDict.Keys) To UBound(objDict.Keys)
    Debug.Print i & "-->" & objDict.Keys(i)
Next i
```

In the prior code, we utilize the Upper and Lower Bound methods (`UBound`, `LBound`) of an array to list the keys.

The following code will retrieve a specific key:

```
'retrieve 3rd key
Debug.Print "3rd key=" & objDict.Keys()(2)
```

Recall that arrays are zero-based. You can place all keys into an array like this:

```
Dim aKeys() As Variant
aKeys = objDict.Keys
Debug.Print "Dictionary contains " & UBound(aKeys) + 1 & " keys."
Debug.Print "The first key is " & aKeys(0)
```

It is easy to remove a key from a dictionary, but before removal, always check if the key exists using the `Exists` method. This method is what makes the dictionary object easier to manipulate than collections and arrays. The keys are case-sensitive, so pay attention to the case when checking for the key existence.

```
'remove the key if it exists
If objDict.Exists("France") Then
    objDict.Remove "France"
Else
    Debug.Print "This key does not exist"
End If
```

The `Count` property returns the total items in dictionary and the `Item` property is used to set the value of a dictionary item. In the following example, we replace the value for “Germany.”

```
'count the items in the dictionary
Debug.Print "Now the dictionary contains " & objDict.Count & " keys."
objDict.item("Germany") = "Frankfurt"
Debug.Print objDict("Germany") & " is a city in Germany."
```

Using the `For Each` loop, you can list all the items or keys in the dictionary, like this:

```
' list all the items in the dictionary
```

```
Dim item As Variant
For Each item In objDict.Items
    Debug.Print item
Next
' List all keys in the dictionary
Dim key As Variant
For Each key In objDict.Keys
    Debug.Print key & " --> " & objDict.item(key)
Next
```

To clear your dictionary of all items, use the `RemoveAll` method:

```
' remove all key/item pairs
objDict.RemoveAll
' verify that dictionary is empty
Debug.Print objDict.Count
End Sub
```

**NOTE**

*The prior procedure can be found in the Chap28.accdb database in the companion files.*

Recall that key searches are case sensitive by default. However, you can use the `CompareMode` property to change this behavior. The mode change must be specified right after creation of the dictionary object, before adding any data to the dictionary. Let's look at some code.

```
Sub CaseNotSensitive()
Dim objDict As New Scripting.Dictionary
objDict.CompareMode = TextCompare
objDict.Add "Math", 89
objDict.Add "English", 70
objDict.Add "Chemistry", 90
Debug.Print objDict.Exists("math")
End Sub
```

In the example above, we set the `CompareMode` to `TextCompare` to allow for searches that are not case sensitive. So instead of typing “Math” for the key name, we can simply enter “math” in lower case and the `Exists` method is able to locate the key. The `CompareMode` can also be set to two other settings: `BinaryCompare` and `DatabaseCompare`. `BinaryCompare` performs binary comparisons; it is case sensitive, so “EventName” is not the same as “eventName” or `eventname`. The `DatabaseCompare` performs a comparison based on information in the database. The latter only exists in Microsoft Access (it's not available in other VBA-capable office applications).

## Dictionary versus Collection

The following table shows several advantages of the dictionary object over the native VBA collection object. In your programming endeavors, you should always pick the object that is most suitable for a particular task.

Dictionary	Collection
Retrieving items in a dictionary is faster than in a collection.	Retrieving items is slower than in a dictionary.
Easier to search for a given item	Harder to search for a given item
The Item value can be changed directly.	The item value must be first removed and then the changed item can be added back.
Uses keys to locate a particular item; keys can be checked for existence.	The keys are used to look up data but cannot be retrieved. Uses index values that are harder to work with.
Offers compare mode for changing case-sensitivity	Collections are case-sensitive, and sensitivity cannot be changed.
Key values can be any data type.	Key values must be strings.
Items can be easily removed using the RemoveAll method.	Removing items from a collection requires re-defining the Collection object.
Cannot store a reference to a custom collection	Can store a reference to a custom collection
Requires a reference to the Microsoft Scripting Runtime Library or an object created using the late binding	Collection is an object available in the VBA library.

## Action Item 28.1

Included in the companion files is an Order Entry and Lookup form (see Figure 28.3) that uses the VBA dictionary object for its various operations. Look for the file called Chap28\_Dictionary.accdb.

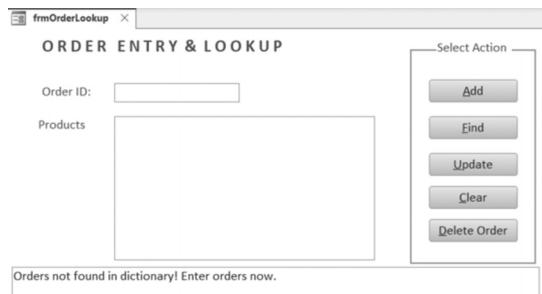
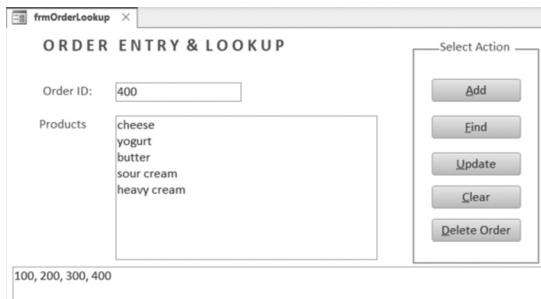


FIGURE 28.3 This Demo Form employs the VBA Dictionary Object for its order entry and lookup tasks.



**FIGURE 28.4** Entering an order via the Demo form.

## INTRODUCTION TO REGULAR EXPRESSIONS

---

As you know, the VBA has many useful functions that apply to strings. You can use the `Len` function to return the number of characters in a string, for example,

```
?Len("Today is Sunday")
```

returns the number 15.

You can use the `Left`, `Right`, and `Mid` functions to return a portion of a string. For example,

```
?Left("Today is Sunday", 5)
```

returns the string `Today`. Replace `Left` with `Right`, and you return the five rightmost characters from that string. If you need to return a string starting from a particular character in the string, you can use `Mid` function to extract the required portion of the string. For example, let's extract "is" from our test string:

```
?Mid("Today is Sunday", 7, 2)
```

This returns the string: `is`.

If you instead want to find the position of the first occurrence of one string within another string, you can call the `InStr` function like this:

```
? InStr("Today is Sunday and it's a holiday", "Sunday")
```

This will display the number 10, indicating that the string was found starting at character 10. If you don't need to search from the beginning of the string, specify the position to search from in the first argument, like this:

```
?InStr(5, "Today is Sunday and it's a holiday", "o")
```

This searches for the letter "o", starting from the fifth character in the specified string. The "o" is found in the 29<sup>th</sup> position.

As you can see, using string manipulation functions in VBA is straightforward. However, there will be many situations in your VBA programs where these methods will not be enough. You may need to match a certain pattern of characters in various expressions. A pattern match can involve a character, a word, a group of words, or an entire sentence. This is where a working knowledge of regular expressions can help. Simply put, a *Regular Expression (often referred to as RegEx or RegExp)* is a sequence of characters that specifies a search pattern in text. Regular expressions are commonly used in search engines, Find/Replace dialogs in text editors and word processing applications. Before you can try out some pattern matching in VBA, let's go over a few basic concepts.

### **Character Matching in RegExp Patterns**

---

The *pattern* is basically a schema that you put together to match character combinations in strings. Patterns are composed of simple characters such as /abc/ or a combination of simple and special characters such as /ab\*c/. The latter indicates that we want to match a single “a” followed by zero or more “b” characters followed by “c.” The \* after “b” means zero or more occurrences of the preceding item. The following table shows some examples of special characters used in regular expressions patterns. For more detailed information, you will need to look to other sources. This section supplies just a bare minimum of the regex knowledge for you to complete a specific VBA programming task that will be introduced later in this chapter.

Pattern	Description	Example	Found Matches
. (a single dot)	Matches any single character except vbNewLine	d.g	deg, dig, dog
[characters]	Matches any single character between brackets []	[jv]	Would match j and v in “java”
[^characters]	Matches any single character that is not between brackets []	[^jv]	Would match “a” in “java”
[start-end]	Matches any character that is part of the range in brackets []	[0-9] [A-Z]	Matches any number in the range 0 to 9 Matches any character in the range “A” to “Z”
\	Escapes special characters so that special characters can be searched for	\[	Escaped character. Matches a “[” character.
\n	New line	\n	Matches a new line (vbNewLine)

(Contd.)

Pattern	Description	Example	Found Matches
\r	Carriage Return	\r	Matches a carriage return (vbCr)
\t	Tab	\t	Matches a tab character (vbTab)
\w	Matches any word character alphanumeric and underscore.	\w	Would match “morning” in “morning”
\W	Matches any non-alphanumeric characters and the underscore	\W	Would match “@” in your email address
\s	Matches any white space character (spaces, tabs, line breaks)	\s	Would match the space in “good morning”
\S	Matches any non-white space character	\S	Would match “good” and “morning” in “good morning”
\d	Matches any decimal digit	\d	Would match “7” in “7Eleven”
\D	Matches any character that is not a digit character (0-9)	\D	Would match “Eleven” in “7Eleven”

### Quantifiers in RegExp Patterns

Regular expression patterns can include quantifiers that allow you to control how many times a match occurs.

Quantifier	Description	Example	Found Matches
*	Matches zero or more of preceding characters/digits	b\w*	Matches a “b” character Matches any word character (alphanumeric & underscore) Matches 0 or more of the preceding items
+	Matches 1 or more of the preceding characters/digits	b\w+	Same as above, but matches 1 or more
?	Matches zero or one	colou?r	Matches color, colour
{n}	Matches “n” many times	c{2}	Matches a “c” character twice. Case sensitive. Will match the second and third letter “c” in “Cecilia is sick.”
{n,}	Matches at least “n” occurrences of the preceding item	a{3,}	Matches all of the “a” character in “baaarber,” “baaaaarber”
{n,m}	Matches the specified quantity of the previous character/digit	a{1,3}	Will match 1 to 3 of the previous items. It will match the “a” in “barber,” the two “a’s in “barber,” and the three “a’s in “baaarber”

You can use parentheses () to group multiple items in your pattern together. You can also indicate the beginning and ending of lines and words and other patterns indicating that a match is possible. These topics are beyond the scope of this book. Numerous books have been devoted to the subject of regular expression pattern matching operations. Hundreds of websites offer useful tools that will help you decipher an unknown regex pattern and help you understand its building elements. While regex can take some time to learn, once mastered, these skills can be reused in many other programming languages for validating, replacing, and extracting data from strings.

### Using the RegExp Object in VBA

Many programming languages provide built-in support for working with regexes. To use regexes in your VBA programs and take advantage of the built-in programming assistance (Intellisense), you will need to add a reference to an external library. In your VBE screen, choose Tools | References dialog box (see Figure 28.5), scroll down and select the *Microsoft VBScript Regular Expressions 5.5*, and then click OK. Once you've added this library to your VBA project, the Object Browser (press F2) will display all available properties and methods of the RegExp object (see Figure 28.6).

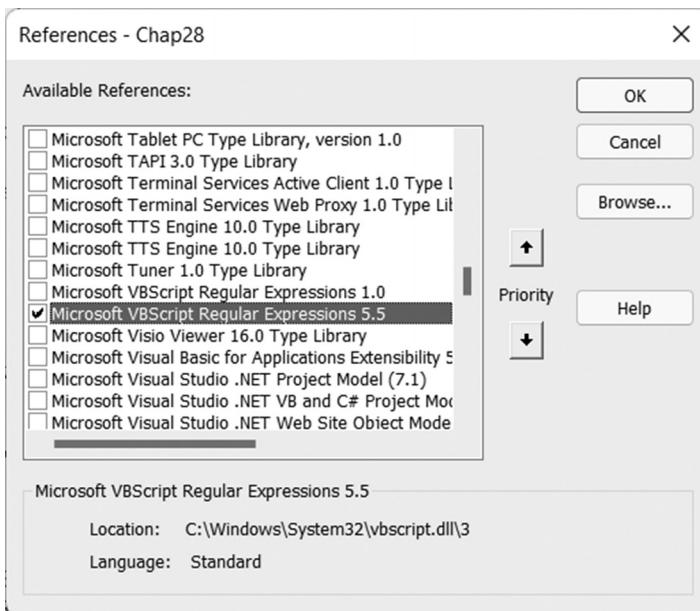


FIGURE 28.5 Setting a Reference to the Microsoft VBScript Regular Expressions 5.5 Object Library.



FIGURE 28.6 Regular Expressions can be used via the VBScript Regular Expressions 5.5 library displayed as VBScript\_RegExp\_55 in the Object Browser.

## The RegExp Object Declaration

If you already set up the reference to the *Microsoft VBScript Regular Expressions 5.5 library*, you can declare the RegExp object using the following early-binding syntax:

```
Dim oRegExp As RegExp  
Set oRegExp = New RegExp
```

If you don't want to add the reference to the RegExp object via the References dialog box, use any of the examples below:

```
Dim oRegExp As Object  
Set oRegExp = CreateObject("VBScript.RegExp")
```

Or

```
Dim oRegExp As Object  
Set oRegExp = New VBScript_RegExp_55.RegExp
```

## RegExp Properties

The RegExp object offers four properties that allow you to specify the pattern to be matched and various options that should be turned on or off. The most

important property is the `Pattern` property. This is where you specify the pattern that you are going to use for matching against the string.

The `Global` property can be set to `True` or `False`. Set it to `True` to find all matches in the pattern. If set to `False` only the first match will be found.

The `IgnoreCase` property set to `True` will make the matching case insensitive.

The `Multiline` property should be set to `True` if your string consists of multiple lines and you want to run the same pattern through all the lines.

## RegEx Methods

---

To work with the `RegExp` object, you can use the following methods:

Use the `Test` method to search for a pattern in a string. When a match is found, `True` is returned.

The `Replace` method allows you to replace the occurrences of the pattern with a replacement string.

The `Execute` method will return all the matches of the pattern that were found in the provided string.

## Writing VBA Programs Using the RegEx Object

---

Let's create example procedures that use some of the methods and properties of the `RegExp` object. The first procedure will use the `Test` method to check whether the supplied text string contains a correctly formatted phone number. Assume we want to ensure that phone numbers are in the following US format: (999) 999-9999.

NOTE: All code files and figures for the hands-on projects may be found in the companion files.

### ① Hands-On 28.1 Testing for a Pattern Match

1. Create a new database called `Chap28.accdb` in your `C:\VBAAccess2021_ByExample` folder.
2. In the VBE screen, add a new Module and rename it `RegExpressions`.
3. Enter the following VBA procedure and run it.

```
Sub RegExp_TestDemo()
    Dim oRegExp As RegExp
    Dim strToSearch As String
    Set oRegExp = New RegExp
    strToSearch = "Customer phone number: (201) 234-7899."
    
    ' match US Phone numbers in the format: (999) 999-9999
    oRegExp.Pattern = "\(\d{3}\) \d{3}-\d{4}"
```

```
MsgBox oRegExp.Test(strToSearch)
End Sub
```

When this procedure executes, you should see “True” in the message box.

4. Remove the parentheses surrounding the area code and run the procedure again.

The phone number will no longer match the pattern we defined, so the Test method will return False.

Let’s look at the pattern expression we defined for the phone:

```
\(\d{3}\) \d{3}-\d{4}
```

Element	Explanation
\(	Escaped character. Matches a “(” character.
\d	Matches any digit character (0-9).
{3}	Quantifier. Matches 3 of the preceding digits.
\)	Escaped character. Matches a “)” character.
	Matches a space character (empty space)
\d	Matches any digit character (0-9).
-	Dash character. Matches a “-” character.
{4}	Quantifier. Matches 4 of the preceding digits.

In the next Hands-On, we will write a procedure that searches for all occurrences of characters provided in the pattern and replaces them with nothing, meaning that they will be removed from the search string.



## Hands-On 28.2 Replacing All Occurrences of the Pattern Match

1. In the VBE window, enter the code of the procedure shown in Figure 28.7.

