

THOMSON

COURSE TECHNOLOGY

WEB DOWNLOADS  
AVAILABLE



Microsoft®  
**Access VBA**  
Programming,  
Third Edition

for the  
**absolute  
beginner™**

NO EXPERIENCE REQUIRED

"This series shows that it's possible to teach newcomers a programming language and good programming practices without being boring."

—LOU GRINZO,  
reviewer for Dr. Dobbs Journal

MICHAEL VINE



# Microsoft® Access™ VBA Programming for the Absolute Beginner Third Edition

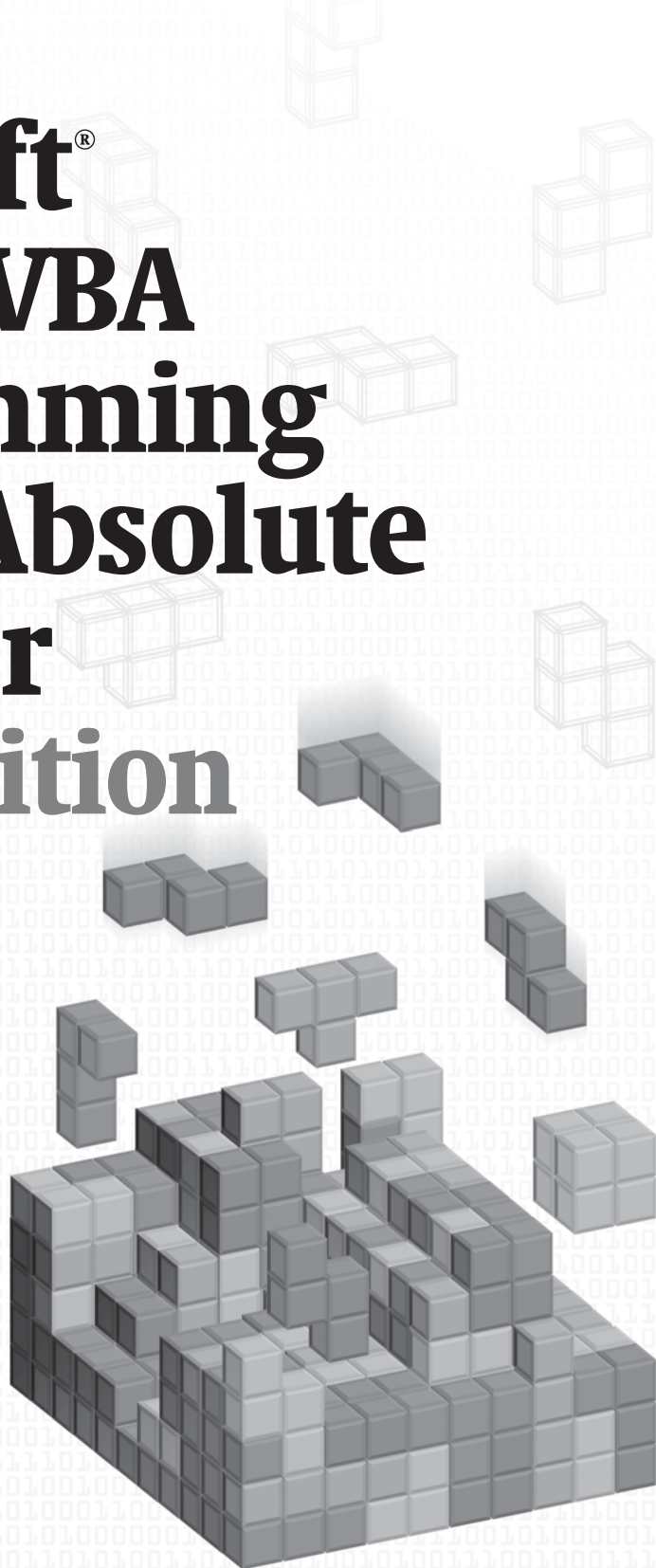
MICHAEL VINE

THOMSON



COURSE TECHNOLOGY™

Professional ■ Technical ■ Reference



© 2007 Thomson Course Technology, a division of Thomson Learning Inc. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system without written permission from Thomson Course Technology PTR, except for the inclusion of brief quotations in a review.

The Thomson Course Technology PTR logo and related trade dress are trademarks of Thomson Course Technology, a division of Thomson Learning Inc., and may not be used without written permission.

Microsoft, Access, and VBA are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are the property of their respective owners.

*Important:* Thomson Course Technology PTR cannot provide software support. Please contact the appropriate software manufacturer's technical support line or Web site for assistance.

Thomson Course Technology PTR and the author have attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

Information contained in this book has been obtained by Thomson Course Technology PTR from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Thomson Course Technology PTR, or others, the Publisher does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from use of such information. Readers should be particularly aware of the fact that the Internet is an ever-changing entity. Some facts may have changed since this book went to press.

Educational facilities, companies, and organizations interested in multiple copies or licensing of this book should contact the Publisher for quantity discount information. Training manuals, CD-ROMs, and portions of this book are also available individually or can be tailored for specific needs.

ISBN-10: 1-59863-393-7

ISBN-13: 978-1-59863-393-1

eISBN-10: 1-59863-754-1

Library of Congress Catalog Card Number: 2007923301

Printed in the United States of America

07 08 09 10 11 TW 10 9 8 7 6 5 4 3 2 1

**THOMSON**



**COURSE TECHNOLOGY**

**Professional ■ Technical ■ Reference**

Thomson Course Technology PTR,  
a division of Thomson Learning Inc.

25 Thomson Place

Boston, MA 02210

<http://www.courseptr.com>

**Publisher and General  
Manager, Thomson Course  
Technology PTR:**

Stacy L. Hiquet

**Associate Director of  
Marketing:**

Sarah O'Donnell

**Manager of Editorial  
Services:**

Heather Talbot

**Marketing Manager:**

Mark Hughes

**Acquisitions Editor:**

Mitzi Koontz

**Marketing Assistant:**

Adena Flitt

**Project Editor:**

Jenny Davidson

**Technical Reviewer:**

Keith Davenport

**PTR Editorial Services**

**Coordinator:**

Erin Johnson

**Interior Layout Tech:**

Digital Publishing Solutions

**Cover Designer:**

Mike Tanamachi

**Indexer:**

Katherine Stimson

**Proofreader:**

Kim V. Benbow

*To Sheila: 143*

# ACKNOWLEDGMENTS

Writing any book is not easy, especially a technical programming book. It takes many great, patient, and talented people to write, edit, design, market, finance, and produce a book. Without the assistance of Mitzi Koontz, Jenny Davidson, and Keith Davenport, it would be impossible for me to share with you my knowledge of programming in such a professional and fun manner.

# ABOUT THE AUTHOR

**M**ichael Vine has taught computer programming, web design, and database classes at Indiana University/Purdue University in Indianapolis, IN, and at MTI College of Business and Technology in Sacramento, CA. Michael has over 13 years' experience in the information technology profession. He currently works full time in a Fortune 100 company as an IT Project Manager overseeing the development of enterprise data warehouses.

*This page intentionally left blank*

# CONTENTS

<b>CHAPTER 1</b>	<b>AN INVITATION TO ACCESS 2007.....</b>	<b>1</b>
	What Is Microsoft Access?.....	1
	Microsoft Access 2007 Limitations.....	2
	Microsoft Office Suites.....	3
	System Requirements.....	4
	Working with Older Database Formats.....	6
	What's New in Access 2007.....	6
	User Interface.....	7
	Templates.....	9
	Datasheet View.....	11
	Layout View.....	12
	Calendar.....	13
	Rich Text.....	13
	Split Forms.....	14
	Multivalued Fields.....	14
	Data Types.....	15
	File Format.....	16
	Help.....	16
	Summary.....	17
<b>CHAPTER 2</b>	<b>ACCESS ESSENTIALS.....</b>	<b>19</b>
	Database Normalization.....	19
	1 <sup>st</sup> Normal Form.....	21
	2 <sup>nd</sup> Normal Form.....	22
	3 <sup>rd</sup> Normal Form.....	23
	Creating a New Access 2007 Database.....	25
	Tables and Fields.....	26
	Table Relationships.....	31
	Forms.....	35
	Common Controls.....	37
	Hungarian Notation.....	42
	Queries.....	43



Summary.....	46
Programming Challenges.....	48

## CHAPTER 3 INTRODUCTION TO ACCESS VBA.....49

The Event-Driven Paradigm.....	49
Object-Based Programming.....	50
The VBA IDE.....	51
Introduction to Event Procedures.....	53
Introduction to VBA Statements.....	55
Accessing Objects and Their Properties.....	56
The Forms Collection.....	57
The Me Keyword.....	58
Assignment Statements.....	59
Command and Label Objects.....	60
Getting User Input with Text Boxes.....	65
Variables and Beginning Data Types.....	67
Variable Naming Conventions.....	70
Variable Scope.....	71
Option Statements.....	72
VBA Arithmetic and Order of Operations.....	73
Chapter Program: Fruit Stand.....	74
Summary.....	79
Programming Challenges.....	80

## CHAPTER 4 CONDITIONS.....81

If Blocks.....	81
Nested If Blocks.....	83
Compound If Blocks.....	84
Select Case Structures.....	87
Dialog Boxes.....	88
Message Box.....	88
Input Box.....	91
Common Controls Continued.....	93
Option Group.....	93
Option Buttons.....	95
Check Boxes.....	98
Toggle Buttons.....	99
Chapter Program: Hangman.....	101
Summary.....	107
Programming Challenges.....	108

<b>CHAPTER 5</b>	<b>LOOPING STRUCTURES.....</b>	<b>109</b>
	Introduction to Looping Structures.....	109
	Do While.....	111
	Do Until.....	112
	Loop While.....	113
	Loop Until.....	114
	For.....	114
	List and Combo Boxes.....	116
	Adding Items.....	117
	Removing Items.....	121
	Managing Columns.....	122
	Random Numbers.....	124
	Chapter Program: Math Quiz.....	126
	Summary.....	129
	Programming Challenges.....	130
 <b>CHAPTER 6</b>	 <b>COMMON FORMATTING AND CONVERSION</b>	
	<b>FUNCTIONS.....</b>	<b>131</b>
	String-Based Functions.....	131
	UCase.....	132
	LCase.....	133
	Len.....	133
	StrComp.....	134
	Right.....	136
	Left.....	137
	Mid.....	137
	InStr.....	138
	Date and Time Functions.....	139
	Date.....	139
	Day.....	140
	WeekDay.....	140
	Month.....	140
	Year.....	140
	Time.....	140
	Second.....	141
	Minute.....	141
	Hour.....	141
	Now.....	142
	Conversion Functions.....	142
	Val.....	142
	Str.....	142