```
Sub RegExp_ReplaceDemo()
Dim oRegExp As RegExp
Set oRegExp = New RegExp

Dim strToSearch As String

strToSearch = """userId":1,"title":"Account Manager","taskCompleted":false"""

' Find all matches in the pattern
oRegExp.Global = True

' Match any character in this set
oRegExp.Pattern = "[\\\""]"

MsgBox oRegExp.Replace(strToSearch, "")

End Sub
```

**FIGURE 28.7** This VBA procedure uses the Replace method of the RegExp object to remove characters specified in the pattern from the searched string.

2. Execute the `RegExp_ReplaceDemo` procedure.

Figure 28.8 displays the message box with the resulting string.

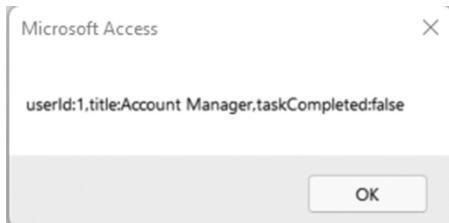


FIGURE 28.8 The Replace method of the `RegExp` object has successfully removed all occurrences of { } “” characters from the original string.

Note that prior to defining the pattern, we used the `Global` property of the `RegExp` object to ensure that all the characters specified in the `Pattern` are removed, not just their first occurrence.

3. Comment out the line of code that sets the `Global` property to `True` and run the procedure again.

This time, the message box should display partially cleaned-up string; only the first occurrence of each character in the pattern was removed, giving you the following text string:

```
"userId":1,"title":"Account Manager","taskCompleted":false}
```

Finally, let's create a procedure that uses the `Execute` method of the `RegExp` object to display all the matches for the pattern that was found in the search string. We will use the same pattern and search string as in the previous procedure.



### Hands-On 28.3 – Obtaining a List of all the Matches in the Pattern

1. Create a copy of the procedure from Hands-On 28.2 and rename it `RegExp_ExecuteDemo`.
2. Add the following two declaration statements:

```
Dim matches As Object  
Dim itm As Variant
```

3. Uncomment the statement that sets the `Global` property to `True`.
4. Comment out the statement that displays the message box. We will not be making any replacements in this procedure.

5. Add the following code before the End Stub statement:

```
Set matches = oRegExp.Execute(strToSearch)
For Each itm In matches
    Debug.Print itm
Next
```

6. Run the completed RegExp\_ExecuteDemo procedure.

After running the procedure, you should see all the matches that were found, listed on separate lines like this:

```
{
"
"
"
"
"
"
"
"
}
```

Now you've learned the basics of using RegExp object in your VBA programs. We will revisit this object in a later Hands-On project when we need to locate and extract specific strings from a JSON response obtained from an external resource. If you've never heard about JSON, read on. The next section introduces you to communicating with Web servers: sending requests and processing responses.

## INTRODUCTION TO REST API

---

Over the past few years, the method that developers use to connect to external resources and share information between various computer systems frequently includes a mysterious initialism: API. This vastly popular term is formed from the initial letters of *Application Programming Interface*.

Application Programming Interfaces (APIs) allow programs and scripts to communicate with each other. These programming interfaces expose certain data, services, and functionality of an application so other developers can use them. This allows one product to interact with other products. In other words, the APIs are specifically built to be consumed by another application programmatically. APIs allow you to automate many tasks, create user-friendly

dashboards and client applications both for mobile and Web use. More than that, they allow you to extend your product functionality by grabbing the required resource from somewhere else. Basically, APIs make things easier. Think of how many times you have seen a Google map embedded in some website. That map came from the Google Maps API! Many popular websites, such as YouTube, Twitter, Facebook, as well as multitude of commercial and government websites, provide APIs that allow you to get and update their data. To use these APIs, you don't need to know how they were created internally. All you need is to get acquainted with their documentation, find out how to ask for what you need and how to process the response. Some APIs are free, others require that you pay for the service. Some will ask you to create a developer account to obtain an API key for the authentication purposes. This key consists of a set of letters and numbers that uniquely identify you to the application. An API key is like a password; it is important to keep it secure.

There are different types of APIs and the one that is most popular now is called *Representational State Transfer (REST)*. The REST API, also referred to as a RESTful API, was created by computer scientist Roy Fielding. REST is a set of rules (also known as architectural style) that developers must follow to create programs on a server that allow communication with various client applications. In a RESTful system, a client application sends a request to the server usually over the HTTP protocol. This request might be to fetch data (GET request), alter the state of the data (PUT request), create data (POST request), delete some data (DELETE request), or modify some details about the resource (PATCH request). The POST, GET, PUT and DELETE are special verbs that specify a CRUD (Create, Read, Update, Delete) action that needs to be performed on the server. After the server completes the action, it sends a response back to the client application, often in the form of a representation of the requested resource. A very important fact in the REST API is that all client requests are stateless. This means that each request must contain all the necessary information for the server to process the request. In other words, the client cannot depend on the server to remember prior requests. Server responses can be formatted in plain text/HTML, XML, or JSON.

In this section, you learn how to use VBA to use a GET method to fetch data from some free APIs that you can access without applying for a developer account or using authentication. There is a lot to know about REST API and, unfortunately, there isn't enough room in this chapter to cover all methods of using it. Even the simple GET method can get quite complex when you need to pass parameters to the service and authenticate yourself. Thus, for simplicity's

sake, we will focus on the basic syntax that should provide you with enough understanding of the topic, so you are ready to learn more about it the next time you encounter it.

### Accessing REST APIs with VBA

VBA does not have a special method for accessing REST APIs. However, there is a special `XMLHttpRequest` object that allows you to use external API from your VBA code. You must declare `XMLHttpRequest` object by using an early or late binding.

As mentioned earlier in this chapter, to use an early binding, you must set up a reference to an external library. In this case, the library you need is called *Microsoft XML, vb6.0*. Figure 28.9 shows the selection of this library in the References dialog box. You will need to scroll down in the list of Available References to find it.

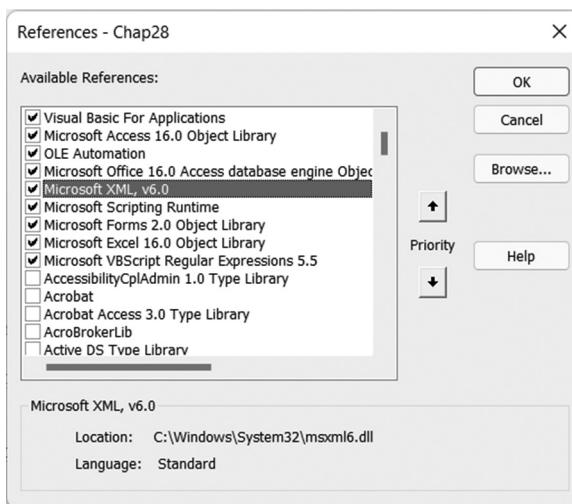
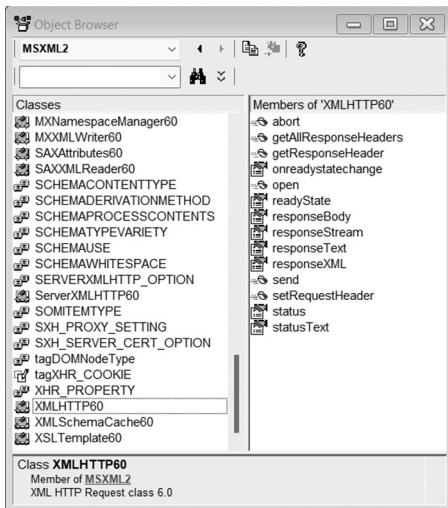


FIGURE 28.9 Activating the Microsoft XML, vb.60 Object Library

Once the library is selected in the References dialog box, you can view the object's properties and methods (see Figure 28.10) using the Object Browser (View | Object Browser or Press F2).



**FIGURE 28.10** Exploring the Microsoft XML, vb.60 Object Library using the Object Browser.

To use this object, declare it like this in your VBA code:

```
Dim httpReq As MSXML2.XMLHTTP60
Set httpReq = New MSXML2.XMLHTTP60
```

Or put everything in the declaration line:

```
Dim httpReq As New MSXML2.XMLHTTP60
```

Note that the `httpReq` is the name of the object variable; here, you can specify any name you like. Some people prefer to call it `xmlhttp`; others use `xhr`; use the name that you feel most comfortable with.

With the late binding, you can skip selecting a reference. Simply declare the object variable of the generic object type and use `CreateObject` function like this:

```
Dim httpReq As Object
Set httpReq = CreateObject("MSXML2.ServerXMLHttp")
```

In this chapter's examples, we will be using the early binding so we can rely on the built-in programming assistance while writing our code.

### **Methods and Properties of the XMLHttpRequest Object**

---

Let's look at the methods and properties that we need to be familiar with to make successful requests to any external API.

The most important methods are `Open` and `Send`. Use the `Open` method to initialize a request. You will need to provide the two required arguments. The first argument is a method which can be either GET, POST, or PUT. The second argument is the URL of the resource you are calling. You may optionally pass a Boolean value (TRUE/FALSE) to indicate whether the API call is meant to be asynchronous (TRUE is default) or synchronous (FALSE). If you pass False, processing waits until the response is returned from the server.

Using the `httpReq` object variable defined earlier, here is how you would set up a basic GET request:

```
httpReq.Open "GET", "strURL ", False
```

where the `strURL` is the URL address of the server you want to access.

The actual request to the server is made using the `Send` method. If the request was declared asynchronous (TRUE) then this method returns immediately, otherwise it waits until the response is received. You can pass additional optional arguments with the `Send` method. The simple syntax of this method is shown below:

```
httpReq.send
```

The `Send` method will send the request to the resource you indicated in the second parameter of the `Open` method.

The `Abort` method is handy for aborting the current request. You will use it when you want to stop the request.

Any problems and issues with an API can often be resolved by examining API headers, and we have three methods to deal with headers: `setRequestHeader`, `getResponseHeader`, and `getAllResponseHeaders`. Headers provide extra information about each API call and response. The most common API headers are listed in the following table:

<b>Authorization</b>	This header contains the authentication credentials for HTTP authentication.
<b>WWW-Authenticate</b>	The server may send this header if it needs some form of authentication before sending the response for the requested resource. It may include an error code 401, which means “unauthorized.”
<b>Accept_Charset</b>	The client may send this header with a request to let the server know which character sets (UTF-8, ISO-8859-1, Windows-1251, etc.) are acceptable by the client.
<b>Content-Type</b>	This header tells the client what media type (e.g., application/xml or application/json) the response is sent in. This helps client to correctly process the response received from the server.

<b>Cache-Control</b>	This header contains caching directives (instructions) defined by the server for the response. These directives determine how a resource is cached, where it's cached and its maximum age before expiring. The no-cache entry in the Cache-Control header indicates that returned responses can't be used for subsequent requests to the same URL before checking if server responses have changed.
----------------------	---

To get all the response headers from the HTTP request, use the `getAllResponseHeaders` method. This method must be used after you've the `Send` method to send the request. For example,

```
httpReq.send  
Debug.Print httpReq.getAllResponseHeaders
```

The second statement above should print to the Immediate window the string containing response headers. What you get in the response headers depends on the server. Here is an example of the headers obtained from the resource we'll be querying in Hands-On 28.4:

```
content-type: application/xml  
expires: -1  
server: Microsoft-IIS/10.0  
x-aspnet-version: 4.0.30319  
request-context: appId=cid-v1:39f1cd0a-de7e-435f-bf7d-  
39de930d88c6  
access-control-expose-headers: Request-Context  
x-powered-by: ASP.NET  
date: Thu, 14 Apr 2022 21:48:23 GMT  
strict-transport-security: max-age=31536000 ; includeSubDomains  
; preload
```

As you can see from this example, the headers are a key / value pairs in text format separated by a colon.

To find out a specific header value you want, use the `getResponseHeader` method, providing it with the argument denoting the specific header value you want. For example, to get the value of the `content-type` header, use the following line of code:

```
Debug.Print HttpReq.getResponseHeader("content-type")
```

The `setRequestHeader` method is used by the client to provide information to the server about the types of content that are acceptable for the response, the acceptable character sets, list of acceptable encoding etc. For a full list of standard request fields see the article at [https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_header\\_fields#Requests](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Requests). It is the responsibility of the server to

consider the sent-in client requirements. The `setRequestHeader` method has two required arguments that specify the name of the header and its value. You must call it after calling `Open` method, but before calling `Send`. Here is an example:

```
httpReq.Open "GET", strURL, False  
httpReq.setRequestHeader("Accept", "text/xml")  
httpReq.Send
```

The prior example tells the server that the client is looking for a response in text/xml format.

**NOTE**

*Before sending a request to the server, you may want to query the server to find out if the server is operational and what server resources are available. By sending a “HEAD” request, instead of GET or POST, the server will send back its response headers. Here is a short VBA procedure that demonstrates how to query the server:*

```
Sub GetOnlyHeaders()  
Dim xhr As New MSXML2.XMLHTTP60  
xhr.Open "HEAD", "https://itunes.apple.com/  
    search?term=celine+dion", False  
xhr.send  
  
If xhr.ReadyState = 4 Then  
    Debug.Print xhr.getAllResponseHeaders  
End If  
End Sub
```

With the methods covered, let's look at the `XMLHttpRequest` object's properties.

The `readyState` property specifies the state of the request. There are five possible values:

- 0 = uninitialized
- 1 = loading
- 2 = loaded
- 3 = interactive
- 4 = complete

Every time the `readyState` changes, the `onreadystatechange` event is fired. When `readyState` is 4 and `status` is 200, the response is ready.

The `status` property returns the HTTP status code from the server. The most common status is 200, which means “OK,” signifying that the request was successful. The 403 status means that the access is forbidden. The 404 denotes that the requested page / resource was not found. For a complete list of statuses, go to [https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes).

The `statusText` property returns the text version of the HTTP status code. It is useful for programming error messages. You will see the example of using `status` and `statusText` in the Hands-On 28.4.

Finally, the last four properties `responseText`, `responseXML`, `responseStream`, and `responseBody` specify the form in which the HTTP response can be returned. Use the `responseXML` property if you know that the response body is an XML formatted text. If you need the response body as a string, use the `responseText` property. The `responseStream` property will return server response in the form of binary-encoded data (UTF-8, UCS-2, UCS-3, Shift\_JIS, and so on). The `responseBody` property is useful when the response body is binary. This property returns the response content as a byte stream.

### Making a Basic GET Request

---

You may be wondering how exactly you can put this newfound knowledge to use. Here again, the best learning you can benefit from is working out an example.

In the first Hands-On example on calling REST API, we access the NHTSA (National Highway Traffic Safety Administration) Product Information Catalog and Vehicle Listing (vPIC) API Programming Interface. The following is a direct link: <https://vpic.nhtsa.dot.gov/api/>

It describes API methods that can be used to get various types of data in XML, CSV, and JSON format. We will get a list of all the makes available in vPIC dataset. We will get this list in XML format and will push it into an Access table. Let's get started.



#### Hands-On 28.4 – Requesting Data from REST API

1. In the VBE screen, insert a new standard module to your Chap28.accdb database. Use the Properties box to rename the module `APIRequest_XML`.
2. In the `APIRequest_XML` module code window, enter the following procedure:

```
Sub RequestData_XML_toAccessTable()
    Dim httpReq As MSXML2.XMLHTTP60
    Dim Resp As New MSXML2.DOMDocument60
    Dim strURL As String
    Dim strFileName As String

    strURL = "https://vpic.nhtsa.dot.gov/api/vehicles/getallmakes"

    Set httpReq = New MSXML2.XMLHTTP60
    httpReq.Open "GET", strURL, False
    httpReq.send
```

```

Debug.Print httpReq.getAllResponseHeaders
If httpReq.status = 200 Then
    strFileName = "C:\VBAAccess2021_ByExample\AllMakes.xml"
    Debug.Print httpReq.responseText
    Resp.LoadXML httpReq.responseText
    Resp.Save strFileName
    Application.ImportXML _
        DataSource:=strFileName, _
        ImportOptions:=acStructureAndData
Else
    Debug.Print "Error " & httpReq.status & "-" & httpReq.
statusText
End If
Set httpReq = Nothing
End Sub

```

3. Click in the selection bar next to the `httpReq.send` statement in the procedure to put a breakpoint on this line of code and press F5 to run the procedure.
4. When the break mode is activated and you see the yellow selection in the code window, keep pressing F8 to debug the code line by line. Keep the Immediate window open while debugging the procedure so you can see the output of the Debug statements. Activate the Immediate window by pressing Ctrl+G.

The first debug statement in this procedure should return the contents of the headers, as shown earlier in this chapter. Notice that we are asking for the headers after we have sent the request to the server. This statement is for demonstration purposes only. Most of the time, you don't need to look at headers unless problems arise, and you need to troubleshoot some issues. After sending the request, the server gives you back some status code. 200 means that the request was successful, and the server produced the response. Only then, we should proceed with the rest of the code. There is no point of asking for a response when an error code was returned in the server status property. If the status is not 200, we will output to the Immediate window the status code and the equivalent text error message using the `status` and `statusText` properties discussed earlier. If request was successful, we use the `responseText` property to get the response and print it to the Immediate window. Debug statements help you understand the output from the server. Now that we obtained the output, we also want to save it so we can use it for a specific purpose. In this procedure, we use the `LoadXML` method of the `XMLDocument` object to load the server response into an XML document which we declared at the top of the procedure in the `Resp` object variable. Note that we have extensively covered working with XML in Chapter 27. To produce an

XML file with the string returned from the server, we use the Save method of the `XMLDocument` object. After the XML file is produced (`AllMakes.xml`) we use the `ImportXML` method of the Access Application object to import the structure and the data into a new table in the current database.

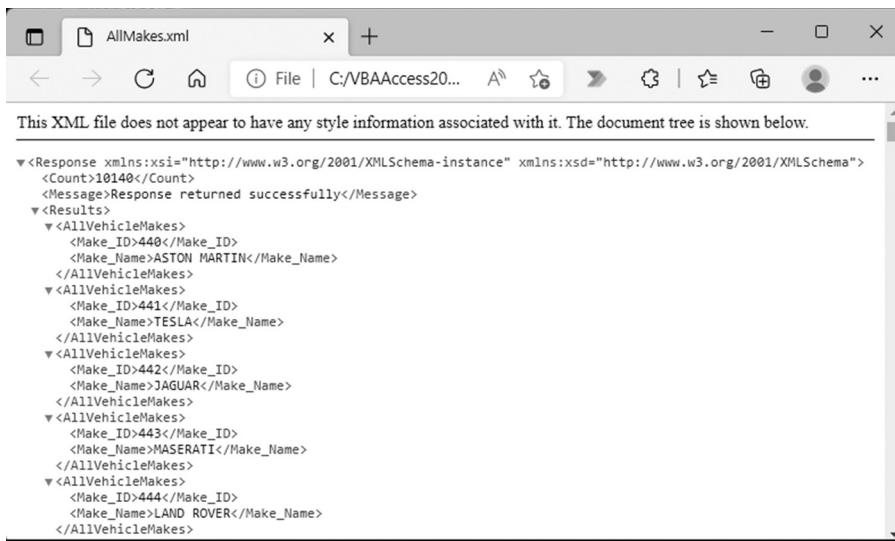
5. When you have reached the end of the procedure by pressing the F8 key, switch to the main Access application window to view the result. In the navigation bar, you should have two tables named `AllVehicleMakes` and `Response`. A total of 10140 records were written to the `AllVehicleMakes` table. The `Response` table is like a control file that provides the total count of records retrieved and the corresponding text message. Figures 28.11 and 28.12 illustrate the contents of these two tables, while Figure 28.13 shows the format of the XML file that provided data for these tables.

AllVehicleMakes	
Make_ID	Make_Name
440	ASTON MARTIN
441	TESLA
442	JAGUAR
443	MASERATI
444	LAND ROVER
445	ROLLS ROYCE
446	EBR
447	JIALING
448	TOYOTA
449	MERCEDES-BENZ
450	FREIGHTLINER
451	FULMER FABRICATIONS
452	BMW

FIGURE 28.11 The `AllVehicleMakes` table was created from the REST API response.

Response	
Count	Message
10140	Response returned successfully
*	

FIGURE 28.12 The `Response` table was created from the REST API response.