Chr.....	143
Asc.....	144
Formatting.....	144
Formatting Strings.....	145
Formatting Numbers.....	145
Formatting Date and Time.....	146
Chapter Program: Secret Message.....	148
Summary.....	151
Programming Challenges.....	152

## CHAPTER 7   **CODE REUSE AND DATA STRUCTURES.....153**

Code Reuse.....	153
Introduction to User-Defined Procedures.....	155
Subprocedures.....	156
Function Procedures.....	158
Arguments and Parameters.....	158
Standard Modules.....	161
Arrays.....	163
Single-Dimension Arrays.....	164
Two-Dimensional Arrays.....	166
Dynamic Arrays.....	167
Passing Arrays as Arguments.....	169
User-Defined Types.....	170
Type and End Type Statements.....	170
Declaring Variables of User-Defined Type.....	172
Managing Elements.....	173
Chapter Program: Dice.....	176
Summary.....	183
Programming Challenges.....	184

## CHAPTER 8   **DEBUGGING, INPUT VALIDATION, FILE PROCESSING, AND ERROR HANDLING.....185**

Debugging.....	185
Stepping Through Code.....	186
Breakpoints.....	187
Immediate Window.....	188
Locals Window.....	190
Watch Window.....	190
Input Validation.....	192
IsNumeric.....	192
Checking a Range of Values.....	194

Error Handling.....	196
The Err Object.....	199
The Debug Object.....	200
File Processing.....	201
About Sequential File Access.....	202
Opening a Sequential Data File.....	202
Reading Sequential Data from a File.....	203
Writing Sequential Data to a File.....	204
Closing Data Files.....	205
Error Trapping for File Access.....	206
Chapter Program: Trivial Challenge.....	208
Summary.....	213
Programming Challenges.....	214

## **CHAPTER 9    MICROSOFT ACCESS SQL.....215**

Introduction to Access SQL.....	215
Data Manipulation Language.....	218
Simple SELECT Statements.....	218
Conditions.....	220
Computed Fields.....	222
Built-In Functions.....	223
Sorting.....	227
Grouping.....	229
Joins.....	230
INSERT INTO Statement.....	232
UPDATE Statement.....	233
DELETE Statement.....	234
Data Definition Language.....	235
Creating Tables.....	235
Altering Tables.....	236
DROP Statement.....	237
Summary.....	237
Programming Challenges.....	239

## **CHAPTER 10    DATABASE PROGRAMMING WITH ADO.....241**

ADO Overview.....	241
Connecting to a Database.....	242
Working with Recordsets.....	246
Introduction to Database Locks.....	247
Introduction to Cursors.....	248
Retrieving and Browsing Data.....	249
Updating Records.....	258

Adding Records.....	261
Deleting Records.....	262
Chapter Program: Choose My Adventure.....	264
Summary.....	274
Programming Challenges.....	275

## **CHAPTER 11    OBJECT-ORIENTED PROGRAMMING WITH ACCESS VBA.....277**

Introduction to Object-Oriented Programming.....	277
Creating Custom Objects.....	278
Working with Class Modules.....	279
Property Procedures.....	281
Method Procedures.....	284
Creating and Working with New Instances.....	287
Working with Collections.....	290
Adding Members to a Collection.....	291
Removing Members from a Collection.....	292
Accessing a Member in a Collection.....	292
For Each Loops.....	293
Chapter Program: Monster Dating Service.....	294
Summary.....	300
Programming Challenges.....	301

## **CHAPTER 12    MACROS AND PERFORMANCE TUNING.....303**

Macros.....	303
Stand-Alone Macros.....	304
Macro Troubleshooting and Error Handling.....	311
Converting Macros to VBA.....	313
Access Database Performance Considerations.....	315
Forms.....	316
VBA Code.....	317
Queries and Indexes.....	317
Performance Analyzer.....	318
Summary.....	320
Programming Challenges.....	321

## **APPENDIX A    COMMON CHARACTER CODES.....323**

## **APPENDIX B    KEYBOARD SHORTCUTS FOR THE CODE WINDOW.....327**

<b>APPENDIX C</b>	<b>TRAPPABLE ERRORS.....</b>	<b>329</b>
<b>APPENDIX D</b>	<b>VISUAL BASIC ENVIRONMENT OPTIONS.....</b>	<b>333</b>
<b>APPENDIX E</b>	<b>RESERVED WORDS AND SYMBOLS.....</b>	<b>337</b>
	<b>INDEX.....</b>	<b>345</b>

# INTRODUCTION

**I**ntroduced in the early 1990s, Microsoft Access has become one of the most powerful and popular applications in the Microsoft Office suite of applications. Microsoft Access 2007 allows database developers and programmers to build dynamic and easily portable databases. Access comes with many easy-to-use features such as graphical forms, database templates, SQL query builders, as well as a subset of the Visual Basic language known as VBA for building data-driven applications.

*Microsoft Access VBA Programming for the Absolute Beginner, Third Edition*, is not a guide on how to use Access and its many wizards. There are already many books that do that! Instead, *Microsoft Access VBA Programming for the Absolute Beginner* concentrates on VBA programming concepts including variables, conditions, loops, data structures, procedures, file I/O, and object-oriented programming with special topics including database programming with ADO, Access SQL, Macros, and Access performance tuning recommendations.

Using Thomson Course Technology PTR's *Absolute Beginner* series guidelines' professional insight, clear explanations, examples, and pictures, you learn to program in Access VBA. Each chapter contains programming challenges, a chapter review, and a complete program that uses chapter-based concepts to construct a fun and easily built application.

To work through this book in its entirety, you should have access to a computer with Microsoft Access installed. The programs in this book were written in Microsoft Office 2007, specifically Access 2007. Those readers using older versions of Microsoft Access, such as Access 2002 or Access 2003, will find many of the VBA programming concepts still apply.

## WHAT YOU'LL FIND IN THIS BOOK

To learn how to program a computer, you must acquire a progression of skills. If you have never programmed at all, or have little to no experience with the Microsoft Access application, you will probably find it easiest to go through the chapters in order. Of course, if you are already an experienced programmer or seasoned user of Access, it might not be necessary to do any more than skim the earliest chapters. In either case, programming is not a skill you can learn by

reading. You'll have to write programs to learn. This book has been designed to make the process reasonably painless.

Each chapter begins with a brief introduction to chapter-based concepts. Once inside the chapter, you'll look at a series of programming concepts and small programs that illustrate each of the major points of the chapter. Finally, you'll put these concepts together to build a complete program at the end of the chapter. All of the programs are short enough that you can type them in yourself (which is a great way to look closely at code), but they are also available via the publisher's website ([www.courseptr.com/downloads](http://www.courseptr.com/downloads)). Located at the end of every chapter is a summary that outlines key concepts learned. Use the summaries to refresh your memory on important concepts. In addition to summaries, each chapter contains programming challenges that will help you learn and cement chapter-based concepts.

Throughout the book, I'll throw in a few other tidbits, notably the following:



These are good ideas that experienced programmers like to pass on.



There are a few areas where it's easy to make a mistake. I'll point them out to you as we go.

**Pay special attention to these areas for clarification or emphasis on chapter concepts.**

## IN THE REAL WORLD

**As you examine concepts in this book, I'll show you how the concepts are used beyond beginning programming.**



## WHO THIS BOOK IS FOR

*Microsoft Access VBA Programming for the Absolute Beginner, Third Edition* is designed for the beginning Access VBA programmer. Persons with backgrounds in other programming languages and databases will find this book to be a good tutorial and desk reference for Access VBA. Specifically this book is for the following groups:

- High school or college students enrolling or enrolled in an Access programming class
- Programming hobbyists and enthusiasts
- Office personnel with beginning database programming responsibilities
- Professional database developers wanting to learn the Microsoft Access VBA language
- Home users wanting to learn more about Access and VBA



# AN INVITATION TO ACCESS 2007

**I**ntroduced well over a decade ago, Microsoft Access is a fully functional RDBMS (Relational Database Management System) that has become one of the most popular and powerful programs in the Microsoft Office suite of applications.

Originally titled “Office 12”, Microsoft Office 2007 was released to business customers in late 2006 and to the general public in early 2007. Part of the Microsoft Office 2007 Professional, Professional Plus, Ultimate, and Enterprise suites, Access 2007 provides both beginning and professional database developers a cost effective way to leverage key database functionality with an easy-to-use graphical interface.

As of this writing, Microsoft allows you to download a free 60-day trial of Microsoft Office 2007 from this site: <http://us1.trymicrosoftoffice.com/>. Keep in mind that this is a really large download at around 400 megabytes, so you’ll want to ensure you download it with a fast Internet connection such as cable modem or DSL.

## WHAT IS MICROSOFT ACCESS?

In its rawest digital form, a database is simply a collection of data stored in a file for future retrieval and analysis. Modern databases have mechanisms for storing large amounts of data in a structured form and allow for many users to access the same database and even the same data at the same time. These database programs

often have many other features that include graphical interfaces for end users, reporting and query tools, security management controls, native programming support (e.g., VBA), and application interfaces to connect to other programs such as web and application servers. Microsoft Access is one such modern database that does all of this and more!

Because of its low cost and user-friendly interface, Microsoft Access has become quite popular for teaching beginning database skills and is just as popular in back office operations for building quick and relatively simple data management programs. In fact, Microsoft Access is often considered an upgrade from Microsoft Excel spreadsheets for organizations requiring data storage and analysis, but unfortunately don't have software development resources or the necessary skills to build complex database designs and solutions.

## MICROSOFT ACCESS 2007 LIMITATIONS

Prior to designing a database solution, it's important to understand and consider the benefits and limitations of the off-the-shelf tools you're evaluating. Microsoft Access has never been considered an "enterprise" database solution such as Microsoft SQL Server or Oracle, but nonetheless, it does satisfy most minor to intermediate organizational database requirements. To aid you in your evaluation of Access as a database solution, Table 1.1 describes the boundaries of Access 2007.

**TABLE 1.1 ACCESS 2007 LIMITATIONS**

<b>Limitation</b>	<b>Value</b>
Access database file size	2 gigabytes
Number of concurrent users	255
Number of database objects (including tables)	32,768
Table size	2 gigabytes
Number of characters in a table name	64
Number of characters in a field name	64
Number of open tables	2,048
Number of characters in a text field	255
Number of characters in a memo field	65,535 entered from UI, 2 gigabyte entered programmatically
Size of an OLE Object field	1 gigabyte
Number of indexes on a table	32
Number of fields in an index or primary key	10
Number of characters in a validation rule	2,048
Number of characters in a validation message	255

Number of characters in a record (excluding Memo & OLE Object field) when the UniCodeCompression property is set to Yes	4,000
Number of characters in a field property setting	255
Number of characters in an object name	64
Number of characters in a password	20
Number of characters in a user or group name	20
Number of open tables	2,048
Number of enforced relationships	32 per table
Number of tables in a query	32
Number of fields in a recordset returned by a query	255
Maximum recordset size	1 gigabyte
Sort limit	255 characters
Number of levels of nested queries	50
Number of characters in a cell in the query design grid	1,024
Number of characters in a parameter in a parameterized query	255
Number of ANDs in a WHERE or HAVING clause	99
Number of characters in a SQL statement	Approx 64,000
Number of characters in a label	2,048
Number of characters in a text box	65,535
Form or report width	22 inches (55.87 cm)
Section height	22 inches (55.87 cm)
Height of all sections	200 inches (508 cm)
Number of levels of nested forms or reports	7
Number of fields or expressions that can be sorted or grouped in a report	10
Number of headers or footers in a report	1 report header/footer, 1 page header/footer, 10 section headers/footers
Number of controls and sections added over the lifetime of a form or report	754
Number of SQL statement characters in a Recordsource or Rowsource property	32,750
Number of actions in a macro	999
Number of characters in a Macro/VBA condition	255
Number of characters in a Macro/VBA comment	255
Number of characters in an Action Argument	255
Number of form, report, and class modules	1,000

## MICROSOFT OFFICE SUITES

If you've been a user of Access for some time, you're probably aware that the Access database system won't come with every suite of Microsoft Office. If you're preparing to purchase one

of the new 2007 Office suites and want to leverage Access, you'll need to ensure you purchase or upgrade to the correct Microsoft Office, as shown in Table 1.2.

**TABLE 1.2    MICROSOFT OFFICE SUITES**

Office Suite	Includes Access 2007
Microsoft Office Basic 2007	no
Microsoft Office Home & Student 2007	no
Microsoft Office Standard 2007	no
Microsoft Office Small Business 2007	no
Microsoft Office Professional 2007	yes
Microsoft Office Professional Plus 2007	yes
Microsoft Office Ultimate 2007	yes
Microsoft Office Enterprise 2007	yes

## SYSTEM REQUIREMENTS

Whether you will be installing Access 2007 by itself or as part of a Microsoft 2007 Office suite, you'll need to be aware that system requirements have increased. System requirements for installing Access 2007 alone are described in Table 1.3.

**TABLE 1.3    MICROSOFT ACCESS 2007 SYSTEM REQUIREMENTS**

Component	Requirements
Processor	500 megahertz (MHz) or faster processor
Memory	256 MB of RAM or greater
Hard Disk	1.5 gigabytes (GB); a portion of which will be freed after installation
Drive	CD-ROM or DVD
Display	1024×768 or higher resolution monitor
Operating System	Microsoft Windows XP with SP2, Windows Server 2003 with SP1 or later operating system

System requirements increase slightly if you plan on installing the entire suite of Microsoft Office applications that come with Access 2007, including Professional, Professional Plus, Enterprise, and Ultimate suites, as detailed in Table's 1.4, 1.5, 1.6, and 1.7, respectively.

**TABLE 1.4    MICROSOFT OFFICE PROFESSIONAL 2007 SYSTEM REQUIREMENTS**

<b>Component</b>	<b>Requirements</b>
Processor	500 megahertz (MHz) or faster processor
Memory	256 MB of RAM or greater
Hard Disk	2 gigabytes (GB); a portion of which will be freed after installation
Drive	CD-ROM or DVD
Display	1024×768 or higher resolution monitor
Operating System	Microsoft Windows XP with SP2, Windows Server 2003 with SP1 or later operating system

**TABLE 1.5    MICROSOFT OFFICE PROFESSIONAL PLUS 2007 SYSTEM REQUIREMENTS**

<b>Component</b>	<b>Requirements</b>
Processor	500 megahertz (MHz) or faster processor
Memory	256 MB of RAM or greater
Hard Disk	2 gigabytes (GB); a portion of which will be freed after installation
Drive	CD-ROM or DVD
Display	1024×768 or higher resolution monitor
Operating System	Microsoft Windows XP with SP2, Windows Server 2003 with SP1 or later operating system

**TABLE 1.6    MICROSOFT OFFICE ENTERPRISE 2007 SYSTEM REQUIREMENTS**

<b>Component</b>	<b>Requirements</b>
Processor	500 megahertz (MHz) or faster processor
Memory	256 MB of RAM or greater
Hard Disk	2 gigabytes (GB); a portion of which will be freed after installation
Drive	CD-ROM or DVD
Display	1024×768 or higher resolution monitor
Operating System	Microsoft Windows XP with SP2, Windows Server 2003 with SP1 or later operating system

**TABLE 1.7    MICROSOFT OFFICE ULTIMATE 2007 SYSTEM REQUIREMENTS**

Component	Requirements
Processor	500 megahertz (MHz) or faster processor
Memory	256 MB of RAM or greater
Hard Disk	3 gigabytes (GB); a portion of which will be freed after installation
Drive	CD-ROM or DVD
Display	1024×768 or higher resolution monitor
Operating System	Microsoft Windows XP with SP2, Windows Server 2003 with SP1 or later operating system

## WORKING WITH OLDER DATABASE FORMATS

To convert an Access 95 or 97 version database (MDB file format), simply open it in Access 2007 to launch the Database Enhancement dialog box that will assist you in upgrading the database to the new 2007 file format (ACCDB). Once you have converted the older database, you will be able to make design changes to it, but you will no longer be able to open it in the version (Access 95 or 97) in which it was originally created.

Access versions 2000, 2002, and 2003 are easily opened in Access 2007 and can be saved in the new Access 2007 file format (ACCDB) or saved as versions 2000, 2002, or 2003 (MDB file format) for backward-compatibility purposes.

## WHAT'S NEW IN ACCESS 2007

It's an exciting time to contemplate upgrading your Access database and/or VBA programming skills. As an author, college instructor, and information technology professional, I've been working with Microsoft Access databases for quite some time, and I have to say Access 2007 has matured with a number of enhancements that even the most seasoned database professional will appreciate, all in a little package.

At first glance, the most noticeable change to Microsoft Access is the user interface, which Microsoft really turned on its head! It's a really dramatic change that you might find unnerving at first, but after working in it for a while, I'm sure you'll find it quite useful and intuitive, just like I did.

In addition to the enhanced graphical experience, Microsoft has done a great job of pouring lots of innovative features into Access 2007 that should give you, an Access developer, the feeling of driving a more robust database than its predecessors.

So far in my analysis of Microsoft Access 2007, my only complaint is that Microsoft removed support for Data Access Pages. If you've used Data Access Pages before in Access, you most likely appreciated Microsoft's simple to develop, easy to deliver, and useful Internet interface for your Access users. Though opening a prior version of Access containing Data Access Pages in Access 2007 will allow you to view them in Internet Explorer, unfortunately you cannot do much else. Microsoft's strategy behind eliminating Data Access Pages is to move web-based users of Access to Microsoft Windows SharePoint Services. If you'd like more information about Microsoft SharePoint Services, please visit Microsoft's SharePoint site: <http://office.microsoft.com/en-us/sharepointserver/default.aspx>.

Notwithstanding the elimination of Data Access Pages, application developers, instructors, students, and database hobbyists and enthusiasts will find Access 2007 not only supports most of their requirements, but does so in a professional, insightful, and elegant manner.

Without further ado, let's now take a trek together through a sampling of Access 2007's new and enhanced features, including:

- User interface
- Templates
- Datasheet view
- Layout view
- Calendar
- Rich text
- Field List pane
- Split forms
- Multivalued fields
- Data types
- File format
- Help

## User Interface

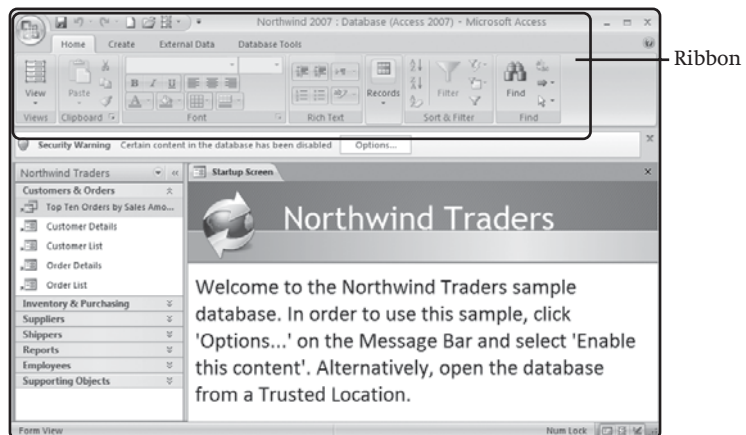
Microsoft has done a great job ensuring the user experience is similar across its most common of Office applications (Word, Excel, PowerPoint, Outlook, and of course Access), using a new and common graphical interface. After opening a new database, you'll notice the Getting Started with Microsoft Office Access page, as seen in Figure 1.1, that allows you to get started creating a new database, leverage pre-built templates, and link to useful Access information online.



**FIGURE 1.1**

The Getting Started with Microsoft Office Access page.

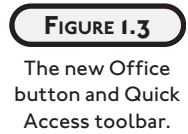
One of the most visibly different interface changes to Access 2007 is a new graphical dashboard at the top of the window called the Ribbon, as revealed in Figure 1.2. The Ribbon moves previously embedded commands from menus to a rich new tabular design. For example, you only need to click on the Create tab to view options for building forms and reports or click on the Home tab to sort, filter, and find records.

**FIGURE 1.2**

The new Ribbon.

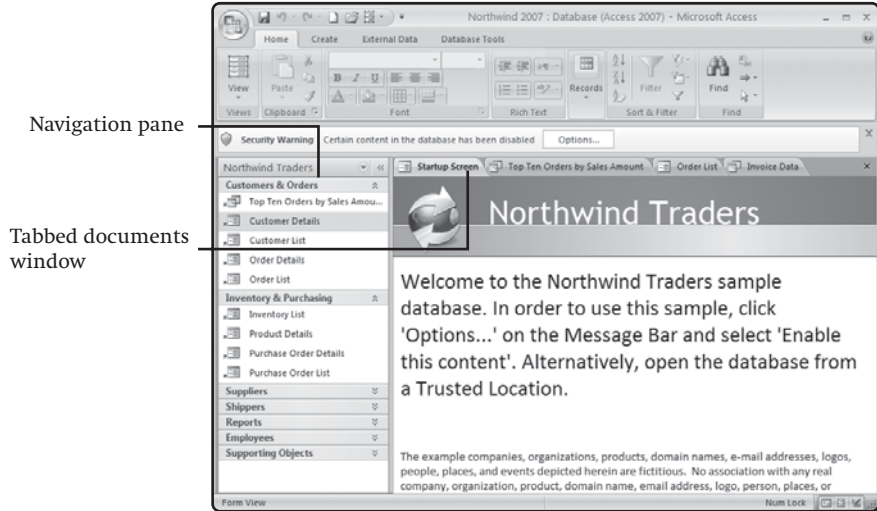
Microsoft's intention of the Ribbon is to allow Office users to focus on what they want to do through an easily seen common set of operations. Of course, individual experiences may vary depending on how long it takes you to acclimate to the sleek new design, but as I've found

Embedded in the new Ribbon are an Office button and a Quick Access toolbar that move common File commands, such as New, Open, Save, Exit, and many others from a traditional Windows-driven menu, to easily found locations, as shown in Figure 1.3.