```
<?xml version="1.0" encoding="utf-8"?>
<Response xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Count>10140</Count>
  <Message>Response returned successfully</Message>
  <Results>
    <AllVehicleMakes>
      <Make_ID>440</Make_ID>
      <Make_Name>ASTON MARTIN</Make_Name>
    </AllVehicleMakes>
    <AllVehicleMakes>
      <Make_ID>441</Make_ID>
      <Make_Name>TESLA</Make_Name>
    </AllVehicleMakes>
    <AllVehicleMakes>
      <Make_ID>442</Make_ID>
      <Make_Name>JAGUAR</Make_Name>
    </AllVehicleMakes>
    <AllVehicleMakes>
      <Make_ID>443</Make_ID>
      <Make_Name>MASERATI</Make_Name>
    </AllVehicleMakes>
    <AllVehicleMakes>
      <Make_ID>444</Make_ID>
      <Make_Name>LAND ROVER</Make_Name>
    </AllVehicleMakes>
  </Results>
</Response>
```

FIGURE 28.13 The XML data returned from the REST API.

## Action Item 28.2

Sometimes it will be useful to read the response from the REST API straight into an Excel spreadsheet so it can be analyzed prior to importing to Access. Included in the companion files is the APIRequest\_XML2Excel.bas file. It contains the complete VBA procedure that makes the same request to the REST API as we did in the previous Hands-On, but processes the data to Excel. Bring this code file to your Chap28.accdb database using the File | Import File option in the VBE window. Run the procedure provided in the imported module using the Step Into (F8) debug feature so you can analyze the code while it executes.

## Overview of JSON

Using XML with REST API requests and responses requires knowledge of the XML language and a good understanding of the process of generating and modifying XML structures. In recent years, a new text format, known as JSON (pronounced “JASON”) has become an alternative to XML. JSON, which stands for *JavaScript Object Notation*, is a language-independent data format often used for data interchange between disparate systems due to its lightweight format. JSON format is a human-readable text that consists of name / value pairs.

Figure 28.14 displays a sample of the JSON file obtained from the free fake API that is available for testing and prototyping (<https://jsonplaceholder.typicode.com/users>).

```

users.json - File Viewer Plus [Free Version]
File Home
View in File File Info Inspect Save Save As Print
Browser
Find Syntax Tree Line Numbers Ruler
Undo Copy Redo Paste Cut Select All Find Next Code Folding Word Wrap
users.json
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
    "bs": "harness real-time e-markets"
  }
},
{
  "id": 2,
  "name": "Ervin Howell",
  "username": "Antonette",
  "email": "Shanna@melissa.tv",
  "address": {
    "street": "Victor Plains",
    "suite": "Suite 879",
    "city": "Wisokyburgh",
    "zipcode": "90566-7771",
    "geo": {
      "lat": "-43.9509",
      "lng": "-34.4618"
    }
  },
  "phone": "010-692-6593 x09125",
  "website": "anastasia.net",
  "company": {
    "name": "Deckow-Crist",
    "catchPhrase": "Proactive didactic contingency",
    "bs": "synergize scalable supply-chains"
  }
}

```

FIGURE 28.14 Example of a JSON file format.

- An object, which is an unordered set of name / value pairs

Objects begin with { (left brace) and end with } (right brace). Each name is followed by a : (colon), and the name / value pairs are separated by , (comma).

- An array, which is an ordered collection of values

Arrays begin with a [ (left bracket) and end with a ] (right bracket). Values are separated by a , (comma). A value can be a string enclosed in double

quotes, a number, a true or false, an object or an array. Values can be nested so you can create very elaborate JSON structures to fit all your needs.

- A string, which is a sequence of zero or more Unicode characters, wrapped in double quotes using backslash escapes
- A number (e.g., digits 1-9, fractions, and exponents)
- Whitespace (e.g., space, linefeed, carriage return, and horizontal tab)

In the following example, the following JSON object represents three courses in an array called “courses.” Notice that the array starts and ends with square brackets. Within the array are three objects, one for each course:

```
{  
  "courses": [  
    {  
      "courseID": 1001,  
      "title": "Access VBA Programming by Example",  
      "dateOffered": "September 14",  
      "location": "virtual"  
    },  
    {  
      "courseID": 1002,  
      "title": "Excel VBA Programming by Example",  
      "dateOffered": "September 18",  
      "location": "New York, Hilton"  
    },  
    {  
      "courseID": 1003,  
      "title": "PowerPoint VBA Programming by Example",  
      "dateOffered": "December 5-7",  
      "location": "Unspecified"  
    }  
  ]  
}
```

Because JSON is a simple text format, you can use Windows Notepad to create, save, and view JSON files. The above snippet can be found in the companion files as Courses.json.

The JSON format is supported by virtually all modern browsers and because of its smaller size than the XML encoding, JSON can provide larger performance gains when sending larger amounts of data over a network. The format you should use for data exchange (plain text /HTML, XML, or JSON) depends on your specific situation and your project requirements.

When working with JSON data, you may come across another term – *JSONP* or *Jason with Padding*. JSONP allows you to get JSON data from a server in a different domain. This will help you get around the cross-domain security policy that modern browsers implement.

A more detailed description and examples of JSON formatting can be found at [www.json.org](http://www.json.org).

### Loading JSON Data into Access

In the next Hands-On example, we will access another free REST API resource, but this time we will receive the response formatted as JSON. The direct link to that resource is <https://api.zippopotam.us>. The website shows how the API can be used to autocomplete City and State based on the zip code you enter. We will build a similar form in Access (see Figure 28.15) that obtains the Zip code data through that API.

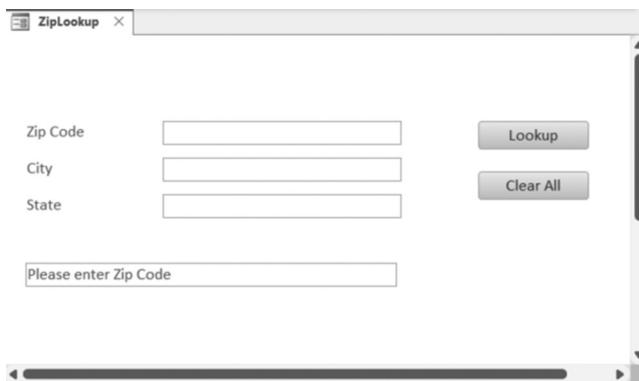


FIGURE 28.15 Access Form used in Zip Code Demo REST API - JSON

If you enter [api.zippopotam.us/us/90210](https://api.zippopotam.us/us/90210) in your favorite browser, you should see the JSON response shown in Figure 28.16.



FIGURE 28.16 JSON structure returned from Zippopotam API

For the JSON format in VBA, we don't have a method that can help us easily extract individual key values from the JSON string. For simple structures like the one shown in Figure 28.16, we can use the basic string functions discussed earlier in this chapter. Or we can use what we've learned so far about Regular Expressions. Let's take the latter route to gain more experience.



### Hands-On 28.5 Requesting Data from REST API (JSON Example)

1. In the **Chap28.accdb** database, create a new Access form, as shown in Figure 28.14.

This form should include

- three unbound text boxes with caption properties set to **Zip Code**, **City** and **State** and name properties set to **txtZip**, **txtCity**, **txtState**
- two command buttons with caption properties set to **Lookup** and **Clear All**, and name properties set to **cmdRequest** and **cmdClearAll**.
- one unbound text box without a label. The name property of the text box is **txtStatus**. This box is used to return any errors received during the lookup process.

In addition, set the following form properties:

**ScrollBars:** Neither

**Record Selectors:** No

**Allow Datasheet View:** No

2. Save the form as **ZipLookup**.
3. Open the ZipLookup form in the Design view and right click the Lookup button. Choose Build Event, select Code Builder from the Choose Builder dialog box and click OK.
4. In the **form\_ZipLookup** code module, enter the following VBA procedure, that will run when the Lookup button is clicked:

```
Private Sub cmdRequest_Click()
    ClearFields

    If Me.txtZip = "" Or IsNull(Me.txtZip) Then
        With Me
            .txtStatus.Visible = True
            .txtStatus = "Please enter Zip Code"
            .txtZip.SetFocus
        End With
    Exit Sub
```

```

End If
If Len(Me.txtZip) <> 5 Or Not IsNumeric(Me.txtZip) Then
    Me.txtStatus.Visible = True
    Me.txtStatus = "Zip code must be 5-digit long."
    Exit Sub
End If
requestData Me.txtZip
End Sub

```

This procedure will make sure that Zip code is not empty and is entered in the correct format. In the first line of this procedure, we will call a ClearFields procedure that will clear the existing entries in the City, State, and Status text boxes and will set the visibility of the Status box to False when it is empty. In the last procedure statement, we will call the requestData procedure and pass it the current value from the Zip text box. The requestData procedure will make a call to the REST API. We will enter it later in the standard code module.

5. Enter the following two procedures in the same Form\_ZipLookup Code module. The first of these procedures will handle the form's Clear All button click.

```

Private Sub cmdClearAll_Click()
    Me.txtZip = ""
    ClearFields
    Me.txtZip.SetFocus
End Sub
Sub ClearFields()
    With Me
        .txtCity = ""
        .txtState = ""
        .txtStatus = ""
        .txtStatus.Visible = False
    End With
End Sub

```

6. Save the changes you've made in the Form\_ZipLookup Code module.
7. Choose **Insert | Module** to add a new standard module to your Chap28 VBA project. Use the properties window to rename it **APIRequest\_JSON**.
8. In the **APIRequest\_JSON** module, enter the code of the **requestData** procedure as follows:

```

Sub requestData(ByVal postcode As String)
Dim httpReq As MSXML2.XMLHTTP60
Dim oRegExp As Object
Dim rec As Variant
Dim aRecords As Variant
Dim aRecord As Variant

```

```
Dim fld As Variant
Dim fldName As String
Dim fldContent As String
Dim strAObj As String
Dim webResponse As String

Set httpReq = New MSXML2.XMLHTTP60
httpReq.Open "GET", "https://api.zippopotam.us/us/" &
    + postcode, False
httpReq.send
If httpReq.status <> "200" Then
    Forms!ZipLookup.txtStatus.Visible = True
    Forms!ZipLookup.txtStatus = "Error Code: " & _
        httpReq.status & " - " & httpReq.statusText
    Exit Sub
End If
'get the entire JSON string
webResponse = httpReq.responseText

Debug.Print "Below is raw web response json" & vbCrLf
Debug.Print webResponse
'convert json string to an array
Set oRegExp = New RegExp

oRegExp.Global = True
oRegExp.Pattern = "[\\[\\]\\{\\}\\}]+"
strAObj = "[{"

If InStr(1, webResponse, strAObj) > 0 Then
    webResponse = Replace(webResponse, strAObj, ", ")
End If

Debug.Print webResponse
webResponse = oRegExp.Replace(webResponse, "")
Debug.Print webResponse

aRecords = Split(webResponse, ", ")
For Each rec In aRecords
    aRecord = Split(rec, ", ")
    For Each fld In aRecord
        fldName = Split(fld, ":")(0)
        fldContent = Split(fld, ":")(1)
        If fldName = "place name" Then
            Forms!ZipLookup.txtCity = fldContent
        End If
    Next
Next
```

```
If fldName = "state" Then
    Forms!ZipLookup.txtState = fldContent
End If
Next
Debug.Print fldName & vbTab & vbTab & fldContent
Next
Set httpReq = Nothing
End Sub
```

9. Save the changes in the module and return to the main Access application window.
10. Open the ZipLookup form in the form view and click the Lookup button. You should see the error message as shown earlier in Figure 28.15.
11. Click the Clear All button. The error message box should disappear.
12. Enter your zip code (or any valid zip code you can recall) and click the Lookup button.

If the Zip code exists, you should see both the City and State text boxes populated. If the zip code is invalid, for example, you've entered 11345, you should see "Error Code: 404."

To retrieve the zip code data, we pass the value from the Zip text box to the RequestData procedure in the string type variable called postcode. Notice that we then add this value to the URL string like this:

```
httpReq.Open "GET", "https://api.zippopotam.us/us/" + postcode, False
```

We send the request to the server in the same way we did it in the previous Hands-On. Our code will stop executing when the status of the response is not equal to 200. Any value but successful execution (200) will not provide us with a response, so we make the Status box visible on the form and load it with the error message. If the server response code is 200, our procedure continues, and we store the server response in the webResponse string variable. We print the response string to the Immediate window so we can examine its format. To get individual values from the JSON string we will convert it to an array using the RegExp object discussed earlier in this chapter. We define the pattern that we want to match as follows:

```
oRegExp.Pattern = "[\\[\\]\\{\\}\\}]+"
```

The Global property of the oRegExp object will ensure that all occurrences of the characters specified in our pattern will be removed when we apply the Remove method. But prior to that, we want to find and replace the character string "[{" that follows the "places" key in the returned JSON string. These

characters indicate that the next section of JSON is an array containing an object. We clean up the string of characters we don't need by using the VBA InStr and Replace functions like this:

```
strAObj = "[{"
If InStr(1, webResponse, strAObj) > 0 Then
    webResponse = Replace(webResponse, strAObj, ", ")
End If
```

After this code completes, we print to the Immediate window the revised string:

```
{"post code": "90210", "country": "United States", "country abbreviation": "US", "places": , "place name": "Beverly Hills", "longitude": "-118.4065", "state": "California", "state abbreviation": "CA", "latitude": "34.0901"}]
```

Notice that “places”: is now followed with a space and a comma.

Next, we use the pattern matching and replace with a comma all the braces and brackets that are remaining in the previous string. As a result, we get the following string:

```
post code: 90210, country: United States, country abbreviation: US, places: , place name: Beverly Hills, longitude: -118.4065, state: California, state abbreviation: CA, latitude: 34.0901
```

Now all that's left to do is split the above string into the name / value pairs so we can get access to individual items. We can use the VBA Split function to break the string each time we encounter a comma and a space:

```
aRecords = Split(webResponse, ", ")
```

The aRecords variable is an array and we can count the number of items we have in it using the UBound function. If you enter ?UBound(aRecords) in the Immediate window during the break mode (while your code is running), you should see 8 as a return value. If you type ?aRecords(0), you will get back the first name / value pair like this:

```
?aRecords(0)
post code: 90210
```

We use the For Each loop to iterate through the array. To make the process easier, we split each of the name / value pairs into individual fields using the colon separating the name from the value. The first (0) item is a field name and the second (1) item is the field content (value). We use the If statement to find

the value for the place name field that we put in the txtCity text box on our form, and then look for the state to get the state. The looping process continues until we find the values we are searching for.

To gain a better understanding of how this code parses the JSON string using the VBA string functions and regular expressions, set a breakpoint on the first Debug statement in the RequestData procedure, then go back to your form, enter a valid zip code, and click the Lookup button. When the VBA encounters the break statement and the code window appears, you can use the Step Into (F8) to step through the entire code line by line asking questions and checking responses in the Immediate window. You can also try to set up some watch expressions that were introduced in Chapter 9.

### Parsing JSON with Third-Party Libraries

---

Parsing REST API responses, especially the complex ones, will not be as straightforward as the example you've tried here. You may need a custom VBA library written specifically to help you handle the intricate JSON format your response can be returned in. Search the Web for this topic and you're bound to find a great set of VBA-JSON tools created by Tim Hall and available via <https://github.com/VBA-tools>. It contains JSON conversion and parsing for VBA (Windows/Mac Excel, Access, and other Office applications). (I have successfully used it in several of my own VBA projects and it made the task of dealing with the JSON output so much easier.) There are other tools that you may also like and find useful in your work. These tools may not work 100% of the time, but they are certainly better than having nothing that's built-into VBA to handle all your data parsing issues that you will come across while making REST API requests. Getting a response and not being able to turn it into the format you can work with can be a painful and time-consuming effort. Do your own research; find the tool you like, read its documentation, and be ready to use it in your project. Now that you know how to program in VBA, you've learned about classes, dictionaries, arrays, and collections, you can create your own tool or improve the one that already exists.

### SUMMARY

---

In this chapter, you explored several external libraries that should help you build more advanced VBA applications. You've learned about the Dictionary object and how it compares to the native VBA collection object. You saw how regu-

lar expressions can make it easier to extract information from JSON formatted data. Finally, you've learned the basics of making a REST API request and parsing both XML and JSON response data. Your real-world assignments will be more complex than the examples presented in this chapter, but these simple examples should help you reach higher levels of VBA development quicker and with more confidence. Good luck with your VBA coding!

## Appendix

# *INSTALLING INTERNET INFORMATION SERVICES (IIS)*

**I**nternet Information Services (IIS) is a Web server application created by Microsoft for use with the Microsoft Windows operating system. The following versions of IIS are currently in use:

- IIS 10 – Windows 10 / Windows Server 2019
- IIS 8.5 – Windows 8.1 / Widows Server 2012 R2
- IIS 8 – Windows 8 / Windows Server 2012
- IIS 7.5 – Windows 7 / Windows Server 2008 R2

The classic version of ASP is not installed by default on IIS 7.5 and later. Therefore, before running the examples in Chapter 27, you need to enable this feature using the Control Panel. Let's walk through the process of getting the Classic ASP to be recognized by your computer.



## Hands-On A.1 Enabling Classic ASP in Windows

1. Use the Start menu to search for Control Panel. Once opened, click on the Programs link, as shown in Figure A-01.
2. Under Programs and Features, click Turn Windows features on or off, as shown in Figure A-02.

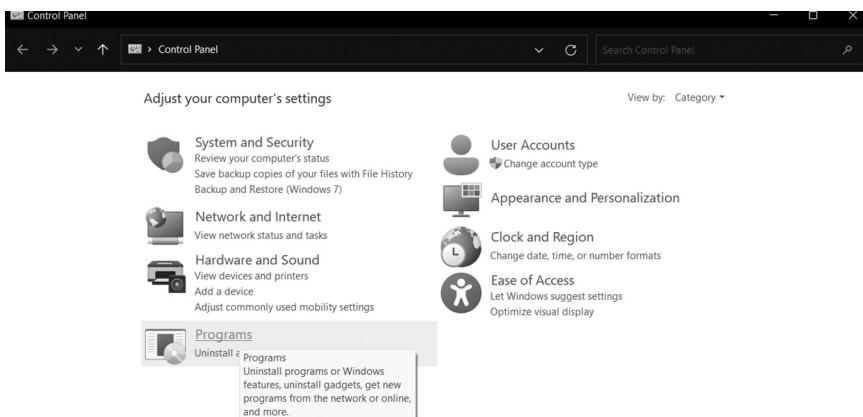


FIGURE A-01. Enabling classic ASP in Windows (Step 1).

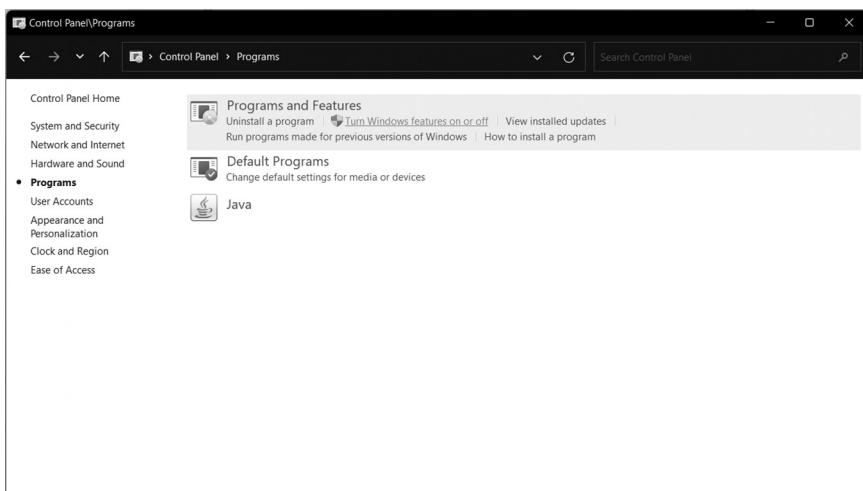


FIGURE A-02 Enabling classic ASP in Windows (Step 2).

3. If you are prompted with a request for permission to continue, click **Continue** in the User Account Control (UAC) window to give Windows permission to proceed.
4. Wait while Windows retrieves all the features.
5. Expand the **Internet Information Services** tree node and make sure your selections under various IIS nodes match those shown in Figure A-03.

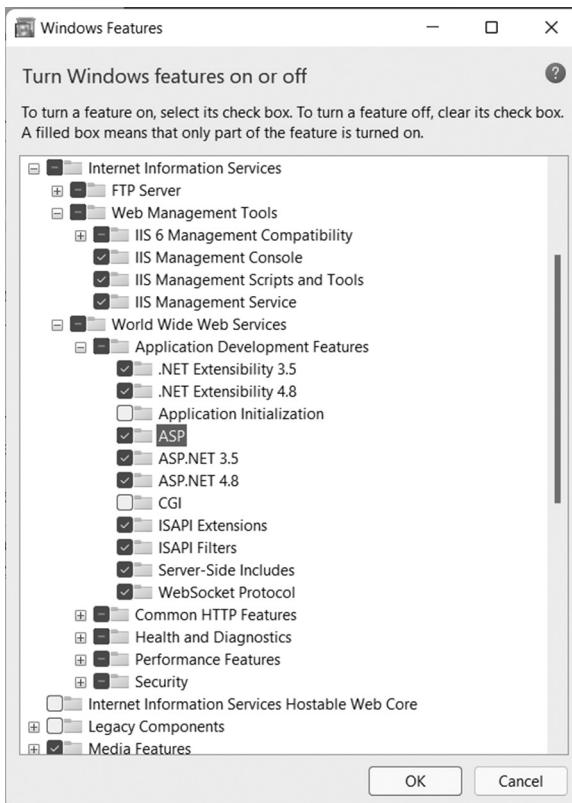


FIGURE A-03 Enabling classic ASP in Windows 10 (Step 3). Make sure ASP is checked under the Application Development Features. Also, IIS Management Console under Web Management Tools should be selected.

6. After checking ASP, click **OK** and wait for Windows to apply the changes. This might take several minutes.
7. Once the features are configured, close all open Control Panel windows.
8. After completing the preceding configuration steps, you should see the folder named **inetpub** on your computer's system drive, as shown in Figure A-04.

**NOTE**

After you have installed IIS, it is important that you run Windows Update to ensure that your system has the most recent security patches and bug fixes.

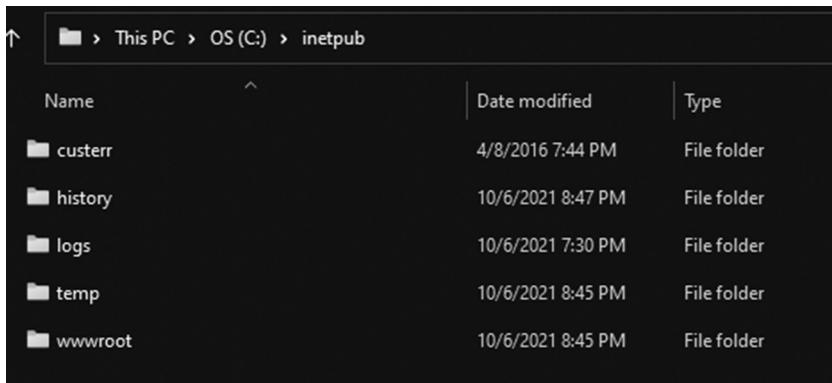


FIGURE A-04 A new folder named *inetpub* should now appear on your computer's system drive.

## CREATING A VIRTUAL DIRECTORY

The default home directory for the World Wide Web (WWW) service is \Inetpub\wwwroot. Files located in the home directory and its subdirectories are automatically available to visitors to your site. If you have Web pages in other folders on your computer and you'd like to make them available for viewing by your website visitors, you can create virtual directories. A virtual directory appears to client browsers as if it were physically contained in the home directory.

**NOTE**

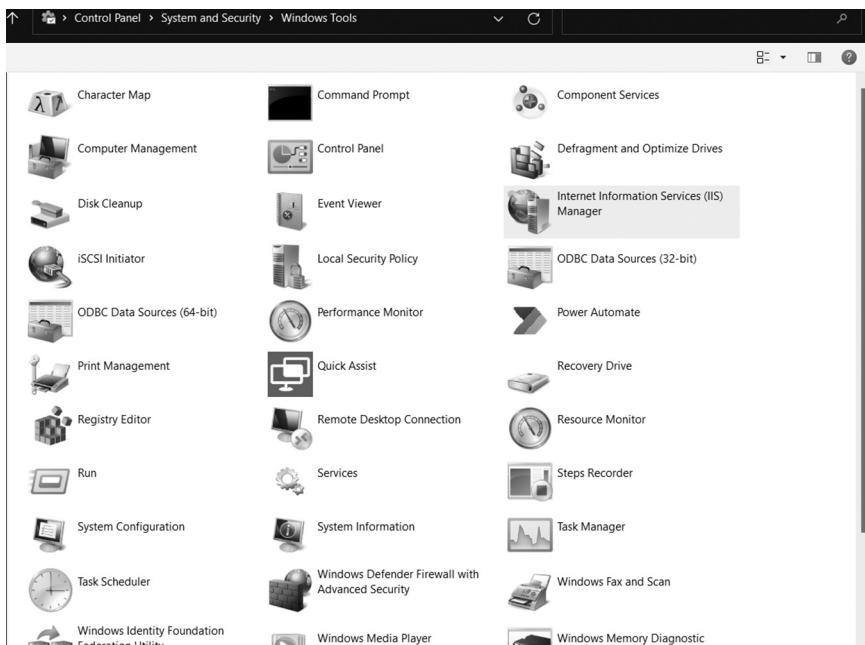
In Chapter 27, you created a directory called VBAAccess2021\_XML (see Hands-On 27.1). In Hands-On A\_2, you will designate it as a virtual directory.



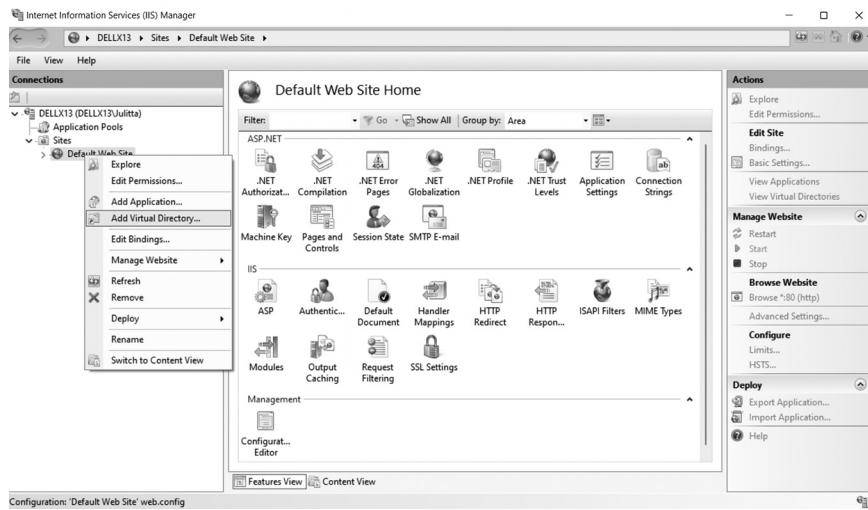
### Hands-On A.2 Creating a Virtual Directory in Windows

1. Open the **Control Panel**, choose System and Security, and then click on **Windows Tools** (Windows 11) or **Administrative Tools** (Windows 10).
2. Double-click **Internet Information Services (IIS) Manager**, as shown in Figure A-05.

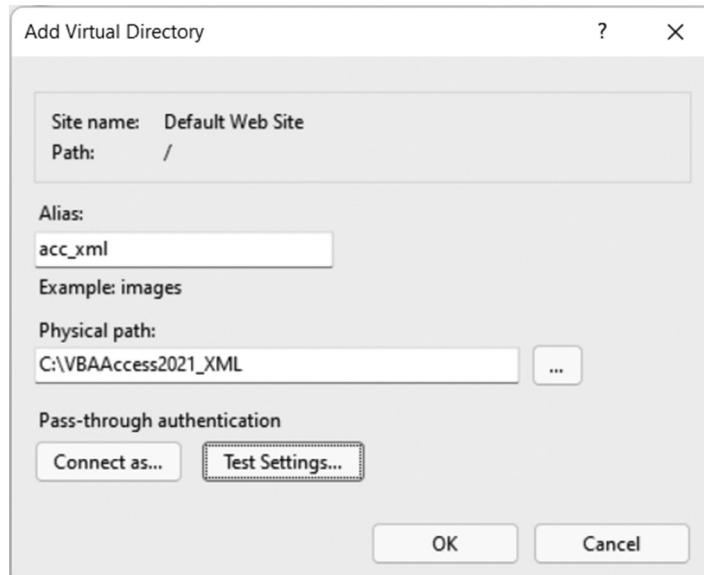
3. Click **Continue** in the User Account Control (UAC) window if Windows asks you for permission to continue. Respond No to any other question.
4. Expand the tree nodes in the Connections pane on the left, right-click on **Default Web Site**, and select **Add Virtual Directory** as shown in Figure A-06. A virtual directory has an *alias*, or name that client browsers use to access that directory. An alias is often used to shorten a long directory name. In addition, an alias provides increased security. Because users do not know where your files are physically located on the server, they cannot modify them.
5. Type **acc\_xml** in the Alias box, as shown in Figure A-07. Set the Physical path to point to the **C:\VBAAccess2021\_XML** folder that you created in Chapter 27 - Hands-On 27.1.
6. Click **OK** to save the changes.
7. Notice the virtual directory named **acc\_xml** now appears under Default Web Site in the Connections pane (Figure A-08). The middle section of the Internet Information Services (IIS) Manager displays the **acc\_xml Home**.
8. Do not close the IIS Manager window, as you will continue with it in the next section.



**FIGURE A-05.** To set up a virtual directory on your computer, you must first activate Internet Information Services (IIS) Manager in the Administrative Tools of the Windows Control Panel.



**FIGURE A-06** You can add a virtual directory by right-clicking Default Web Site in the Connections pane of the Internet Information Services (IIS) Manager window.



**FIGURE A-07.** The Add Virtual Directory dialog box is used to specify the name and path to your Web Site folder. The physical folder named VBAAccess2021\_XML will be shared over the Web as acc\_xml.

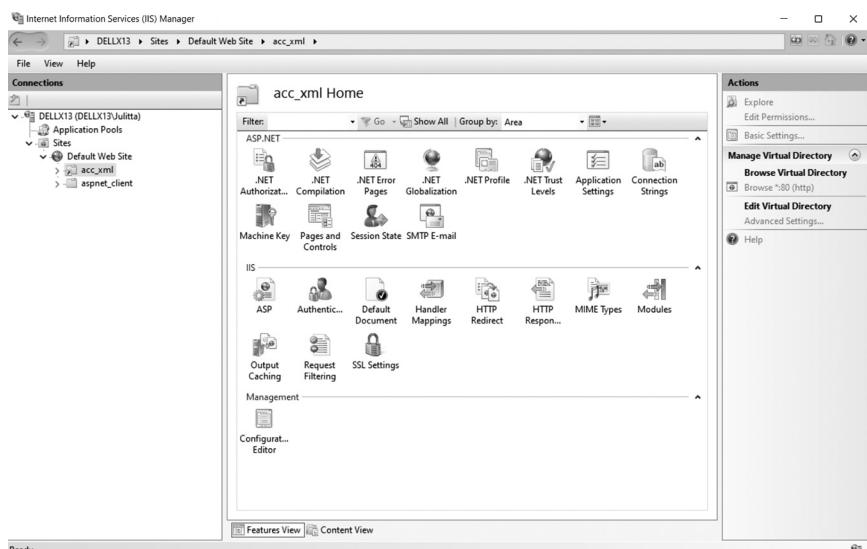


FIGURE A-08 After creating a virtual directory, you should see it listed under Default Web Site in the Connections pane of the Internet Information Services (IIS) Manager window.

## SETTING ASP CONFIGURATION PROPERTIES

To make it easy to debug your code and to ensure that you can use relative paths in your code, you should change a couple of default configuration properties in the IIS Manager. The following hands-on exercise walks you through the steps required to make the necessary modifications.

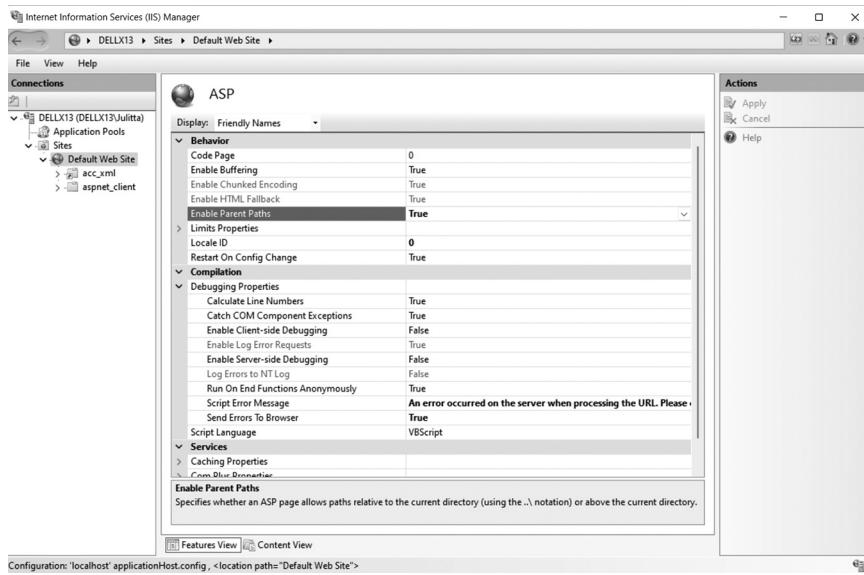


### Hands-On A-3 Configuring ASP Properties

1. In the IIS Manager's Connections pane, select **Default Web Site**, and then in the middle section under IIS, double-click **ASP**.
2. Expand the Debugging Properties tree node and set the **Send Errors To Browser** property to **True**, as shown in Figure A-09.

#### NOTE

*By default, when ASP script errors are encountered, Windows displays the following message: "An error occurred on the Server when processing the URL. Please contact the System Administrator." To prevent this error, be sure to select True next to the Send Errors To Browser property, as shown in Figure A-09.*



**FIGURE A-09.** By setting the Send Errors To Browser property to True, you can easily troubleshoot errors when your Active Server Page encounters an error.

3. In the Behavior section, set **Enable Parent Paths** to **True**, as shown in Figure A-09 in the previous step.  
Parent paths allow you to use relative addresses that contain “..” in the paths of files and folders. In earlier versions of IIS, parent paths were enabled by default. In IIS 7 and above, you need to remember to enable parent paths to prevent errors when relative paths are used.
4. In the Actions area on the right, click **Apply** to save the changes. When changes have been successfully saved, you should see a message in the Alerts area in the right pane of the IIS Manager window that the changes have been successfully saved.
5. Close the Internet Information Services (IIS) Manager window and any Control Panel windows that are still open.

## TURNING OFF FRIENDLY HTTP ERROR MESSAGES

Friendly HTTP error messages don't provide enough information for programmers to effectively troubleshoot script errors. Use the following steps to uncheck the Show friendly HTTP error messages option in your browser so you will get more meaningful error messages that can help you solve your script problems.

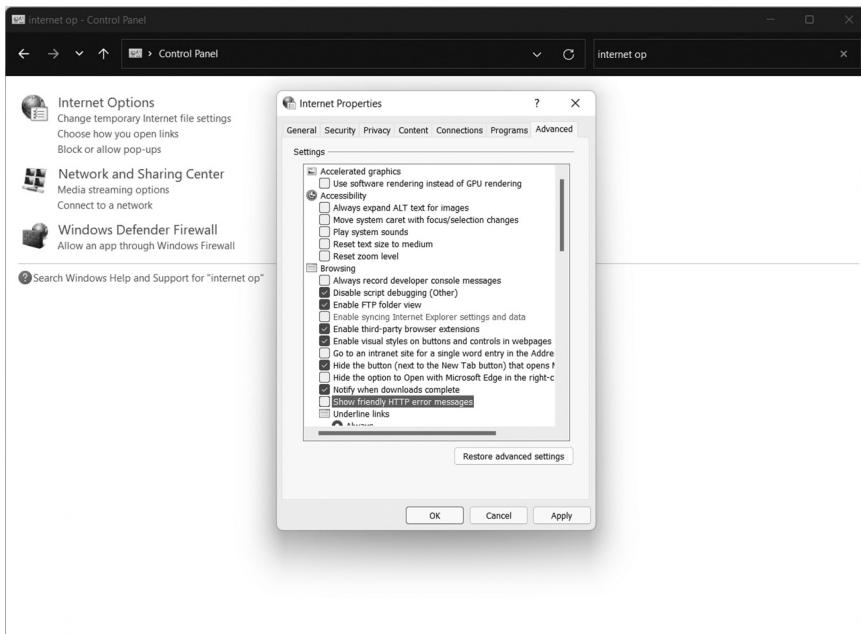
## Hands-On A-4 Turning Off Friendly HTTP Error Messages

1. Open Control Panel and search for Internet Options.
  2. In the Internet Options window, click the **Advanced** tab.
  3. Locate the Browsing settings and uncheck **Show friendly HTTP error messages** as shown in Figure A-10.
  4. Click **OK** to save your changes and exit the Internet Options window.

---

**NOTE**

Your IIS is now configured to run classic ASP scripts on your computer and provide you with meaningful error messages in case errors are encountered in your scripts at runtime.



**FIGURE A-10.** Turn off the Show friendly HTTP error messages option so you can see the actual Windows messages when troubleshooting your ASP scripts.

### ***Important Note***

Your IIS is now configured to run classic ASP scripts on Windows machine (32-bit system). If you are working with the 64-bit operating system, you will need to take additional steps as follows:

- a. Open the **Control Panel**, change the view to show all icons, and then click on **Administrative Tools** or **Windows Tools** in Windows 11.

- b. Double-click **Internet Information Services (IIS) Manager**, as shown in Figure A-05 earlier.
- c. Expand the tree node in the Connections pane on the left, right-click the Application Pools and choose **Add Application Pool**.
- d. In the name box, enter **MyClassicASP**. For the .NET Framework version choose **No Managed Code**. In the Managed Pipeline Mode drop-down, choose **Classic**. After making these selections, click **OK**.
- e. The MyClassicASP entry should now appear in the Application Pool list in the middle section of the IIS Manager window. Right-click this entry and choose **Advanced Settings**.
- f. In the **(General)** section of the Advanced Settings dialog, specify **True** for **Enable 32-bit Applications**.
- g. Click **OK** to close the Advanced Settings dialog.
- h. In the Connections pane on the left, right-click **Default Web Site**, and choose **Manage Web Site | Advanced Settings**.
- i. In the Advanced Settings window, change the Application Pool to **MyClassicASP** and click **OK**.
- j. Close the IIS Manager window.

For more information see the following link:

<http://www.iis.net/learn/application-frameworks/running-classic-asp-applications-on-iis-7-and-iis-8>

Return to Chapter 27, Hands-On 27.5, Step 13.

# INDEX

## A

- AbsolutePosition property, 309–310, 433–434, 445  
.accdb file format, 235, 238, 241, 252, 285, 288, 290, 296, 629, 836, 928  
.accde file format, 235  
.accdt file format, 236  
.accdt file format, 235, 924, 973, 974, Accept\_Charset header, 1082  
Access 2007 database format (ACCDB), 70  
Access 2021 Ribbon interface, 800–803  
    contextual tab, 803  
    Create tab, 800  
    dialog box launcher button, 802  
    More Forms button, 801  
Access Connectivity Engine (ACE), 234  
Access form(s)  
    attachments control, 675, 684–689  
    built in formatting tools, using, 682  
        Themes button, 682  
    creating, 676–679  
        Datasheet form, 676  
        More Forms button, 676  
        Multiple Items form, 676  
        Navigation button, 676  
        Split Form, 676  
    grouping controls using layouts, 680–681  
        Anchoring button, 681  
        Form Layout Tools, 682  
    images in, 682–84  
    rich text support in, 681  
Access report(s)  
    creating, 748  
events  
    Activate, 751–752  
    Close, 751  
    Deactivate, 752  
    Error, 755–756  
    NoData, 752–753  
    Open, 748–750  
    Page, 753–754  
Group, Sort, and Total pane, 765  
OpenArgs property, 767–769  
report section events, 756–753  
    Format, 756–759  
Print, 759–762  
Retreat, 762–763  
Report view, 763–765  