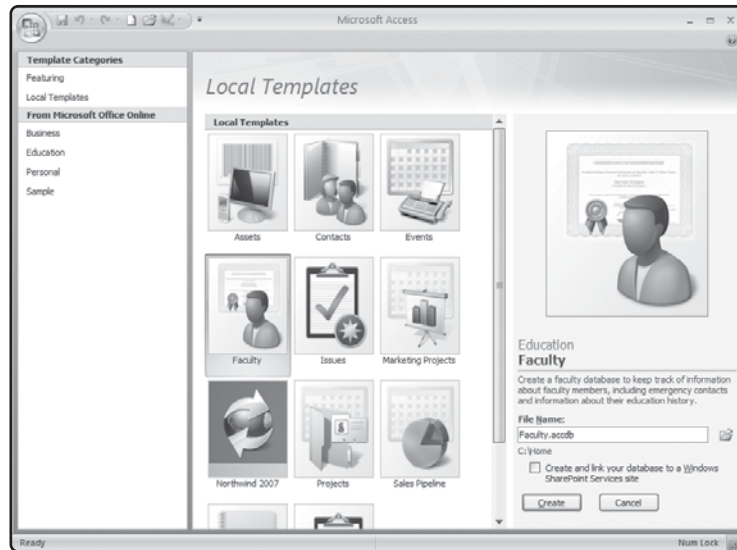


**If the Navigation pane is collapsed, you will need to expand it or resize it with your mouse.**

A definitive head start for Access application developers is Microsoft's set of templates, as shown in Figure 1.5, that include professionally developed out-of-box tables, forms, queries, and reports. Each template is a complete application that will give you a head start with database development by leveraging them as is, or modifying them to suit your needs.

**FIGURE 1.4**

The new Navigation pane and Tabbed documents window.

**FIGURE 1.5**

Out of the box professionally developed templates.



You can leverage the Getting Started with Microsoft Office Access page to link to updated and/or new templates.

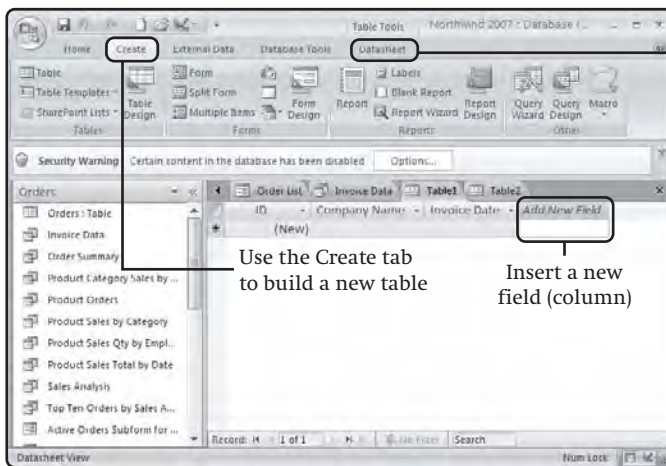
Featured Access 2007 database templates include:

- Assets
- Contacts

- Issues
- Events
- Marketing projects
- Projects
- Sales pipeline
- Tasks
- Faculty
- Students

## Datasheet View

Using the Create tab of the Ribbon, you can click the Table icon to launch the Datasheet view that allows you to easily enter new columns, as shown in Figure 1.6. You can then use the Datasheet tab in the Ribbon to change a field's (column) data type.



Use the Datasheet tab to change a column's data type

Use the Create tab to build a new table

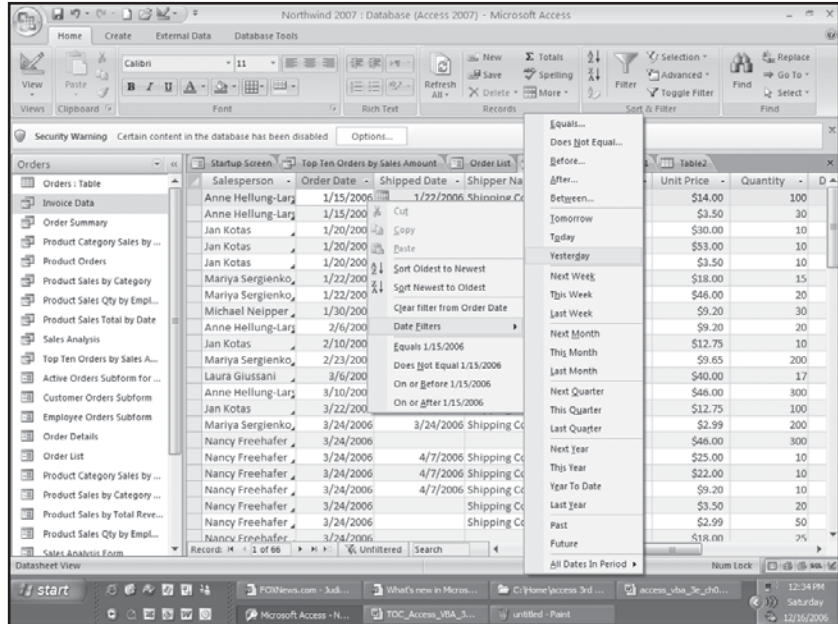
Insert a new field (column)

**FIGURE 1.6**

Working in the Datasheet view.

Sorting and filtering capabilities are accessible via menus that become available when right-clicking a column's value, as shown in Figure 1.7.

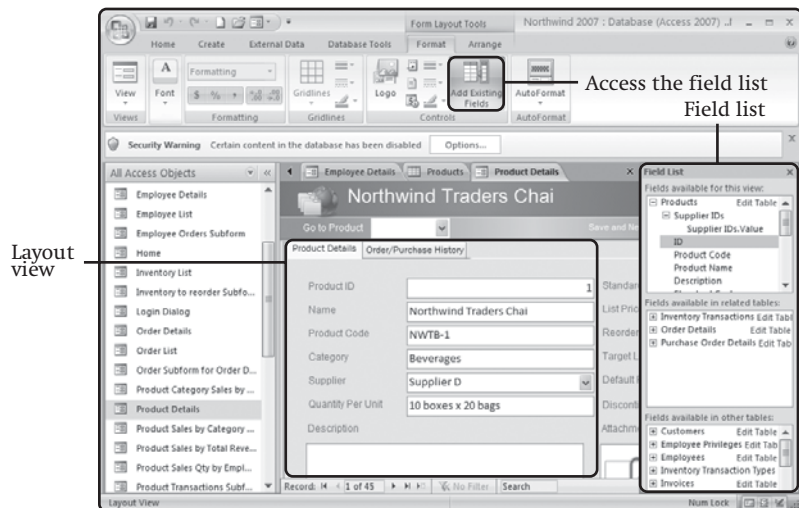
An intelligent design changes the filter menu items based on the data type (numeric, date, and text) of the field clicked.

**FIGURE 1.7**

Filtering rows in the Datasheet view.

## Layout View

An option from the Ribbon's view button, the new Layout view allows you to make changes to a live form or report through various formatting tools and dragging and dropping fields from the Field List window, as seen in Figure 1.8.

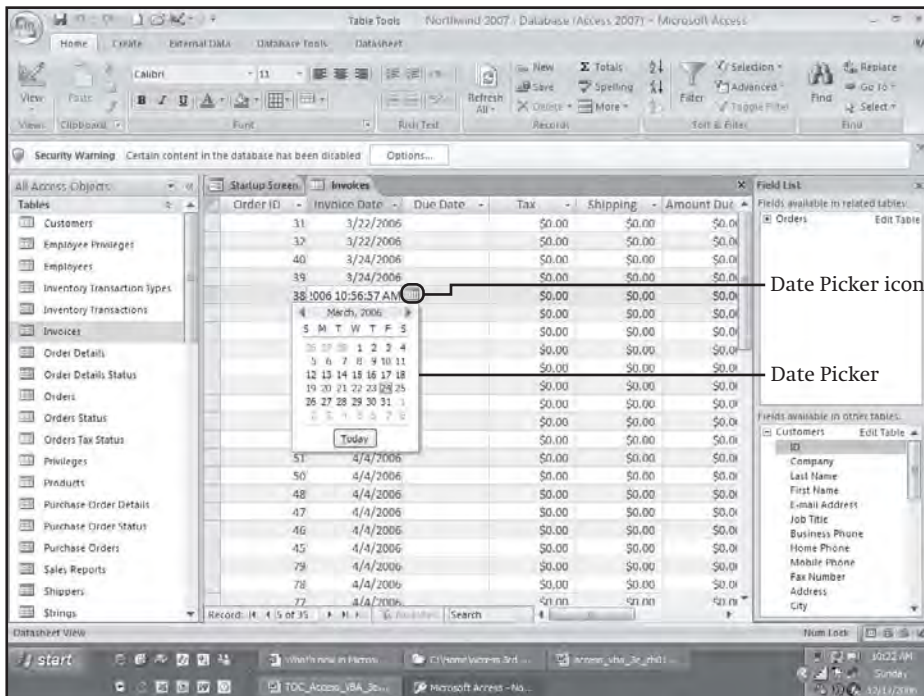
**FIGURE 1.8**

Developing forms and reports in real-time in Layout view.

I really like the new Layout view because it gives you a WYSIWYG (What You See Is What You Get) feeling during report and form development that's not available during Design view.

## Calendar

An interactive calendar icon, called the date picker (as shown in Figure 1.9), has been added to the right of the date fields in the Datasheet view. The date picker allows you to interactively traverse through a calendar to look up and select date values.



**FIGURE 1.9**

Using the date picker to look up and select calendar date values.

The date picker can be turned off in Design view by choosing a table's field and selecting Never for the Show Date Picker property.

## Rich Text

A nice addition to Memo data types is the inclusion of rich text that allows you to create and format text like you might in a Microsoft Office Word document. This includes font type, color, size, and other font layouts such as bold, italicize, and underline.

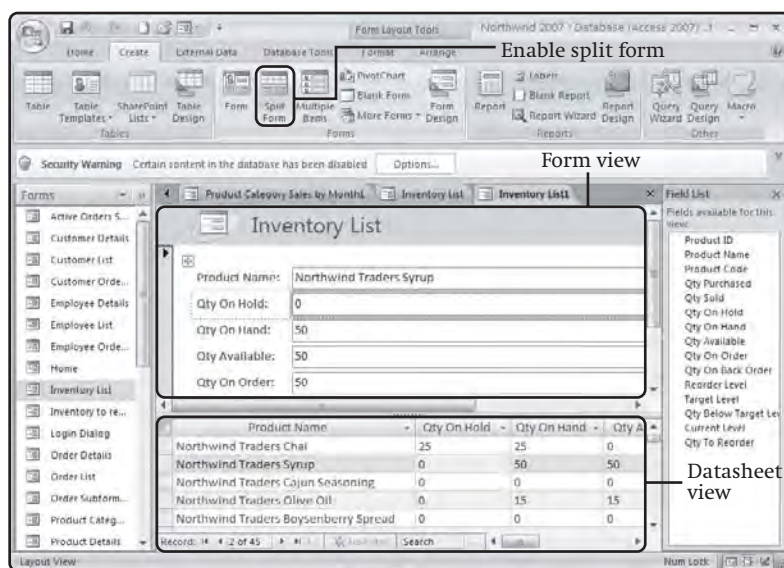
Rich text functionality can be turned off for Memo data types in Design view by changing the Text Format property to Plain Text.



## Split Forms

The new split forms option allows you to work with a form in Datasheet view and Form view at the same time. As shown in Figure 1.10, the split form components (Datasheet view and Form view) connect to the same data source at the same time, allowing you to quickly find a record and edit it in either Form or Datasheet view providing of course, the record is updatable.

To enable or create a split form, simply click the Create tab of the Ribbon and choose Split Form in the Forms section, as shown in Figure 1.10.



**FIGURE 1.10**

Enabling and viewing a split form.

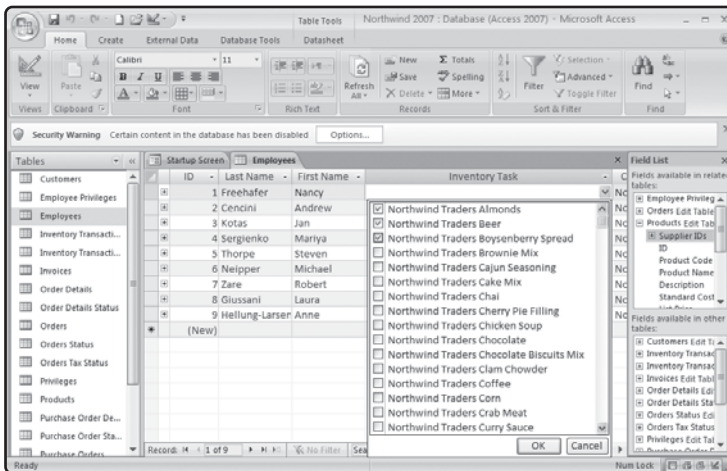
## Multivalued Fields

Without having to build a complex database design, Microsoft Access 2007 allows you to create multivalued fields that store several values for a single field. This is a very interesting new feature if you consider that not all professional database systems allow this type of functionality without some creative design approaches.

In reality, Access 2007 doesn't actually store multiple values in a single field even though the graphical representation shown in Figure 1.11 reveals otherwise. Access actually stores the multiple values separately in system tables and brings them back as necessary for graphical display.

Multivalued fields are created from the Ribbon by selecting the Datasheet tab and clicking the Lookup Column option from within the Fields and Columns area. As revealed in

Figure 1.11, I added a new multivalued field to an Employees table called Inventory Task and assigned multiple values, in this case inventory tasks, to a single employee.

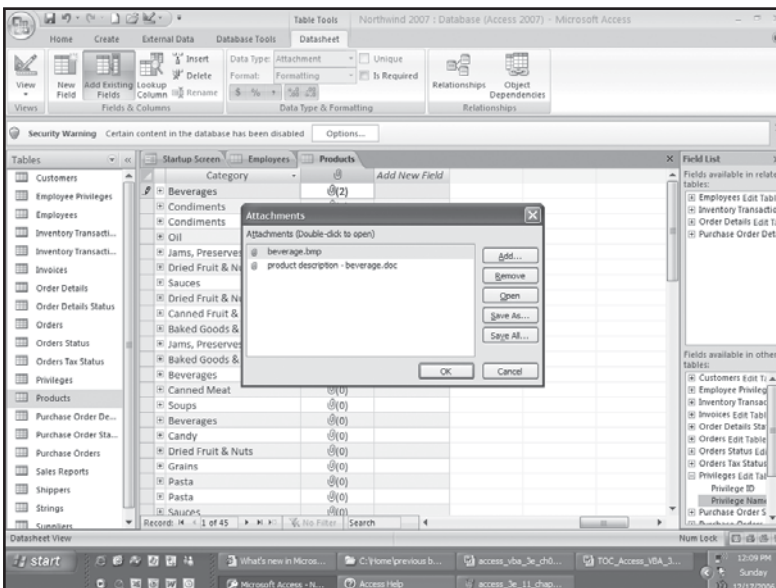


**FIGURE 1.11**

Working with multivalued fields.

## Data Types

A much welcomed enhancement to Access 2007 is the Attachment data type that allows you store both textual and binary files such as digital pictures or movies. Shown in Figure 1.12, you can assign multiple attachments to a single record using the Attachment data type.



**FIGURE 1.12**

Assigning multiple files to a field with the Attachment data type.



## File Format

The new Office Access 2007 file format has been customized to implement and store multi-valued lookup fields, Attachment data types, Memo field history tracking, and integration with Microsoft Outlook 2007 and Microsoft Windows SharePoint Services.

Microsoft Access 2007 introduces the following new file extensions.

- **ACCDB**—The new Office Access 2007 format, which replaces the MDB file extension.
- **ACCDE**—Access 2007 execute only file, which replaces the MDE file extension. ACCDE files have all VBA code removed allowing a user to execute the code, but not modify it. To create an ACCDE file in Access 2007, open your database, select the Database Tools tab from the Ribbon, and click the Make ACCDE icon in the Database Tools section.
- **ACCDT**—Access 2007 database template extension.
- **ACCDR**—A new file extension that enables a database to open in runtime mode. Created by simply changing the database's file extension from ACCDB to ACCDR, the new file extension allows for a locked-down version of your database.

## Help

Access 2007 merged end-user help and VBA developers help into one Help viewer, as shown in Figure 1.13. This will certainly be a welcome change that eliminates switching back and forth between multiple help windows to research a question.



**FIGURE 1.13**

Use the new Help viewer to access both end-user and developer assistance.

## SUMMARY

- A database is a collection of data stored in a file for future retrieval and analysis.
- Modern database programs often have features that include graphical interfaces, reporting and query tools, security management controls, native programming support (e.g., VBA), and application interfaces to connect to other programs.
- Access 2007 allows you to convert older versions of Access databases to the new file format (ACCDB) and save new Access 2007 databases to an older Access format (MDB).
- The Access 2007 database system comes with Microsoft Office 2007 Professional, Professional Plus, Ultimate, and Enterprise suites.
- System requirements for Access 2007 and Microsoft Office 2007 suites have increased in CPU, memory, and disk necessities.
- Access 2007 removes support for Data Access Pages.
- Part of the new Office Access 2007 user interface improvements, the Ribbon moves previously embedded commands from menus to a rich new tabular design.
- A number of free database templates come with Access 2007, which include professionally developed out-of-box tables, forms, queries, and reports.
- Sorting and filtering has been greatly enhanced with the new Datasheet view.
- The new Layout view allows you to make changes to a live form or report.
- The date picker, available in Datasheet view, allows you to interactively traverse through a calendar to look up and select date values.
- The Memo data type now includes support for rich text.
- The new split forms option allows you to work with a form in Datasheet view and Form view at the same time.
- Multivalued fields allow you to store multiple values for a single field.
- The new Attachment data type allows you to store both textual and binary files.
- Access 2007 implements new file extensions, ACCDB, ACCDE, ACCDT, and ACCDR that support numerous enhancements.
- End-user help and VBA development support have been merged into one Help viewer.

*This page intentionally left blank*



# ACCESS ESSENTIALS

**N**ow that you've seen what's new in Access 2007, it's time to take a look at how to design a database through normalization and implement it using tables, relationships, forms, controls, and queries. I'll show you how this is done by designing and building a small database to manage students and homework.

For those of you already familiar with database normalization, Access databases, and objects, you may want to skip right to Chapter 3, "Introduction to Access VBA," but if this is your first encounter with Access, or if you need a refresher in beginning Access database skills, I strongly recommend this chapter as a prerequisite for the remainder of this book.

## DATABASE NORMALIZATION

A relational database allows developers to link one or more tables using keys. Before physically creating and linking two tables, however, database developers model their data, or normalize it, using database normalization techniques.

Database normalization is in essence the process by which one optimizes table structures to enhance query performance and eliminate data integrity issues by investigating data requirements and their relationships to each other. To get

started, let's review the following key database normalization terms that are essential to this chapter and the remainder of the book.

- **Entity**—An entity is the main data object for which you are collecting information to be saved in a table. For example, a student entity would generally be composed of student-like data such as first name, last name, date of birth, and a student id.
- **Attribute**—Attributes are the bits of related information that make up an entity. Attributes such as first name, last name, and student id make up the student's entity.
- **Primary key**—A primary key is either a single field or a combined set of fields that uniquely identifies a single row of data in a table. Examples of unique identifiers that would make a good candidate for a primary key are social security, ISBN, student id, or any other value that would uniquely identify every row in a table. Note that most database management systems, including Microsoft Access, allow you to create a primary key for each table using a built-in data type called `AutoNumber`, which increments automatically each time a new row is entered into the table.
- **Foreign key**—The foreign key creates the foundation of a relationship between two tables by inserting the primary key from one table into another table. For example, a student id from a student's entity would be placed into a grades entity to show a relationship between students and their grades.
- **One-to-one relationship**—Though not very common, a one-to-one relationship denotes that each row of information for one entity relates to exactly one row of information for another entity. For example, you might want remove sensitive information from an `Employee` table and put it into a separate table called `Employee_Private` to hold private data such as social security and salary data. In this case, for each record in the `Employee` table, there would be one matching record in the `Employee_Private` table.
- **One-to-many relationship**—The most common of table relationships is the one-to-many relationship, which is created by adding the primary key (one or more fields) from one table into a second table that will hold many occurrences of said primary key. For example, the relationship between an `Assignment` table and an `Assignment_Results` table is one-to-many because for each assignment there will be multiple assignment results (rows) representing the many students who took the assignment.
- **Many-to-many relationship**—To build a many-to-many relationship, you need to create a third table that breaks the many-to-many relationship into two one-to-many relationships. You accomplish this by inserting the primary key from each of the two tables into the third table and, as a result, the third table's primary key is the combination of keys from the first two tables.