saving reports in .pdf or .xps file format, 766  
sorting and grouping data, 765–766  
Access templates, 973–978  
.accdt file format, 934–938  
    custom blank database template,  
        creating, 973  
Access versions and file formats in Access  
    2007–2021, 235–238  
Access Web Database, 961  
AcDataErrAdded, 735  
AcDataErrContinue, 736, 737, 755  
AcDataErrDisplay, 715, 736, 755  
ACE. *See* Access Connectivity Engine (ACE)  
Action Catalog in Access 2021, 930  
Action queries, 354, 356, 474, 475, 603, 607, 927  
Activate event, 704–705, 751–752  
Active procedure call, 215  
Active Server Pages (ASP), 249, 1004, 1025  
    Configuring ASP Properties, 1105–1106  
    Deleting record, 329–331  
    Friendly HTTP error messages, turning off,  
        1106–1108  
    Internet Information Services (IIS),  
        1099–1108  
    Modifying record, 326–329  
    Virtual directory, creating, 1102–1105  
ActiveConnection parameter, 415  
ActiveConnection property, 385, 399, 424, 464, 485, 488  
ActiveX controls, 524, 530, 928  
ActiveX® Data Objects (ADO), 243, 462  
    ADO Classic *versus* ADO.NET, 245  
    common data providers, 252, 375  
    components of, 244  
    differences between ADO and DAO, 452  
ActiveX Data Objects, with XML, 1046  
AdAsyncExecute, 421  
AdAsyncFetch, 421  
AdAsyncFetchNonBlocking, 421  
AdCmdFile, 420  
AdCmdStoredProc, 420, 466  
AdCmdTable, 385, 399, 420, 466

AdCmdTableDirect, 422  
AdCmdText, 384, 422  
AdCmdUnknown, 385, 421  
Add method, 72, 180, 336, 456, 533, 534, 893, 1027, 1066  
ADD USER statement, 620  
Add Watch dialog box, 209–212  
AddNew method, 319, 322, 329, 448, 449, 483, 486, 738  
.ade file format, 237  
AdExecuteNoRecords, 422, 474  
AdExecuteRecord, 423  
AdExecuteStream, 422  
AdKeyForeign, 412  
AdLockBatchOptimistic, 418, 488  
AdLockOptimistic, 382, 399, 418  
AdLockPessimistic, 418  
AdLockReadOnly, 384, 418, 468  
ADO. *See* ActiveX® Data Objects (ADO)  
ADO classic *vs.* ADO.NET, 245  
ADO Recordsets, 414–447  
    asynchronous fetching, 421–423  
    bookmarks, using, 442–445  
    cursor location, 418–419  
    cursor types, 416–417  
    cursors, 416–417  
    finding record based on multiple conditions, 441–442  
    finding record position, 433–434  
    finding records using find method, 438–439  
    finding records using seek method, 439–441  
    GetRows method to fill the recordset, using, 446–447  
    lock types, 417–418  
    moving around in Recordset, 432–433  
    opening Recordset, 423–432  
        based on criteria, 430–431  
        based on SQL Statement, 429–430  
        based on table or query, 424–428  
        directly, 431–432  
    Options parameter, 419–421  
    reading data from field, 434–435  
    returning Recordset as string, 435–438  
ADO schema, 1050  
ADODB (ActiveX Data Objects), 244  
AdOpenDynamic, 416, 428, 468  
AdOpenForwardOnly, 416, 443, 468  
AdOpenKeyset, 399, 416, 417, 443  
AdOpenStatic, 382, 384, 417, 428, 467, 468, 488  
AdOptionUnspecified, 423  
ADOX (ADO Extensions for DDL and Security), 244  
ADOX Object Model, 462  
.adp file format, 236–237  
AdSchemaColumns, 402  
AdSchemaProviderTypes, 403  
AdSortAscending, 410  
AdSortDescending, 410  
Advanced ADO/DAO features  
    cloning recordset, 506–512  
    creating custom recordset (ADO), 483–486  
    data shaping, 516–535  
        creating shaped recordset (ADO), 516–520  
        with other databases, 516  
    shaped recordsets with grandchildren, 524  
    shaped recordsets with multiple children (ADO), 520–524  
    working with, 515–519  
writing complex SHAPE statement, 520  
writing simple SHAPE statement, 513–514  
disconnected recordset (ADO), 486–488  
displaying current and previous records by using Clone method, 507–512  
fabricating recordset, 482–486  
filling combo box with disconnected recordset (ADO), 491–493  
hierarchical recordsets  
    creating form with TreeView control, 524–526  
    writing event procedure for form load event, 526–535  
saving records to disk (ADO), 489–506  
taking persisted data on road  
    creating unbound Access form to view and modify data, 493–494

- saving recordset to disk, 493
- viewing and editing data offline, 501–502
- writing procedures to control form and data, 495–501
- transaction processing, 535–538
  - creating transaction with ADO/DAO, 536–538
- Advanced Data TableGram, 1046
- Advanced event programming
  - declaring and raising events, 914–919
  - responding to control events, 910–914
  - sinking events in standalone class module, 900–909
  - cRecordLogger class, creating, 902–905
  - cRecordLogger custom class with another form, 907–908
  - file preparation, 901–902
  - instance of custom class, creating, 905–907
  - Name property, 902, 911, 916, 916, 916, 917, 943
  - Object drop-down list, 903, 918
  - Procedure drop-down list, 903
  - testing cRecordLogger custom class, 907–908
- writing event procedure code, 910
- AfterDelConfirm event, 703–704
- AfterInsert event, 696
- AfterUpdate event, 699–700
- AfterUpdate event (control), 733–735
- ALL keyword, 342
- ALTER COLUMN clause, 555
- ALTER DATABASE PASSWORD statement, 614, 615
- ALTER TABLE statement, 553–556
- ALTER USER statement, 618
- American National Standards Institute (ANSI), 544
- AND operator, 116, 117, 341
- ANSI. *See* American National Standards Institute (ANSI)
- ANSI SQL-89, 544
- ANSI SQL-92 or SQL-2, 544
- ANSI SQL query modes, 544
  - setting, 544–545
- Append method, 282, 301, 386, 408, 412, 482, 485, 643, 646
- Append Only memo fields, 290–296
- Append queries, 352–353
- Application-defined property, 278
- Application Programming Interfaces (API), 1078
- ApplyFilter event, 718–719
- ApplyTransform function, 996
- Arguments, 4, 74, 88
  - optional, 92–93
  - passing arguments by reference and value, 91
- Array function, 163–170
- Array variable, 151
- Array(s), 148–154
  - Array function, 152–153
  - Debug button, 170
  - declaring, 150–151
  - dimensioning, 163
  - dynamic, 161–163
  - Erase function, 165–166
  - errors in, 170–173
  - fixed-dimension, 161
  - functions, 163–170
  - initial value of array element, 159
  - initializing and filling, 152
    - Array function, 152–153
    - For...Next loop, 153–154
    - individual assignment statements, 152
  - IsArray function, 164–165
  - LBound and UBound functions, 169–170
  - looping statements, 156–159
    - For Each...Next statement, 142–143
    - For...Next statement, 139–142, 176
    - loops in real life and, 158–159
    - passing elements of array to another procedure, 157–159
  - one-dimensional array, 148, 151, 152, 154–155, 160, 170, 295
  - Option Base 1 statement, 148, 151, 154, 157, 189
  - parameter, 168–169
  - passing arrays between procedures, 159
  - passing arrays to function procedures, 174–175

- range of, 155
- sorting, 175–177
- static, 161–163
- two-dimensional array, 160–161
- upper and lower bounds, 169–170
- ArrayString**, 295
- Assert statement**, 208–209
- Asterisk (\*)**, 315
- Async property**, 1006
- Asynchronous record fetching**, 421
- Attachment data type**, 272, 277, 288
- Attachment fields**, 288–290
- Attachments control**, in Access forms, 675–689
  - Attachments dialog box, 685
  - Current File text box, 688
  - Forward and Backward buttons, 685
  - AttachmentCurrent event procedure, 687
  - unbound text box, 681
- Attribute-based document**, 984
- Attribute-centric XML**, 1049, 1054–1058
- Authentication credentials**, 1082
- AutoExec macro**, 928–932
  - contents of, 928
  - macro actions, arguments, and program flow, 929–932
- OpenForm**, 932
- SetDisplayedCategories**, 931
- AutoNumber**, 404–405
  
- B**
- Backstage View**, 855–859
  - customizing, 859
  - hiding buttons and tabs, 859
- BeforeDelConfirm event**, 702–703
- BeforeInsert event**, 695
- BeforeUpdate event**, 696–699
- BeforeUpdate event (control)**, 732–733
- Beginning of file (BOF)**, 307
- BeginTrans method**, 364, 336, 536
- BETWEEN...AND operator**, 341
- BinaryCompare**, 1068
- Bookmark property**, 307, 315, 443
- Bookmarks**, 442–445
- Boolean expressions**, 112
- Break mode**, 194, 199, 204
- Breakpoints**, 200–205
- Bubble sort**, 175
- Built-in functions**, 43, 44, 94–107
- ByRef keyword**, 91
- ByVal keyword**, 90, 91, 174
  
- C**
- Cache-Control header**, 1082
- Calculated field**, 282–285
- Call Stack dialog box**, 215–216
- Callback procedures**, 808, 811, 819, 830, 833, 849, 850
- CancelBatch method**, 505
- CancelUpdate method**, 329
- Cascading Stylesheets (CSS)**, 1056
- Case Else clause**, 125
- Catalog**, 385
- Categorized tab**, 30
- Category argument**, 795, 931
- CData directive**, 995
- CDate function**, 120, 296
- ChangePassword method**, 629, 644, 669
- CHECK constraints**, 568–574
- ChildNodes property**, 1040–1041
- Class**, 866, 867–878
  - Class methods, 874–878
  - Class modules, 13–14, 173, 178–188, 865–867
    - form, 13
    - naming, 868
    - report, 13
    - standalone, 13
- Click event**, 15, 497, 499, 500, 691, 692, 706, 707, 724, 727, 737, 741, 784, 786, 918, 942, 645
- Click event (control)**, 737–742
- Client-side cursor (adUseClient)**, 418
- Client-side script**, 943, 947
- Clone method**, 523, 524, 560
- Cloning recordset**, 523–529
- Close event**, 767
- Close method**, 260, 268, 283, 389, 437, 444

- Code window  
  activate, 32  
  Break mode, 206–207  
  splitting, 33  
  understanding, 32–34
- Collection(s) *See also Specific collections*  
  custom (*See Custom collection*)  
  own, 180–188
- Collection class, 894
- Collection object, 866, 893
- ColumnHistory method, 293, 295
- Combo box, 15, 201, 844–846
- Command Button Wizard, 948
- Command object, 415, 420, 424, 462, 466, 471, 604
- CommandBars object and Ribbon, 852–854
- CommandText property, 423, 464, 466
- CommandType property, 420, 466
- CommitTrans method, 364, 536
- CompactDatabase method, 266
- Compacting database  
  CompactDatabase (DBEngine object), 266  
  CompactDatabase (Microsoft Jet and  
    Replication Objects (JRO)  
    Library), 244,
- CompareMode property, 1066, 1068
- Concatenation, 58
- Conditional expression  
  If...Then statement, 112–114  
  If...Then...else statement, 118–118  
  If...Then...Elseif statement, 121–122  
  logical operators, 111–112  
  nested if...then statements, 122–125  
  relational operators, 111–112  
  Select Case statement, 125–131  
    specifying multiple expressions in Case  
    clause, 130–131  
    specifying range of values in Case clause,  
      128–129  
    using Is with Case clause, 128
- Conditional statements, 111
- Connection strings, 246–248
- ODBC  
  creating and using DSN-less ODBC  
  connections, 254–255
- creating and using ODBC DSN  
  connections, 248–253
- data sources, 248
- OLE DB, 255–256
- via data link file, 256–257
- Constants in VBA procedures, 81–83  
  declaring, 81  
  intrinsic, 82  
  Private constant, 81  
  Public constant, 81
- Constraint clause, 557, 561, 574, 582, 585
- Constraints, 541, 546, 553, 567  
  CHECK, 568–574  
  FOREIGN KEY, 567, 574  
  NOT NULL, 567  
  PRIMARY KEY, 567, 568  
  UNIQUE, 567
- Container objects, 268, 660
- Content-type header, 1006, 1082
- CopyFromRecordset method, 332, 333, 335, 336
- Copying database  
  with DAO, 371  
  with FileSystemObject, 371–372
- Counter, 139
- Count property, 1066
- CreateAdditionalData method, 1027
- CreateDOM function, 1005
- CreateField method, 275, 276, 282, 302, 304, 594
- CreateIndex method, 301, 304
- CREATE INDEX statement, 583
- CreateObject function, 1005, 1081
- CreateObject function, 335, 371, 379, 437, 457, 1081
- Create Object method, 1064
- CREATE PROCEDURE (or CREATE PROC)  
  statement, 599
- CreateProperty method, 279
- CreateQueryDef method, 344
- CreateReport method, 748
- CreateTableDef method, 276, 301
- CREATE TABLE statement, 546, 547, 553, 574, 582, 585
- CreateTextFile method, 379, 437, 457

- CREATE USER statement, 617  
CREATE VIEW statement, 594  
CSS, 1056  
Current event, 693–694  
CurrentDb method, 276, 378  
CurrentView property names and values, 752  
Cursor, 416–417  
CursorPosition parameter, 418–419  
CursorType parameter, 416–417  
CursorType property, 399  
Custom application, running, 889–890  
Custom collection  
    adding objects to, 180  
    creating, 184  
    removing, 182  
Custom data entry form, 730  
Custom objects, 867–878  
    creating class methods, 868–869  
    creating class module, 179  
    defining properties for class, 870–874  
    event procedures in class module, 877–878  
    instance of class, creating, 875–876  
    naming class module, 868  
    variable declarations, 869–870
- D**
- DAO. *See* Data Access Objects (DAO)  
DAO Recordsets, 305–318  
    finding *n*th record in Snapshot, 317–318  
    finding records in Dynasets or Snapshots, 315–317  
    finding records in Table-type recordset, 313–315  
    moving between records, 312–313  
    navigating through recordset, 307  
    opening Snapshot and counting records, 309–311  
    opening Table-, Dynaset-, and Snapshot-type Recordsets, 308–309  
    retrieving contents of specific field, 311–312  
        types, 306  
Data Access Objects (DAO), 476  
Data Access technologies  
    Access versions and formats, 235–238  
    file formats supported in Access  
        2007–2021, 235–238  
connection to current Access database, 378–380  
copying database  
    with DAO, 266–267  
    with FileSystemObject, 371–372  
creating a reference to ADO library, 245–246  
creating new Access database  
    with ADO, 369–372  
    with DAO, 265–266  
database engines: JET/ACE, 233–234  
library references, 238–241  
Microsoft Access databases, opening  
    in read-only mode with ADO, 273  
    in read-only mode with DAO, 270  
    in read/write mode with ADO, 270–272  
    in read/write mode with DAO, 267  
    secured with password, 274–275  
    with user-level security, 279–280  
Object libraries, 241–245  
    Microsoft Access 16.0 Object Library, 241  
    Microsoft ActiveX Data Objects 6.1  
        Library (ADO), 243–244  
    Microsoft DAO 3.6 Object Library, 241–242  
    Microsoft Office 16.0 Access Database Engine Object Library, 241  
    VBA object library, 241  
opening databases, spreadsheets  
    and text files  
    connecting to SQL server database, 380  
    opening Microsoft Excel workbook, 380–381  
DatabaseCompare, 1068  
Data Definition Language (DDL), 475, 545, 613, 627  
Data Definition Language (DDL) queries, 927  
Data Definition Query window, 577–580  
Data events, 692–704  
    AfterDelConfirm, 703–704  
    AfterInsert, 696  
    AfterUpdate, 697–699  
    BeforeDelConfirm, 702–703

- BeforeInsert, 695
- BeforeUpdate, 696–697
- Current, 693–694
- Delete, 701–702
- Dirty, 700
- OnUndo, 700–701
- Data Link Properties dialog box, 256, 259
  - Advanced tab, 257
  - Connection tab, 257
  - Provider tab, 257
- Data macros, 948–964
  - copying, 964
  - creating, 950–959
  - execution errors, 962–963
  - existing named, 960
  - named data macro, 957, 960
  - ReturnVars, using, 960–962
  - using, 948–949
- Data Manipulation Language (DML), 340, 474, 475, 546, 600
- Data members, 869, 873
- Data providers, 243, 252, 375, 421
- Dataroot element, 988–989
- Data shaping, 512–535
  - creating shaped recordset (ADO), 516–519
  - with other databases, 516
  - shaped recordsets with grandchildren, 524
  - shaped recordsets with multiple children (ADO), 520
  - working with, 515–519
  - writing complex SHAPE statement, 520–535
  - writing simple SHAPE statement, 513–514
- Data type(s), 51–54
  - ADO vs. Microsoft Access data types, 387–388
  - converting, 107–109
  - listing, 399–400
  - user-defined, 53
  - variant, 53
- Database engines
  - JET/ACE, 233–234
  - versions, 233
- Database errors
  - On Error GoTo 0, 373
  - On Error GoTo Label, 372
- On Error Resume Next, 372
- VBA Err object and ADO Errors collection, 373–375
- Database security
  - adding users to groups, 620–621
  - changing user password, 618–619
  - creating group account, 619–620
  - creating user account, 616–618
  - deleting group account, 626–627
  - deleting user account, 668–669
  - granting permissions for object, 623–624
  - removing database password, 661–662
  - removing user account from group, 615–616
  - revoking security permissions, 625–626
  - setting database password, 614–615
- Database security, implementing
  - opening secured MDB database, 640–643
  - securing Access MDB database, 634–640
  - share-level security, 630–631
- user and group accounts (ADO)
  - creating, 643–647
  - deleting, 647–649
  - listing, 649–650
  - listing users in groups, 650–652
- user and group permissions
  - changing user password, 669–671
  - checking permissions for objects, 663–665
  - object owner, retrieving name of, 652–655
  - setting database password using
    - CompactDatabase method, 665–667
  - setting database password using
    - NewPassword method, 667–668
  - setting permissions for containers, 660–662
  - setting permissions for database, 658–660
    - setting permissions for object, 655–658
- user-level security, 630–631
- workgroup information file, 279, 631–640
  - Access versions, 631
  - creating and joining, 633–640
- DblClick (control), 742–745
- DblClick event, 707, 721, 742
- DblClick (Form section event), 721–722

- DDL. *See* Data Definition Language (DDL)
- Deactivate event, 705, 752
- Debug button, 170
- Debugging, 195
- Default Value property, 278, 563
- Delete event, 701–702
- Delete method, 299, 329, 331, 360, 389, 392, 452, 480, 1079
- Delete query, 353–356
- DelimFound function, 727
- Dirty event, 700
- DISALLOW NULL option, 587, 588
- Disconnected recordsets, 486–488  
    creating, 487–488
- DISTINCT keyword, 342, 343
- DISTINCTROW keyword, 343
- Dictionary object, 1063, 1066, 1069
- Document Object Model (DOM), 1005, 1037
- Document Type Definition (DTD), 982
- DOM (Document Object Model), 996, 1005
- DOMDocument object, 996, 1005
- DomDocument60, 1039
- Do...While statement, 134–137
- DROP COLUMN clause, 556, 559
- DROP CONSTRAINT clause, 559
- DROP GROUP statement, 626
- DROP INDEX statement, 591
- DROP PROCEDURE (or DROP PROC)  
    statement, 606
- DROP USER statement, 621
- DROP VIEW statement, 621
- DSN (Data Source Name), 248–256  
    File, 249  
    System, 249  
    User, 249
- Dynamic array, 161–163
- Dynamic link library (DLL), 234, 982
- Dynamic-type Recordset, 307
- Dynaset-type Recordset, 320, 329
- E**
- Early binding, 1064
- Edit method, 322, 327, 329, 452
- EditModeEnum constants, 329
- Element-based document, 984
- Element-centric XML, 1049–1050
- ElseIf clause, 121
- Embedded macros  
    copying, 942–943  
    creating, 941–942
- End of file (EOF), 253, 274, 303, 330, 331, 428, 511
- Enter event (control), 730–732
- Erase function, 165–169
- Err object, 222–226
- Error, mistake and, 222
- Error event, 715–716  
    DataErr, 715  
    Response, 715
- Error handler, 208, 222, 228, 344, 369, 372, 378, 407, 408, 551
- Error trapping, 223–224  
    Err object, using, 222–223  
    On Error statement, 222,  
        procedure testing, 226–228  
        setting options in visual basic project, 228–229
- Event, 867
- Event data macros, 949
- Event-driven programming, advanced  
    concepts in  
        declaring and raising events, 914–919  
        responding to control events, 911–914  
        sinking events in standalone class module, 900–910  
            creating cRecordLogger class, 902–905  
            creating instance of custom class, 905–907  
            file preparation, 901–902  
            testing cRecordLogger custom class, 907–908  
            using cRecordLogger custom class with another form, 908–909  
            writing event procedure code, 910
- Event handler. *See* Event procedures
- Event procedures, 3, 6, 13, 15–22, 72, 497, 500, 692, 697, 702, 704, 705, 711, 721, 731, 745, 771, 782, 866, 867, 877–878, 900, 910

- compiling, 22–23
- writing, 17–18
- Event properties, 6, 15–22, 691
- Event sink, 900
- Event source, 900
- Event statement, 900, 915
- Event trapping, 15, 692
- Event(s), 15, 16, 691–746, 867
  - AfterUpdate (control), 733–735
  - BeforeUpdate (control), 732–733
  - Click (control), 752–758
  - data, 692–704
    - AfterDelConfirm, 703–704
    - AfterInsert, 696
    - AfterUpdate, 697–699
    - BeforeDelConfirm, 702–703
    - BeforeInsert, 695
    - BeforeUpdate, 696–697
    - Current, 693–694
    - Delete, 701–702
    - Dirty, 700
    - OnUndo, 700–701
  - DblClick (control), 742–745
  - Enter (control), 730–732
  - error, 715–716
    - DataErr, 715
    - Response, 715
  - filter, 716–719
    - ApplyFilter, 718–719
    - Filter, 716–717
  - focus, 704–706
    - Activate, 704–705
    - Deactivate, 705
    - GotFocus, 705
    - LostFocus, 706
  - form section, 721–722
    - DblClick, 721
  - keyboard, 710–714
    - KeyDown, 710–711
    - KeyPress, 711–713
    - KeyUp, 713–714
  - mouse, 706–709
    - Click, 706–707
    - DblClick, 707
    - MouseDown, 707–709
- MouseMove, 709
- MouseUp, 709
- MouseWheel, 709
- NotInList (control), 735–737
- OpenArgs property, 723–728
- sequence of, 692
- timing, 719–721
  - Timer, 720
- Exclamation point (!), 52, 328, 342
- Execute method, 350, 352, 353, 354, 367, 388, 421, 422, 423, 429, 462, 466, 473, 551, 571, 598, 600, 604, 605, 1075, 1077
- ExecuteMso method, 853
- Exists method, 1066
- Exiting loops early, 135, 143–144
- Exiting procedures, 144
- Explicit variable declaration, 55
  - advantages of, 55
- ExportNavigationPane method, 798
- ExportXML method, 1022–1024, 1022–1027
  - arguments of, 1024–1025
- Expression Builder, 935–936
- Extensible Markup Language (XML), 981
- Extensible Markup Language (XML), 981–1060
- Extensible Stylesheet Language (XSL), 992

## F

- Fabricating recordset, 482–486
- Fast commands, 855
- Field Properties Lookup tab, 288
- FileDateTime function, 486
- FileFormat parameter, 336
- FileLen function, 485, 486
- Filter events, 716–719
  - ApplyFilter, 718–719
  - Filter, 716–717
- Filter property, 338–339, 768
- Find methods, 313, 315, 327, 329
- Fixed-dimension arrays, 161
- Focus events, 704–706
  - Activate, 704–705
  - Deactivate, 705

GotFocus, 705  
 LostFocus, 706  
 For Each...Next loop statement, 73, 142–143, 156–157  
 Foreign key, 412  
 Foreign key constraint, 574  
 Form module, 13, 526, 866, 910, 916  
 Form section event, 721–722  
 Format event (Report Section Event), 756–759  
     Cancel, 756  
     effect on report sections, 756  
     FormatCount, 756  
 For...Next loop statement, 141, 153–154, 159, 163, 188, 211, 295, 727  
 Forward-only-type Recordset, 306  
 Friendly HTTP error messages, turning off, 1106–1108  
 Function procedures, 3, 5, 86–88  
     methods of running  
         from Immediate window, 87  
         from subroutine, 87–88  
     passing arguments to, 90–91  
     data types and functions, 88–90  
 Function procedures (functions), 5–6  
 Functions, 85–109  
     built-in functions, 94  
     InputBox function, 104–107  
     IsMissing function, 93  
     MsgBox function, 94–107  
         formatting, 95–98  
         MsgBox buttons argument settings, 98–99  
         prompt argument, 95  
         returning values from, 103–104  
         using functions with arguments, 101–102  
     passing arguments to function procedures  
         by reference and value, 90–91  
         specifying data types, 88–90  
         using optional arguments, 92–93  
     running function procedure  
         from Immediate window, 86–87  
         from subroutine, 87–88

**G**

Galleries, 801  
 GetAllResponseHeaders, 1082–1083  
 GetElementsByTagName method, 1042  
 GetEnabledMso method, 853  
 GetImageMso method, 853  
 GET method, 1079, 1082  
 GetObject function, 335  
 GetObjectOwner method, 653  
 GetPermissions method, 663  
 Get request, 1085  
 GetResponseHeader, 1082–1083  
 GetRows method, 446–447  
 GetString method, 435, 437, 454, 457, 466, 468, 480, 488  
 Global property, 1075, 1095  
 Global variable, 70, 71, 72, 73, 74, 214, 758  
 GotFocus event, 19, 691, 705  
 GRANT statement, 623  
 Group argument, 796  
 Group auto-scaling, 854–855  
 GUIDs, 656

**H**

HasChildNodes method, 1040–1041  
 Header, 1082, 1084  
 HTML tags, 994  
 HTTP GET protocol, 1006  
 HyperText Markup Language (HTML), 982  
 Hyphen (-), 342

**I**

If block instructions, 118  
 If...Then statement, 112–114  
     formats of, 115–116  
     multiline, 114–115  
     with AND operator, 117–118  
 If...Then...else statement, 119–120  
 If...Then...Elseif statement, 121–122  
     ElseIf clause, 121  
 IgnoreCase property, 1075  
 IGNORE NULL option, 587, 589–590  
 IgnoreNulls property, 302

IIS (Internet Information Services), 1007  
Immediate window  
  in break mode, 205–207  
Implicit variable declaration, 55  
  disadvantages of, 55  
ImportNavigationPane method, 798  
ImportXML method, 1022–1023, 1036, 1087  
ImportXML method, arguments of,  
  1036–1037  
IN operator, 341  
Indexed Sequential Access Method  
  (ISAM), 234  
Index(es)  
  adding index to existing table, 583–585  
  adding multiple-field index to existing table,  
    304–305  
  creating indexed with restrictions,  
    587–590  
  DISALLOW NULL option, 588–589,  
  IGNORE NULL option, 589–590  
  PRIMARY option, 587,  
  creating indexes using ADO, 406–408  
  creating indexes using DAO, 352–354  
  creating primary key, 406  
  creating single field index using ADO,  
    408–410  
  creating table with primary key,  
    585–586  
  creating tables with indexes, 582–583  
  deleting indexes, 591  
  deleting table indexes (ADO),  
    411–412  
  listing indexes in table (ADO), 410  
IndexNulls property, 408  
Infinite loop, 137  
Informal (implicit) variables, 56  
InputBox function, 75, 87, 89, 94, 104–107  
InputBox method, 318, 348  
Instance, 866, 867  
InStr function, 295, 296, 325, 727, 905, 1070  
Internet Information Services (IIS), 1007,  
  1099–1108  
Intrinsic constants, 82–83  
InvalidateControl method, 849, 851  
Invalidate method, 852

IRibbonControl properties, 811–813  
IRibbonUI object, 849–852, 854  
  Invalidate method, 852  
  InvalidateControl method, 849, 851  
IsArray function, 164–165  
IsMissing function, 93  
Is Nothing expression, 1045  
IS NULL operator, 341  
Item property, 1066  
Items method, 1066  
IXMLDOMNodeList method, 1042 1045  
IXMLDOMNode object, 1040–1041

## J

JavaScript, 997, 1088  
JavaScript Object Notation (see JSON)  
Jet. See Microsoft Jet (Joint Engine Technology  
  (JET)), 233  
JRO (Jet and Replication Objects), 24  
JSON, 1088–1097  
JSONP, 1091

## K

KeyAscii, 912  
Keyboard events, 710–714  
  KeyDown, 710–711  
  KeyPress, 711–713  
  KeyUp, 713–714  
KeyCode, 710, 713  
KeyDown event, 710–711  
  KeyCode, 710  
  Shift, 710  
KeyPress, 711–713  
Key property, 1066  
Keys method, 1066  
KeyUp event, 713–714  
  KeyCode, 713  
  Shift, 713  
Keywords, 4, 5, 6, 20, 40, 45, 56, 91, 118, 125,  
  126, 130, 150, 170, 205, 211, 217, 226,  
  340, 343, 364, 536, 557, 575, 585, 871,  
  873  
Kill statement, 1049

**L**

.laccdb file format, 238  
 Late binding, 1064  
 Layout view, 122, 141, 675, 676, 680, 681, 682, 747, 748, 752, 764, 765, 770, 822, 939, 941,  
 LBound function, 169–170  
 .ldb file format, 237  
 Left function, 296, 1070  
 Len function, 296  
 Library, 43  
 Library references, 238–241  
     default object libraries, 239  
     missing library, 239  
     References dialog box, 240–241  
 Lifetime of variables, 71  
 LIKE operator, 341  
 List Properties/Methods, 20  
 Load method, 1006  
 LoadXML method, 1086  
 Localhost, 1007  
 Locals Window, 206, 214–215  
 Local variables. See Procedure-level (local) variables  
 Location index, 304  
 LockType property, 399, 418  
 Logic errors, 199, 208  
 Logical operators, 111–112  
 Loop, 133  
     infinite, 137  
 Looping, 131  
 Looping statements  
     Do...Until statement, 137–138  
     Do...While statement, 134–136  
     For Each...Next statement, 142–143  
     exiting loops early, 143–144  
     For...Next statement, 139–142  
     infinite loops, avoiding, 137  
     nested loops, 144–145  
     paired statements, 142  
     variables and loops, 138  
 LostFocus event, 722

**M**

Macro security (Access 2021), 924–928  
 Macro(s), 923–978  
     Access 2021 macro security, 924–928  
     AutoExec macro, 928–932  
         contents of, 928  
         macro actions, arguments, and program flow, 929–931  
     OpenForm, 932  
     SetDisplayedCategories, 931  
     converting macros to VBA code, 969–972  
     data macros, 948–949  
     embedded macros, creating, 941–942  
     error handling in, 965–967  
     generating macros using Command Button Wizard, 948  
     Info tab, 926  
     Macro Settings options, 925  
     Microsoft Office Security Options dialog box, 927  
     standalone macros, creating, 933–935  
     submacros, creating, 940–941  
     temporary variables in, 968–969  
     VBA and, 969  
 Make-Table query, 348–349  
 .mdb file format, 236, 247, 252, 611, 614, 617, 630, 633, 644, 667, 948  
 .mdw file format, 237  
 Method, 874–875  
 Microsoft Access 16.0 Object Library, 238  
 Microsoft Access database  
     connection to current Access database, 378–380  
     copying database  
         with DAO, 266–267  
         with FileSystemObject, 371  
     creating new Access database  
         with ADO, 369–370  
         with DAO, 265–266  
     opening database  
         in read-only mode with DAO, 270  