Leveraging the terms above and through a series of normal forms (processes), I can model student and homework data used to create this chapter's sample database. Before proceeding, however, let's take a look at my data requirements, which include:

- Store student information with name, college, and chosen major.
- Store assignment information with a description and assigned grades for each student for one class.

## 1<sup>st</sup> Normal Form

The first form in the database normalization process is rather simple and is most appropriately called 1<sup>st</sup> Normal Form. The essential rules for the 1<sup>st</sup> Normal Form are:

- Create separate tables for each group of related data and ensure each row is unique by identifying a primary key.
- Eliminate duplicate columns from the same table.

Since my goal for this chapter is to create a database that can manage both student and homework data for a single class, I will create two entities that separate distinct groups of data.

A Student entity with associated attributes:

- Student\_Id (primary key)
- First\_Name
- Last\_Name
- College
- Major

An Assignment\_Results entity with associated attributes:

- Assignment\_Id (primary key)
- Student\_Id (primary key)
- Assignment\_Description
- Assignment\_Completed
- Assignment\_Score

Entities need to have an identifier (attribute) that will uniquely identify each row of information. Remember that unique identifiers in the world of databases are called primary keys. The primary keys for the student entity will be Student\_Id and the primary key for the homework results entity will be comprised of two attributes, Assignment\_Id and Student\_Id. Note

primary keys are often a single attribute, but they can be created using more than one attribute, as in the case of the homework results entity.

## 2<sup>nd</sup> Normal Form

Continuing our normalization process, we'll now look at the 2<sup>nd</sup> Normal Form, which concentrates on removing redundant data from tables. Specifically, the essential rules of the 2<sup>nd</sup> Normal Form are:

- Meet all requirements of the 1<sup>st</sup> Normal Form.
- Remove redundant data and place them in separate tables.
- Create relationships between tables through the use of foreign keys.

To demonstrate, let's look at some sample data for our `Assignment_Results` table (entity) in Table 2.1.

**TABLE 2.1     SAMPLE DATA FOR THE ASSIGNMENT\_RESULTS TABLE**

Assignment_Id	Student_Id	Assignment_Description	Assignment_Completed	Assignment_Score
1	555-55-5555	Boolean algebra exercises 10-40	yes	85
1	444-44-4444	Boolean algebra exercises 10-40	yes	99
1	222-22-2222	Boolean algebra exercises 10-40	yes	78
2	555-55-5555	Venn diagram exercises 1-25	yes	89
2	444-44-4444	Venn diagram exercises 1-25	yes	100
2	222-22-2222	Venn diagram exercises 1-25	no	0

Please note I've migrated from using the term entity to using table, but for the sake of this discussion they are synonymous with each other.

Looking at Table 2.1 you might notice the existence of duplicate data being stored in my `Assignment_Results` table, namely in the `Assignment_Description` field. Adhering to the 2<sup>nd</sup> Normal Form rule, I need to remove redundant textual data, which over time will degrade

query performance, by defining a new table called `Assignment` and removing the `Assignment_Description` field from the `Assignment_Results` table. Tables 2.2 and 2.3 demonstrate sample values from my new `Assignments` table and updated `Assignment_Results` tables.

**TABLE 2.2 SAMPLE DATA FOR THE ASSIGNMENTS TABLE**

<b>Assignment_Id</b>	<b>Assignment_Description</b>
1	Boolean algebra exercises
2	Venn diagram exercises

**TABLE 2.3 SAMPLE DATA FOR THE ASSIGNMENT\_RESULTS TABLE**

<b>Assignment_Id</b>	<b>Student_Id</b>	<b>Assignment_Completed</b>	<b>Assignment_Score</b>
1	555-55-5555	yes	85
1	444-44-4444	yes	99
1	222-22-2222	yes	78
2	555-55-5555	yes	89
2	444-44-4444	yes	100
2	222-22-2222	no	0

Looking at Tables 2.2 and 2.3, you can see the `Assignment_Id` field has become a primary key in the `Assignments` table and a foreign key in the `Assignment_Results` table. This association is considered a one-to-many relationship because for every one `Assignment_Id` in the `Assignments` table, I will have many `Assignment_Ids` in the `Assignment_Results` table.

### 3<sup>rd</sup> Normal Form

Though not the last form in the database normalization process, the 3<sup>rd</sup> Normal Form is typically the last form exercised in beginning database development and, therefore, is the most appropriate conclusion for this section. The rules for the 3<sup>rd</sup> Normal Form are:

- Meet all requirements of the 2<sup>nd</sup> Normal Form.
- Remove all attributes from a table that are not directly dependent on the primary key.

To demonstrate 3<sup>rd</sup> Normal Form, consider sample data from our `Students` table in Table 2.4.



**TABLE 2.4** SAMPLE DATA FOR THE STUDENTS TABLE

Student_Id	First_Name	Last_Name	College	Major
555-55-5555	Sheila	Smith	School of Engineering	Industrial Engineering
444-44-4444	Michael	Vine	School of Science	Computer Science
222-22-2222	Wyatt	Jones	School of Science	Computer Science

Looking at Table 2.4 not only can we see repeating information, which breaks the rule for 2<sup>nd</sup> Normal Form, but we also have an attribute, College, that is not directly dependent on the Students table primary key, Student\_Id. Rather, the College attribute is actually dependent on the Major attribute. To adhere to the rules of the 3<sup>rd</sup> Normal Form, I will remove the College attribute from the Students table and replace the Major attribute with a Major\_Id field. My new Students table with sample data is revealed in Table 2.5.

**TABLE 2.5** SAMPLE DATA FOR THE STUDENTS TABLE

Student_Id	First_Name	Last_Name	Major_Id
555-55-5555	Sheila	Smith	1000
444-44-4444	Michael	Vine	2000
222-22-2222	Wyatt	Jones	2000

As you might expect, additional tables would need to be created to support both college and major information. I will now define those two new tables with sample data to hold information about the colleges and majors and their relationships to each other, as shown in Tables 2.6 and 2.7.

**TABLE 2.6** SAMPLE DATA FOR THE MAJORS TABLE

Major_Id	Major_Description	College_Id
1000	Industrial Engineering	111
1001	Civil Engineering	111
2000	Computer Science	222

**TABLE 2.7    SAMPLE DATA FOR THE COLLEGES TABLE**

College_Id	College_Description
111	School of Engineering
222	School of Science

Now that we've defined our data and normalized it, let's create our database in Access 2007.

## CREATING A NEW ACCESS 2007 DATABASE

As mentioned in Chapter 1, Access 2007 comes with a suite of downloadable templates that you can leverage for creating database solutions. I personally like templates and find that they can be a time-saving feature during database development. From my professional experience however, most out-of-the-box database templates don't meet all your requirements and thus get altered, which means you need to be familiar enough with the database design you're working with to have created the template in the first place! Since this is a database essentials chapter, we'll keep away from the templates and create our own database from scratch.

Let's get started by launching the Access program icon via the Start/Programs menu, or from any other shortcut you may have created during or after the Microsoft Office (Access) installation.

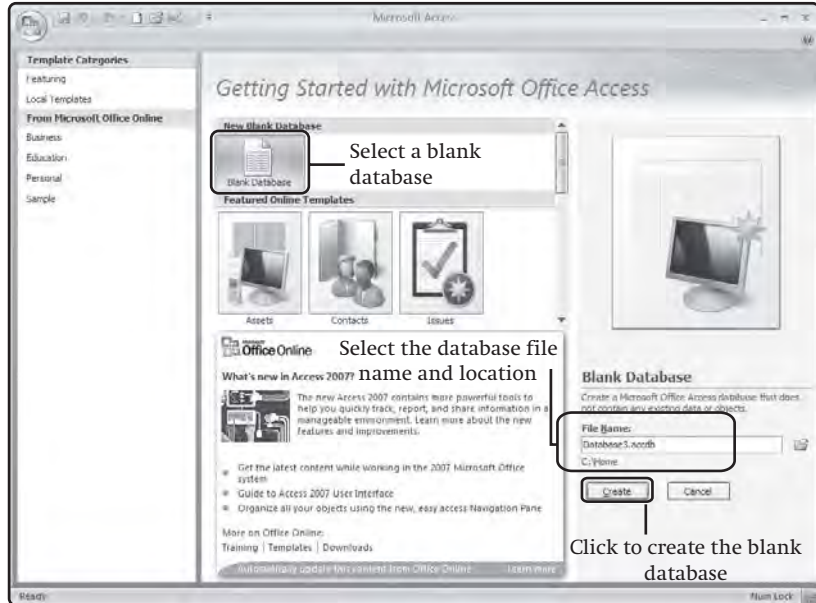
Once opened, you should see the Getting Started with Microsoft Office Access page, from which you click the Blank Database icon in the upper-middle part of the window, as shown in Figure 2.1.

Once you have clicked on the Blank Database icon, you can now name the database file, select its location, and click the Create button, as shown in Figure 2.1.

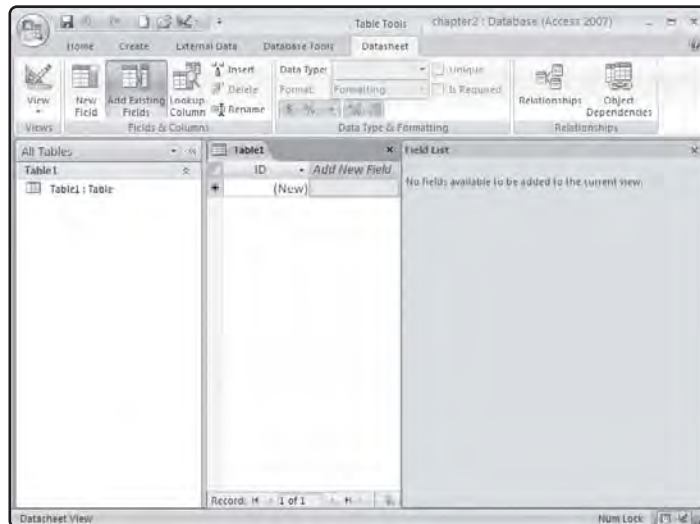


Microsoft Access databases are stored in binary files with the file extension .accdb. These files store relevant information about your Access database, including tables, fields, reports, queries, and much more.

After the database has been created by Access, you'll see a window similar to that shown in Figure 2.2, which in essence is your blank canvas to begin creating and working with various Access components such as tables and fields.

**FIGURE 2.1**

The Getting Started with Microsoft Office Access page.