         in read/write mode with ADO, 377–378  
         Microsoft jet read/write mode with ADO, 375–379  
         secured with password, 270–271

- with user-level security, 267, 614
- Microsoft Access database field
  - creating append only memo fields with DAO, 290–296
  - creating attachment fields with DAO, 288–290
  - creating calculated fields with DAO, 284–285
  - creating multivalue lookup fields with DAO, 285–288
  - creating rich text memo fields with DAO, 296–299
  - listing fields, 394
  - removing field from table (ADO/DAO), 392–393
  - retrieving field properties, 394–395
- Microsoft Access database table
  - adding new fields to existing tables (ADO/DAO), 391–392
  - AutoNumber, changing value of, 404–405
  - copying table (ADO), 388–389
  - creating table (DDL/DAO), 546–549
  - deleting table (ADO), 389–390
  - linking Access table, 396–397
  - linking dBASE table, 301
  - linking Excel worksheet, 397–399
  - listing database tables, 399–400
  - listing tables, 400–403
  - retrieving table properties, 300
- Microsoft Access Jet/ACE database engine, 233–234
  - opening databases, spreadsheets and text files, 380–385
  - connecting to SQL server database, 380
  - opening Microsoft Excel workbook, 380–381
  - opening text file using ADO, 383–385
- Microsoft Access tables
  - indexes
    - adding multiple-field index to existing table (DAO), 303–305
    - creating indexes using ADO, 406–408
    - creating indexes using DAO, 301–303
    - creating primary key, 406
    - creating single field index using ADO, 408–410
  - deleting table indexes (ADO), 411–412
  - listing indexes in table (ADO), 410
- primary keys, 406–408
  - table relationships, using ADO
    - one-to-many relationship, 412–413
    - parent-child relationship, 412
- Microsoft ActiveX Data Objects 6.1 Library (ADO), 243–245
  - ADO classic vs. ADO.NET, 245
  - components of, 244
  - creating reference to, 245–246
  - data providers, 252–253
- Microsoft DAO 3.6 Object Library, 241–242
  - Errors collection, 242
  - Parameters collection, 242
  - Properties collection, 242
  - Recordsets collection, 242
  - Workspaces collection, 242
- Microsoft Jet (Joint Engine Technology (JET)), 233–234
- Microsoft Jet or Jet database engine, 233–234
- Microsoft Office 16.0 Access Database Engine Object Library, 241
- Microsoft XML Object Library, 1080
- Mid function, 296, 1012, 1070
- Module-level variables, 67–68
- Module(s), 3, 174, 866
  - class, 13–14, 866
  - form, 13, 866
  - renaming, 36
  - report, 13–14
  - standalone, 13
  - standard, 7–12
    - executing procedures and functions, 10–13
    - writing procedures in, 7–9
- Mouse events, 706–709
  - Click, 706–707
  - DblClick, 707
  - MouseDown, 707–708
  - MouseMove, 709
  - MouseUp, 709
  - MouseWheel, 709
- MouseDown event, 707–708
- MouseWheel event, 709
- Move methods, 312, 313, 449

MoveFirst method, 318, 449  
 MoveLast method, 306, 309, 311, 318, 348, 434, 439  
 MoveNext method, 306, 307, 329, 428, 452  
 MovePrevious method, 306–307  
 MsgBox function, 4, 62, 87, 91, 94–104  
     formatting, 95–96  
     MsgBox buttons argument settings, 98–99  
     prompt argument, 95  
     returning values from, 103–104  
     syntax of, 95  
     using functions with arguments, 101–103  
     using parentheses, 104  
 MSXML parser, 996  
 Multiline If...Then statement, 114–116  
 Multiline property, 1075  
 Multivalue lookup fields, 285–288  
     adding values to, 323–326  
     creating, 285–288  
     data types, 285–286

## N

Name property, 36, 386  
 Named data macros  
     creating, 957–959  
     editing, 960  
     running, 960  
 Namespace, 988–989  
 NavigateTo method, 795  
 Navigating with bookmarks, 220  
 Navigation pane, 791–795  
     in Access 2021, 791  
     adding custom group, 793–794  
     assigning objects to custom groups in, 794–795  
     with custom groupings, 795  
     customizing, 795–800  
         controlling display of database objects, 795–796  
         locking Navigation pane, 795  
         saving and loading configuration of, 798  
         setting displayed categories, 797–798  
 Grouping options, 791  
 Navigation Options dialog box, 792

Search Bar or navigation options, 791  
 system objects in, 818–819  
 Navigator control, 675  
 Nested if...then statements, 122–125  
 Nested loops, 144–146  
 NewPassword method, 667  
 NoData event, 752  
 NodeType property, 1040–1041  
 Non-row-returning queries, 475  
     Action queries, 475  
     DDL queries, 475  
 NOT NULL constraint, 567  
 NOT operator, 112  
 NotInList event (control), 735–737  
     NewData, 735  
     Response, 735  
 Number sign (#), 52, 53, 342  
 NZ function, 1048

## O

Object Browser window  
     intrinsic constants, 82  
 Object libraries, 241–245  
     Microsoft Access 16.0 Object Library, 241  
     Microsoft ActiveX Data Objects 6.1 Library (ADO), 243–244  
     Microsoft DAO 3.6 Object Library, 241–242  
     Microsoft Office 16.0 Access Database Engine Object Library, 241  
     VBA object library, 241  
 Object variables in VBA procedures, 76–79  
     advantages of, 78  
     disposing of, 79  
 ODBC Data Source Administrator, 248–255  
     File DSN, 249  
     System DSN, 249  
     User DSN, 248  
 OLE DB, 243  
 One-to-many relationship, between tables, 412–413  
 OnError action arguments, 965  
 On Error GoTo statement, 755  
 OnUndo event, 700–701

- OOP (Object Oriented Programming), 865  
OpenArgs property, 722–728  
    of Report object, 767–769  
OpenCurrentDatabase method, 1027  
Open event, 748–749  
OpenForm method, 38, 723  
    parameters, 723  
Open method, 252, 253, 259, 375, 376, 382, 384,  
    414, 415, 417, 419, 421, 424, 428, 430,  
    431, 467, 485, 506, 1006, 1058, 1082,  
    1084  
OpenQuery method, 595  
OpenRecordset method, 274, 275, 306, 307, 309,  
    319, 326, 329, 363, 595  
OpenReport method, 767  
OpenSchema method, 401, 402, 403  
Option Base 1 statement, 148, 151, 154,  
    155, 189  
Option Explicit statement, 9, 64, 66, 757,  
    903, 1027  
Option Private Module statement, 70–71  
Optional arguments, 92–93  
Options parameter, 419–421  
OR operator, 116, 655  
ORDER BY clause, 343, 460, 593, 597, 599  
Own item collections, 180–188  
    adding items to, 180–181  
    accessing, 182  
    creating, 184–186  
    determine 181  
    removing items from, 182–183  
    returning, 186  
    updating, 183  
    keeping track of multiple values using,  
        147–177
- P**
- Page event, 753  
Paired statements, 142  
ParamArray keyword, 173–174  
Parameter query, 346–348  
    creating Parameter query with ADO,  
        470–476  
    creating Parameter query with DAO,  
        346–347  
executing Parameter query with ADO,  
    473–474  
Parameterized stored procedures  
    creating, 600–603  
    executing, 604–606  
Parent-child relationship, between tables,  
    412, 513  
ParseError object, 997  
Parser, 982  
Pass-Through query, 356–358  
Passing arguments  
    ByRef and ByVal, 91  
    optional arguments, using, 92–93  
    by reference and value, 90–91  
    specifying data types, 88–89  
    subroutines and functions, 89, 90–91  
PatternMatch, 1076  
Pattern property, 1075  
.pdf file format, saving reports in, 766  
PercentPosition property, 310  
Percent sign (%), 342  
POST method, 1079, 1082  
Predicate, 342  
PRIMARY KEY constraint, 567  
Primary keys, 405, 541, 581  
PRIMARY option, 587  
Print event (Report Section Event), 759–762  
    effect on report sections, 760  
    PrintCount, 759  
Private constant, 81  
Private keyword, 869  
Procedure-level (local) variables, 66  
Procedure testing, 226–228  
Procedure(s), 3  
    compiling, 22–23  
    execution of VBA, 189–195  
    in standard modules, 7–9  
stepping through VBA  
    running procedure to cursor, 219  
    setting Next statement, 219  
    showing Next statement, 219  
    stepping out of procedure, 219  
    stepping over, 217–218  
stopping, 199–200  
stopping and resetting VBA, 221–229  
testing VBA, 226–227

- types of, 3–7
  - event, 6
  - function, 5–6
  - property, 6–7
  - subroutine, 4–5
- writing function, 85–86
- Programs, adding repeating actions to
  - Do...Until statement, 137–139
  - Do...While statement, 134–137
  - For Each...Next statement, 142–143
    - exiting loops early, 143–144
  - For...Next statement, 139–142
  - infinite loops, avoiding, 137
  - looping statements, 133
  - nested loops, 144–145
  - paired statements, 142
  - variables and loops, 138
- Project Explorer window
  - activate, 28
  - buttons, 29
  - standard toolbar, 29
- Project-level variables, 69–70
- Prompt argument, 95
- Properties window, 30–31
- Property, 870–871
  - Property Get, 890
  - Property Let, 890
- Property procedures, 6–7
  - Defining scope of, 870–874
  - Immediate exit from, 872
  - Property Get procedure, 872–873
  - Property Let procedure, 873–874
  - Property Set procedure, 870
- Property Set, 890
- Public constant, 81
- Public keyword, 69
- PUT method, 1079, 1082
- Q**
  - Quantifier (in regular expression), 1072
- Queries,
  - Append query with DAO, running, 352–353
  - Delete query with DAO, running, 353–356
  - Make-Table query with DAO, creating and running, 348–349
  - non-row-returning, 475
  - other operations with
    - deleting query from database with DAO/ADO, 480–481
    - listing all queries in database with DAO/ADO, 480–481
    - retrieving query properties with DAO, 358–359
    - updatable query, 360–363
  - Parameter query with ADO/DAO, creating and running, 346–348, 470–473
  - Pass-Through query with ADO/DAO, creating and running, 356–358, 476–480
    - row-returning, non-parameterized, 464
    - row-returning, parameterized, 471
  - Select query manually, creating, 340–344
    - examples of queries recognized by SELECT and FROM keywords, 340
    - operators used in expressions, 340
    - predicates in SQL SELECT statements, 342–343
    - WHERE clause in SQL SELECT statements, 344
    - wildcard characters used in LIKE operator patterns, 342
  - Select query with ADO, executing, 465–468
  - Select query with ADO, modifying, 468–469
  - Select query with ADO/DAO, creating, 344–345, 462–464
  - Update query with ADO, executing, 473–475
  - Update query with DAO, creating and running, 350–352
  - Question mark (?), 342
  - Quick Access toolbar, 790, 801, 859–861
  - Quit method, 336
- R**
  - RaiseEvent statement, 915, 916
  - Range of array, 155
  - Reading data from field, 434–435
  - Record(s)

- adding attachments, 320–323
- adding new record with ADO, 448–449
- adding new record with DAO, 319–320
- adding values to multivalue lookup field, 420–423
- copying records to Excel worksheet, 332–336
- copying records to text file (ADO), 456–458
- copying records to Word document, 453–456
- deleting attachments, 331–332
- deleting record with ADO, 452–453
- deleting record with DAO, 329–331
- editing multiple records with ADO, 451–452
- filtering records using filter property (DAO and ADO), 338–339, 458–460
- filtering records using SQL WHERE clause (DAO and ADO), 337–338
- modifying record with ADO, 449–450
- modifying record with DAO, 326–329
- sorting records (ADO), 460–462
- RecordCount property, 310, 311, 318, 322, 348, 428, 434, 468, 474
- Recordset(s)
  - ADO Recordsets, 414–447
    - asynchronous fetching, 421
    - bookmarks, using, 442–443
    - bookmarks to filter recordset, 445–446
    - counting records, 428
    - cursor location, 418–419
    - cursor types, 416–417
    - finding record based on multiple conditions, 441–442
    - finding record position, 433–434
    - finding records using find method, 438–439
    - finding records using seek method, 439–440
  - GetRows method to fill recordset, 446–447
  - lock types, 417–418
  - moving around in Recordset, 432–433
  - opening Recordset, 423–424
  - opening Recordset based on criteria, 430–431
  - opening Recordset based on SQL Statement, 429–430
- opening Recordset based on table or query, 424–428
- opening Recordset directly, 431–432
- Options parameter, 419–423
- reading data from field, 434–435
- returning Recordset as string, 435–438
- DAO Recordsets, 305–318
  - finding *n*th record in Snapshot, 317–318
  - finding records in Dynasets or Snapshots, 315–316
  - finding records in Table-type recordset, 313–315
  - moving between records, 312–313
  - navigating through recordset, 307
  - opening Snapshot and counting records, 309–310
  - opening Table-, Dynaset-, and Snapshot-type Recordsets, 308–309
  - retrieving contents of specific field, 311–312
  - types, 307
- empty, 428
- Recordset objects, 242, 305
- RecordStatusEnum constants, 504–505
- REFERENCES clause, 574
- RefreshDatabaseWindow method, 549
- Relational operators, 111
- RemoveAllTempVar, 968
- RemoveTempVar, 968
- Report events, 747–786
  - Activate, 751–752
  - Close, 751
  - Deactivate, 752
  - Error, 755–756
  - NoData, 752
  - Open, 748–750
  - Page, 753
- ReportML, 985
- Report modules, 13, 866
- Report section events, 756–763
  - Format, 756–759
  - Print, 759–762
  - Retreat, 762–763
- Report view, 763–765
- Representational State Transfer (REST), 1079
- Retreat, 762–763

- Remove method, 1066
  - ReadyState property, 1084
  - RegEx (RegExp), 1071
  - RegExp object, 1073–1074
  - RegExp object, patterns, 1071
  - RegExp object, properties, 1074–1075
  - Regular expressions, 1070–071
  - RemoveAll method, 1066
  - Replace method, 1075
  - ReportML, 985
  - Required property, 278, 319, 873
  - ResponseBody property, 1007, 1085
  - ResponseStream property, 1085
  - ResponseText property, 1085–1086
  - ResponseXML property, 1085
  - Rest API, 1063
  - Restful, 1079
  - Retreat event (Report Section Event), 762–763
  - REVOKE statement, 625
  - Ribbon extensibility or RibbonX, 803
  - Ribbon programming with XML, VBA and
    - Macros, 803–826
  - Edu Systems tab, 805
  - IRibbonControl properties, 811–812
  - library references, 810
  - Ribbon customizations to forms and reports,
    - assigning, 822–826
  - Ribbon XML markup
    - creating, 804–805
    - embedding, 815
    - loading, 808–815
    - storing, 816–817
  - Show add-in user interface errors, 809
  - USysRibbons table, 816, 817, 818, 821, 823, 825, 828, 835, 839, 852, 853, 856
  - XML file, 805–806
  - Ribbon UI customizations
    - Backstage View, 855–859
    - CommandBars object and, 852–854
    - controls in
      - built-in control, 848–849
      - checkboxes, 841–842
      - combo boxes and drop downs, 844–846
      - dialog box launcher, 846–847
      - disabling control, 847–848
      - edit boxes, 843–844
    - refreshing Ribbon, 849–852
    - split buttons, menus, and submenus, 839–841
    - toggle button, 838–839
  - images in
    - attributes and callbacks, 836–837
    - requesting images via getImage callback, 831–835
    - requesting images via loadImage callback, 827–828
  - Quick Access Toolbar (QAT), 859–861
  - tab activation and group auto scaling, 854–855
  - Ribbon user interface (Access 2021), 800–803
    - contextual tab, 803
    - Create tab, 801
    - dialog box launcher button, 802
    - More Forms button, 676
  - Rich text memo fields, 296–299
  - Right function, 273, 296
  - RollbackTrans method, 536
  - Row-returning, non-parameterized queries, 464
  - Row-returning, parameterized queries, 471
  - RowSource property, 323
  - RowSourceType property, 323
  - Runtime errors, 43, 198, 221, 224, 227, 242
- ## S
- Safe expression, 948
  - Sandbox mode, 948
  - Save method, 489, 506, 1045, 1046, 1087
  - SaveAs method, 336
  - Saved (persisted) recordset, 489
  - Saving recordset to disk (ADO), 489–491
  - Schema file, importing, 1016
  - Script delimiters and HTML tags, 994, 1011
  - Scripting Dictionary, 1064
  - Scripting Runtime Library, 1063, 1065
  - Seek method, 313, 314, 326, 328, 329, 439–441
  - Select Case statement, 125–131
    - specifying multiple expressions in Case clause, 130
    - specifying range of values in Case clause, 128–130
    - using Is with Case clause, 128

- SELECT INTO statement, 348, 349, 367, 388  
Select query, 340–344  
    creating, 344–349  
    executing, 465–468  
    modifying, 468–470  
SELECT statement, 513, 514, 523, 594, 1048  
SelectNodes method, 1044  
SelectSingleNode method, 1045–1046  
Send method, 1006, 1082  
Sequence of events, 692  
Server-side cursor (`adUseServer`), 419  
Server object, 1005  
Set Next Statement, 219  
SetDisplayedCategories method, 797  
SetPermissions method, 629, 655, 660, 662  
 SetProperty, 1008  
setRequestHeader, 1006, 1082–1083  
SetTempVar, 968  
SHAPE statement, 513–514, 520  
Share-level security, 613, 627, 630  
Show Next Statement, 219  
Sinking events, 900–910  
Sinking events, in standalone class module,  
    900–910  
    cRecordLogger class, creating, 902–903  
    cRecordLogger custom class with another  
        form, 908–910  
    file preparation, 901–902  
    frmCustomers form, 902, 905  
    instance of custom class, creating, 905–907  
    Name property, 760, 786, 810, 825, 869, 876,  
        911, 916, 943, 1041  
    Object drop-down list, 903, 918  
    Procedure drop-down list, 903  
        testing cRecordLogger custom class, 907–908  
Skipping lines of code, in debugging, 219  
Snapshot-type Recordset, 309, 311, 314, 315,  
    317, 327, 337, 353  
Source parameter, 248, 415, 421  
Split button, 839–340  
Split function, 188, 194, 295, 727, 1096  
Spreadsheet constants, 397  
SQL. *See* Structured Query Language (SQL)  
SQL JOIN statements, 512–513  
SQL Pass-Through Queries, 476, 928  
SQL specifications, 544  
SQL WHERE clause, 337–338  
Square brackets [], 342  
Standalone class modules, 12, 900–910  
Standalone macros  
    creating, 933–938  
    running, 938–939  
Statements, 4  
Static array, 161–163  
Static variables in VBA procedures, 74–75  
Status property, 1084–1086  
StatusText property, 1085–1086  
Stepping through VBA procedure, 216–219  
    running procedure to cursor, 219  
    setting Next statement, 219  
    showing Next statement, 219  
    stepping out of procedure, 219  
    stepping over, 217–218  
Stop statement, 207–208  
Stopping and resetting, of VBA  
    procedures, 221  
Stored procedure(s), 593–609  
    changing database records with,  
        607–609  
    contents of, 685–686  
    creating, 593–597  
    creating parameterized, 600–602  
    deleting, 606–607  
    enumerating, 597  
    executing parameterized, 604–606  
StrMultiFldName argument, 324  
StrNewVal argument, 324  
StrSearch, 296  
StrTblName argument, 324  
Structured Query Language (SQL), 541  
Stylesheet, 994  
Submacros, creating, 819, 821, 939,  
    940–941  
Subroutine procedures (subroutines), 3–4, 186  
Subscripted variables, 151  
Supports method, 440  
Syntax errors, 197, 777, 898  
SysCmd method, 768  
System database (System.mdw), 631  
Substring function, 1012

**T**

- Table object, 385
- Table relationships, using ADO
  - one-to-many relationship, 412–414
  - parent-child relationship, 412
- Table-type Recordset, 305, 306, 307
- TableDef object, 242, 275, 278
- Table(s)
  - constraints, 567
    - CHECK, 567
    - FOREIGN KEY, 567
    - NOT NULL, 567
    - PRIMARY KEY, 567,
    - UNIQUE, 567
  - creating, 543–565
    - in current database (DDL with ADO), 546–549
    - in new database (DDL with ADO/ADOX), 549–551
  - Data Definition Query window, 548–549
  - deleting, 551–552
  - design data types and Access SQL
    - equivalents, 548–549
  - establishing relationship between, 574–575
  - modifying with DDL, 553–565
    - adding multiple-field index to table, 558–559
    - adding new fields to table, 553–554
    - adding primary key to table, 557–558
    - changing data type of table column, 554–555
    - changing seed and increment values of autonumber columns, 563–565
    - changing size of Text column, 555–556
    - deleting column from table, 556–557
    - deleting index, 561
    - deleting indexed column, 559–560
    - setting default value for table column, 562–563
- Templates, 973–978
  - .accdt file format, 973–977
  - custom blank database template, creating, 973
- Temporary variables, 71–74
- creating temporary variable with TempVars collection object, 72
- removing temporary variable from TempVars collection objects, 74
- retrieving names and values of TempVars objects, 72–73
- temporary global variables in expressions, 73
- TempVars collection exposed to macros, 74
- Testing and debugging
  - Add Watch window, 209–213
  - Assert statement, 208–209
  - breakpoints, using, 200–205
  - Call Stack dialog box, 215–216
  - Err object, using, 222–226
  - immediate window in break mode, 205–206
  - Locals Window, 214–215
  - navigating with bookmarks, 220–221
  - quick watch, 213–214
  - stepping through VBA procedure, 216–217
  - Stop statement, 207
  - stopping procedure, 199–200
  - trapping errors, 221–229
- Test method, 1075
- TextCompare, 1068
- Text property, 1041
- Timer event, 720–721
- Toggle button, 838–839
- Toggle folders, 29
- TOP keyword, 343
- TotalRec, 318
- Transaction processing, 363–368
  - creating transaction with ADO/DAO, 364–368
- TransferSpreadsheet method, 397
- TransformNode method, 997, 1006
- TransformNodeToObject method, 1052–1053
- TransformXML method, 1022, 1088
  - arguments of, 1028
- Trapping errors, 221–229
  - Err object, using, 222–226
  - On Error statement, 222
  - procedure testing, 226–229
  - setting options in visual basic project, 228–229
- Troubleshooting errors in arrays, 170–172

Trusted location folder for Access database, 23–26  
Type conversion functions (CSng), 76  
Type declaration characters, 60  
Type mismatch error, 108, 114, 173  
Type property, 277

## U

UBound function, 167–170  
Underscore character (\_), 342  
Uniform Resource Identifier (URI), 989  
Uniform Resource Location (*see URL*), 989  
Uniform Resource Locator (*see URL*), 989  
Uniform Resource Name (*see URN*), 989  
UNIQUE constraint, 567  
UNIQUE keyword, 558  
Unique property, 301  
Universal data link file (.udl), 256  
Update method, 319, 327, 329, 383, 418, 448, 449, 450, 452, 493, 501, 505  
Update query, 350–352  
    creating and running, 350–352  
    executing, 473–474  
UpdateBatch method, 418, 504  
URL, 989  
URN, 989  
User and group accounts (ADO)  
    creating, 643–647  
    deleting, 647–649  
    listing, 649–650  
    listing users in groups, 650–652  
User and Group Accounts window, 616–618  
User and group permissions  
    changing user password, 618–619  
    checking permissions for objects, 663–665  
    object owner, retrieving name of, 653–655  
    setting database password using  
        CompactDatabase method, 665–667  
    setting database password using  
        NewPassword method, 667–669  
    setting permissions for containers, 660–662  
    setting permissions for database, 658–660  
    setting permissions for object, 655–658  
User-defined property, 279

User interface (UI)  
Access 2021 Ribbon interface, 800–803  
Backstage view, customizing, 855–859  
CommandBars object and Ribbon, 852–853  
controls in Ribbon customizations  
    built-in control, 848–849  
    checkboxes, 841–843  
    combo boxes and drop downs, 844–845  
    dialog box launcher, 846–847  
    disabling control, 847–848  
    edit boxes, 843–844  
    refreshing Ribbon, 849–852  
    split buttons, menus, and submenus, 839–840  
    toggle button, 838–839  
creating  
    designing User Form, 878–880  
    writing event procedures, 880–889  
hiding elements of, 806  
images in Ribbon customization  
    attributes and callbacks, 836–837  
    requesting images via getImage callback, 831–837  
    requesting images via loadImage  
        callback, 828–831  
initial window, 789–790  
Navigation pane, 791–795  
Navigation pane, customizing  
    controlling display of database objects, 795–796  
    locking Navigation pane, 795  
    saving and loading configuration of, 798–799  
    setting displayed categories, 797–798  
Quick Access Toolbar (QAT), 859–861  
Ribbon programming with XML, VBA and Macros, 803–826  
    assigning Ribbon customizations to  
        forms and reports, 822–826  
    embedding Ribbon XML markup, 815  
loading Ribbon customizations from  
    external XML document, 809–816  
Ribbon customization XML markup,  
    creating, 804–808

- storing Ribbon customization XML markup, 817–822
- tab activation and group auto-scaling, 854–855
- User-level security, 614, 630–631
- UsysApplicationLog table, 962–963
- USysRibbons table, 816, 817, 818, 822, 823, 835, 852, 856
  
- V**
- Validation Rule property, 278
- Validation Text property, 278
- Variable type, 58
- Variables
  - assigning values to, 61–62
  - concatenation, 58
  - declaring, 55
  - declaring typed, 60–61
  - determining data type of, 79–80
  - explicit variable declaration, 55
  - finding variable definition, 79
  - forcing declaration of, 64–65
  - global, 70, 71, 72
  - implicit variable declaration, 55–56
  - informal, 56
  - initialization, 63
  - lifetime of, 71
  - module-level, 67–68
  - names, 54
  - object, 76–78
  - procedure-level (local), 66
  - project-level, 69–70
  - scope of, 66
  - specifying data type of, 58–59
  - static, 74–76
  - temporary, 71–72
  - type declaration characters, 60
- Variant data type, 51, 53, 55, 59, 60, 92, 142, 149, 156, 767
- VBA. *See* Visual Basic for Applications (VBA)
- VBA functions
  - Array function, 163–165
  - Erase function, 165–169
  - IsArray function, 164–165
- LBound and UBound functions, 169–170
- VBA programs, adding repeating actions
  - Do...Until statement, 137–139
  - Do...While statement, 134–137
  - For Each...Next statement, 142–143
  - exiting loops early, 143–143
  - For...Next statement, 139–140
  - infinite loops, avoiding, 137
  - looping statements, 133
  - paired statements, 142
  - variables and loops, 138
- VBA Project, 35
- VBE. *See* Visual Basic Editor (VBE)
- VBScript, 995, 997, 1005, 1073
- View(s)
  - creating, 593–597
  - deleting, 598–599
  - generating list of saved, 597–598
- Virtual directory, creating, 1102–1104
- Visual Basic Editor (VBE)
  - Code window, 32–24
  - Immediate window, 46–49
  - Object Browser, 42–45
  - other windows, 34
  - Project Explorer window, 28–29
  - Properties window, 30–31
  - renaming module, 36
  - syntax and programming assistance
    - Comment Block button, 42
    - Complete Word button, 40
    - Indent button, 41–42
    - List Constants button, 39–40
    - List Properties/Methods option, 36
    - Outdent button, 41–42
    - Parameter Info button, 38–39
    - Quick Info button, 40
    - Uncomment Block button, 42
  - VBA object library, using, 45–46
- Visual Basic for Applications (VBA), 3
  - assigning name to project, 35
  - data types, 51–53
  - debugging tools of, 195
  - object library, 45–46
  - procedures
    - compiling, 21

- event, 6  
executing, 10–12  
function, 5–6  
property, 6–7  
in standard modules, 7–9  
subroutine, 4–5  
stopping and resetting, of VBA procedures, 221
- W**
- Watch expressions  
adding, 209–213  
removing, 213  
vs. breakpoint, 210
- Well-formed document, 983
- WHERE clause, 337–338, 340, 341, 342, 344, 350, 351, 367, 458
- With...End With construct, 207, 282
- WithEvents keyword, 878, 900, 910, 911, 915, 917
- Workgroup information file, 630, 631–640  
Application Data folder, 631  
creating and joining, 633–634
- Write method, 457
- WriteLine method, 379, 438
- WWW-Authenticate header, 1082
- X**
- .xls, 336  
.xlsl, 336  
.xlsm, 336  
.xlsx, 336  
.xps file format, saving reports in, 766
- XML (Extensible Markup Language)  
ActiveX data objects with, 1046–1060  
attribute-centric and element-centric XML, 1049–1051  
saving ADO recordset as XML to disk, 1046–1047  
applying XSLT transforms to exported data, 1009–1012  
exporting data, 1013–1015  
advanced XML export options, 999–1009  
data export options, 1000–1001
- presentation export options, 1024–1026  
schema export options, 1001–1002  
XML data file, 985–1009  
XML documents formatted with stylesheets, 996–999  
XML schema file, 990–992  
XSL transformation files, 992–996  
exporting to and importing from XML, 1022–1037  
ExportXML method, 1022–1036  
ImportXML method, 1036–1037  
TransformXML method, 1028–1036
- importing data, 1017–1022  
XML data to Access database, 1019–1020  
XSD schema file to Access database, 1013–1015
- manipulating XML documents, 1037–1046  
applying XSL stylesheet, 1052–1054  
changing type of XML file, 1051–1052  
loading and retrieving contents of XML file, 1059–1061  
loading XML document in Excel, 1058–1059  
retrieving first matching node, 1045–1046  
retrieving information from element nodes, 1042–1043  
transforming attribute-centric XML data into HTML table, 1054–1058  
XML document nodes, 1040–1041  
well-formed XML document, 983–984
- XML, 981  
XML data file, 985–1009  
XML data import, 985  
XML data to Access database, 1019–1020  
XML document nodes, 1040–1042  
XMLDocument object, 1086  
XML documents formatted with stylesheets, 996–999  
XML Export Options, 1002–1003

- XML file, exporting, 1020–1022
- XML file, importing, 1017–1020
- XMLHTTP object, 1006
- XMLHttpRequest object, 1080–1082
- XML nodes, 1030–1035
- XmIns attribute, 806, 989
- XML property, 1039
- XML schema file, 990–991
- XPath, 995
- XSD schema, 1001, 1013–1015
- XSL (Extensible Stylesheet Language), 992
- XSLT (*see* XSL)
- XSLT, 983
- XSL transformation files, 992–996
- XSL Transformations (XSLT), 992, 995
- XSLT transformation, applying 1009–1012