**FIGURE 2.2**

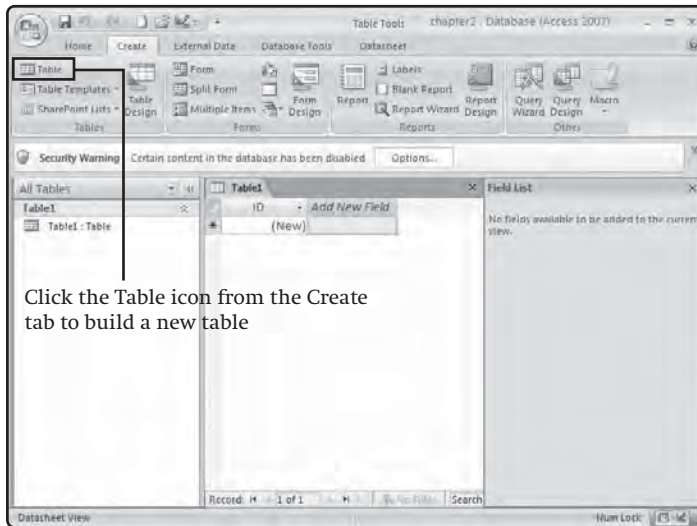
A newly created Access database.

## TABLES AND FIELDS

Each field has a definition that tells Access how the field values should be stored. For example, an `Assignment_Id` field is stored as a number as opposed to an `Assignment_Description` field, which is stored as text.

Each Access table must contain at least one or more fields. Together the fields comprise a row of data also known as a record. Tables should be assigned a primary key, which identifies each row of information in the table as unique. For example, the `Student_Id` field in the `Students` table is an excellent candidate for the primary key. The notion of primary keys is essential for creating relationships between tables in Access.

To create a table in Access 2007, simply click the Table icon from the Create tab of the Ribbon, as shown in Figure 2.3.

**FIGURE 2.3**

Creating tables in Access 2007.

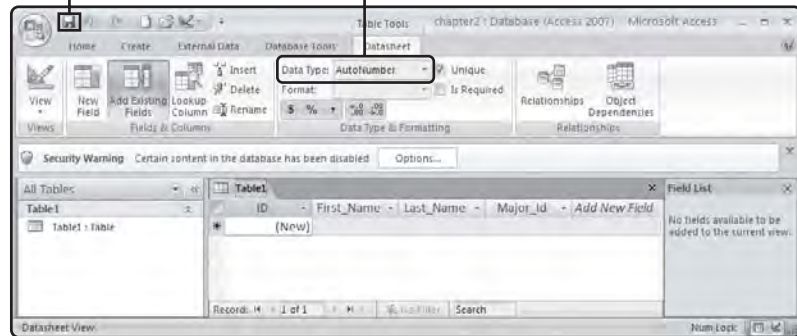
Access puts you in the Datasheet view when first creating a table. The Datasheet view allows you to add fields, lookup columns, relationships, dependencies, and assign data types and primary keys. Additionally, you can also use the Datasheet view to enter values directly into your table.

Looking at Figure 2.4, you can see that I've added three fields by entering the field names into the column headers and assigned the appropriate data types to my new table that will ultimately become the physical embodiment of my `Students` table.

Once your table has been defined with fields, you will name and save your table by clicking on the Save icon from the Quick Access toolbar also shown in Figure 2.4. When the Save icon is clicked, you will be prompted to name your table or other Access object, as shown in Figure 2.5.

Click the Save icon to save your newly created table

Change a field's data type



**FIGURE 2.4**

Adding fields to a new table.

**FIGURE 2.5**

Saving and naming a newly created table.



Clicking the Save icon from the Quick Access toolbar, or clicking the Save icon from the Microsoft Office button only saves the selected (highlighted) table or Access object. To save a newly created table or object, ensure it's first selected, then save or right-click the tab of the newly created object in Datasheet view, and select the Save option. Access will also prompt you to save all unsaved objects when closing the current database or the entire Access application.

Now that I've saved and named my Students table, it's worth taking a glance back at Figure 2.4 to discuss the first field in the Students table called ID. Each time you create a new table in Access 2007, it automatically inserts an AutoNumber data type field that will act as your primary key. If you prefer not to use the auto-generated ID field, you can change it or remove it by switching to Design view. Speaking of data types, let's now take a look at the various data types that can be assigned to fields in Access 2007, as shown in Table 2.8.



Trying to insert data greater than the maximum size allowed in a Text field type (255 characters) will generate a database error. If your Text field storage needs require greater length, use the Memo field type.

To better understand data types and their use, see if you can match the following data descriptions to an Access data type (shown in Table 2.8).

**TABLE 2.8 ACCESS 2007 DATA TYPES**

<b>Data Type</b>	<b>Description</b>
Text	Used for storing alphanumeric data such as text and numbers. The Text data field can store up to 255 characters.
Memo	Used for storing alphanumeric data. The Memo data field can store up to 2GB of data (the size limit for an Access 2007 database).
Number	Stores numbers with a controllable size from 1 to 16 bytes. Can also be used for storing primary keys.
Date/Time	Stores dates and times
Currency	Stores numbers and prevents rounding off during calculations.
AutoNumber	An autogenerated number whose values are often used as the table's primary key.
Yes/No	Stores Boolean values (true/false). Access 2007 uses –1 for true values and 0 for no values.
OLE Object	Used for storing images, documents, and other objects from Microsoft Office and other Windows-based programs and can store up to 2GB of data (the size limit for an Access 2007 database). To render a stored OLE object, you must have an OLE server registered on the computer that runs your database. As a general rule, use the Attachment data type instead, which is more efficient and does not require an OLE server.
Hyperlink	Used to store up to 1GB of web address data.
Attachment	New to Office Access 2007, the Attachment data type can store any file type such as images, documents, spreadsheets, and many more. Moreover, the Attachment data type allows you to view and edit the attached files, which is a much needed improvement over the OLE Object data type.

- 1 An employee's social security number.
- 2 The start time of a test.
- 3 The number of employees in a company.
- 4 The annual budget amount allocated to each department in a company.
- 5 Determines if a user is currently logged into a system.
- 6 Stores the address of homes.
- 7 The textual content of an essay.
- 8 An ID generated each time a new user is created.
- 9 The cost of a book.
- 10 An Excel spreadsheet.

The correct data types for the previous storage needs are listed here:

- 1 Text (If dashes are used when entering data)
- 2 Date/Time
- 3 Number
- 4 Currency
- 5 Yes/No
- 6 Text
- 7 Memo
- 8 AutoNumber
- 9 Currency
- 10 Attachment

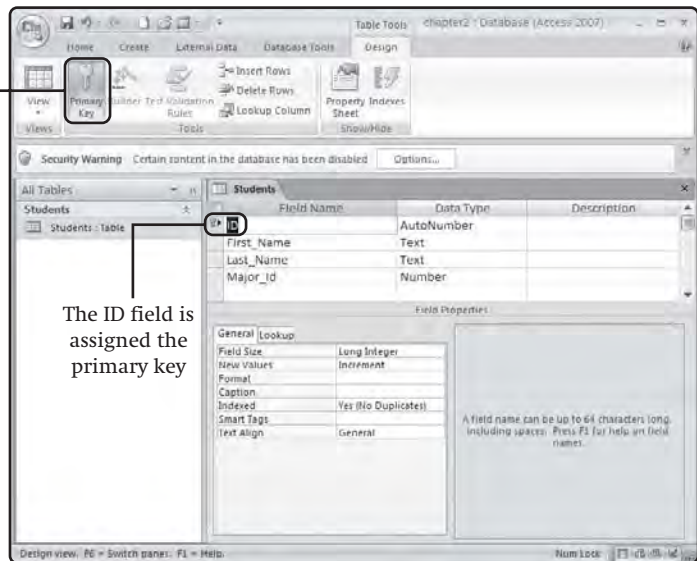
Depending on the data type, each field in a table has a number of attributes such as field size, caption, required input, validation rules, and many others that can be preset and/or updated in Design view, as shown in Figure 2.6.

Use the Primary Key icon to assign a field as primary key

The ID field is assigned the primary key

**FIGURE 2.6**

Viewing a field's attributes in Design view.



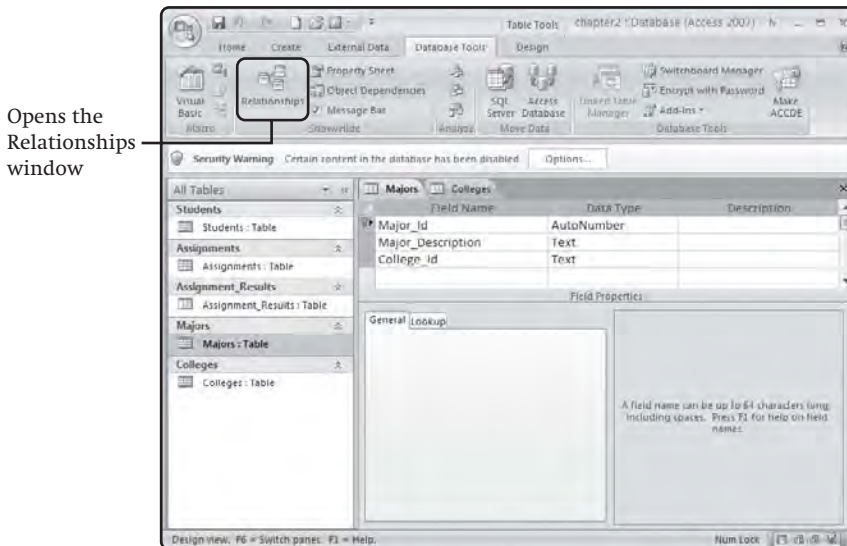
Note the presence of the key symbol next to the ID field in Figure 2.6. The key symbol denotes that field has been assigned as the primary key for the given table. The primary key assignment can be added, changed, or removed by either right-clicking on the field or selecting the Primary Key icon on the Design view tab of the Ribbon. Also, you may remember from the “Database Normalization” section that I chose the Student\_Id (something similar to a social

security number) as the primary key for the *Students* table. In theory, something similar to a social security number will work as a primary key because of its uniqueness, but in practice an auto-generated number such as the one generated by Access's *AutoNumber* data type is a much better choice for the primary key because social security numbers are generally considered private information and wouldn't be appropriate for use in public queries and reports.

The remainder of the tables and fields that will be used in our chapter-based database will be referenced throughout this chapter and can also be found on this book's accompanying website ([www.courseptr.com/downloads](http://www.courseptr.com/downloads)).

## TABLE RELATIONSHIPS

Remembering that table relationships are the essence of a relational database such as Access, we must create physical relationships between our tables to bring data back together that we have separated during the database normalization process. Let's get started by creating a relationship in Access between our two tables, *Majors* and *Colleges*. We'll accomplish this through the Relationships window found under the Database Tools tab of the Ribbon, as shown in Figure 2.7.



**FIGURE 2.7**

Creating table relationships using the Relationships window.

Once the Relationships icon is selected, a Show Table window is launched, shown in Figure 2.8, that allows you to select the tables and/or queries that will be used to create the relationship. In my example, I will select both the *Majors* and *Colleges* tables to create a relationship and then click *Add*.



**FIGURE 2.8**

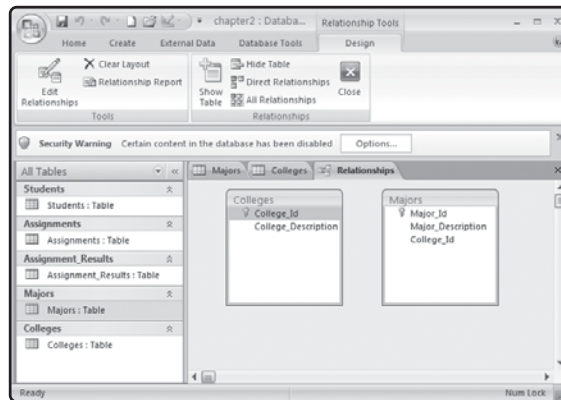
Selecting tables to create a relationship.



You will now see the Relationships window opened with your selected tables, as revealed in Figure 2.9.

**FIGURE 2.9**

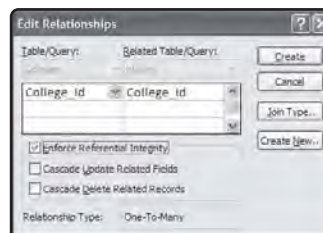
The Relationships window.



From the Relationships window I will click a primary key from one table and drag it to the foreign key in the other table. To drag more than one field, hold the Ctrl key while selecting each field then drag. For my purposes, I want to create a relationship between the two tables (Majors and Colleges) using the College\_Id key from each table. When I drag the College\_Id from the Colleges table onto the College\_Id from the Majors table, a new window opens to edit the relationship, as shown in Figure 2.10.

**FIGURE 2.10**

The Edit Relationships window.

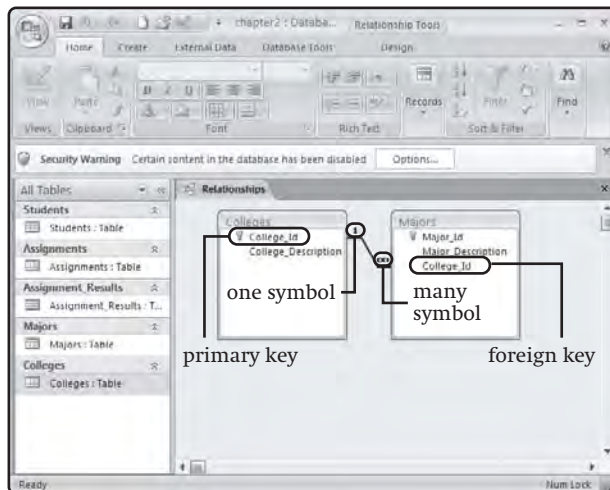


Click the Enforce Referential Integrity check box to ensure the primary key exists when entering a foreign key and then click the Create button. I have now created a one-to-many relationship, which means for every `College_Id` in the `Colleges` table, there are many occurrences of that `College_Id` in the `Majors` table (in other words, each college has many majors). Moreover, the primary key in the `Colleges` table, `College_Id`, has also now become a foreign key in the `Majors` table.

Consider the following options from the Edit Relationships window when creating a relationship between tables:

- Enforce referential integrity, which means values entered into the foreign key must match values in the primary key.
- Enforce cascading updates between one or more tables, which means related updates from one table's fields are cascaded to the other table or tables in the relationship.
- Enforce cascading deletes between the two tables. In short, this means any relevant deletions from one table cascade in the other table(s).
- Both cascading updates and deletes help enforce referential integrity.

As shown in Figure 2.11, my Relationships window will now have a graphical link depicting a one-to-many relationship between my two tables and their keys.

**FIGURE 2.11**

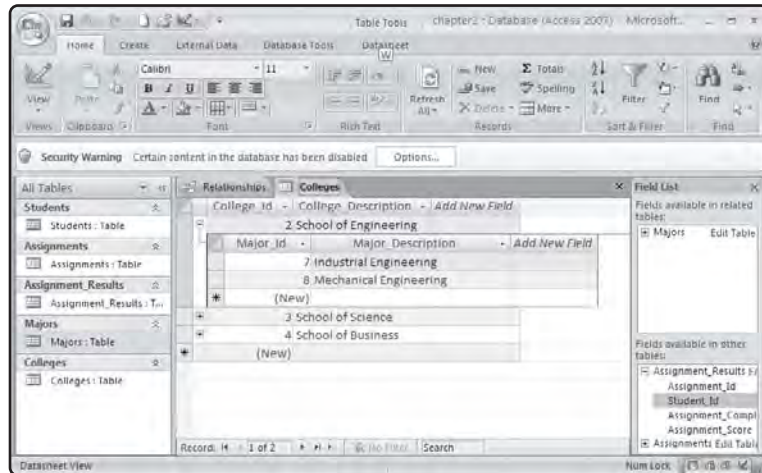
A table relationship established using primary and foreign keys.

To better visualize the relationship, I'll enter a few records into the `Colleges` table via the Datasheet view. You will notice that for each record in the `Colleges` table a small plus sign (+) exists to the left of the row. Shown in Figure 2.12, I expanded the plus sign by clicking it for

one row, which allows me to enter records directly into the Majors table via a one-to-many relationship.

**FIGURE 2.12**

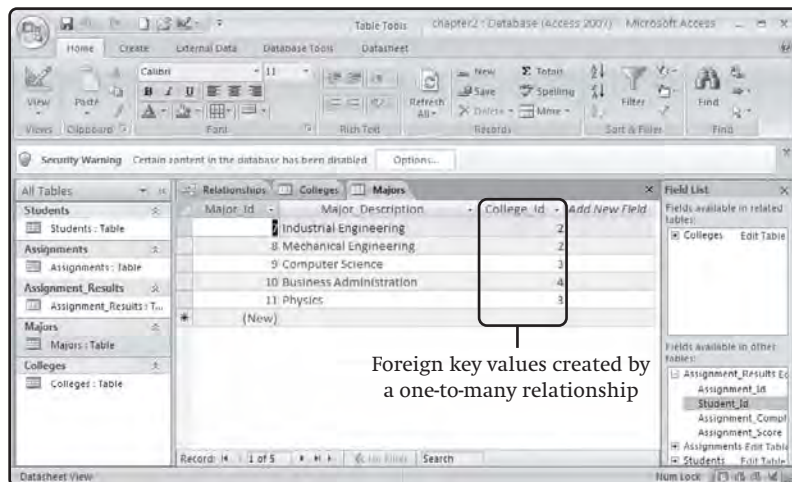
Entering records into tables via a one-to-many relationship.



Shown in Figure 2.13, I opened my Majors table in Datasheet view and it reveals the records I entered earlier via the Colleges table. Also important are the College\_Id values in the Majors table, courtesy of my one-to-many relationship, which have become foreign key values back to the Colleges table. Very cool!

**FIGURE 2.13**

Newly created records via a one-to-many relationship.

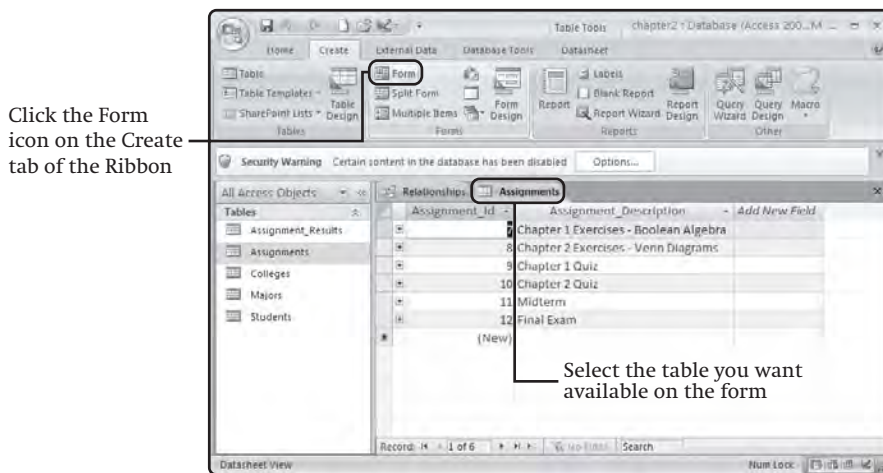


## FORMS

Though careful data analysis is the key to a well-designed and adaptive database, end users are just as likely to appreciate a well-built user interface time and again. A good interface leverages forms and controls in a way that is intuitive to users for managing data. User interfaces should hide the complexities of a database, such as business rules and relationships.

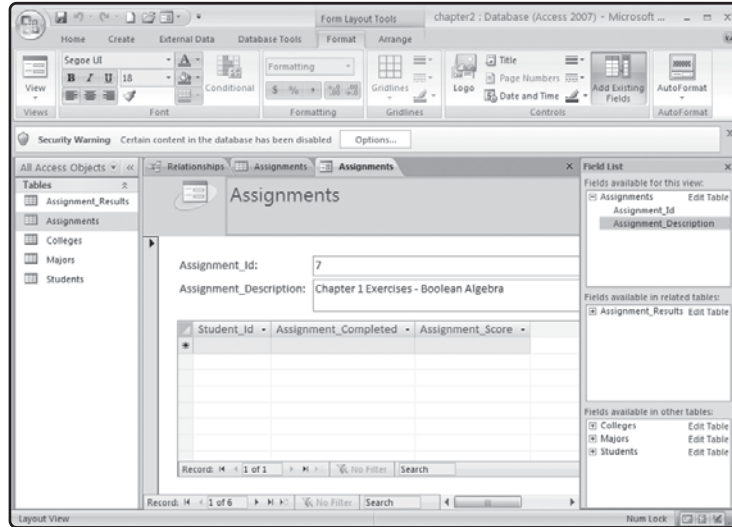
Access forms are graphical controls that act as containers for other graphical controls such as text boxes, labels, and command buttons. Though most of the forms in this book will be created from scratch without the aid of any wizard, Access 2007 has done an excellent job making simplistic form creation easy through the use of the Form tool, which I'll show you next.

To use the Form tool, select the table or query you want available on the form (in my example it's the **Assignments** table) and then click the Form icon from the Create tab of the Ribbon, as shown in Figure 2.14.

**FIGURE 2.14**

Creating a new form in Access using the Form tool.

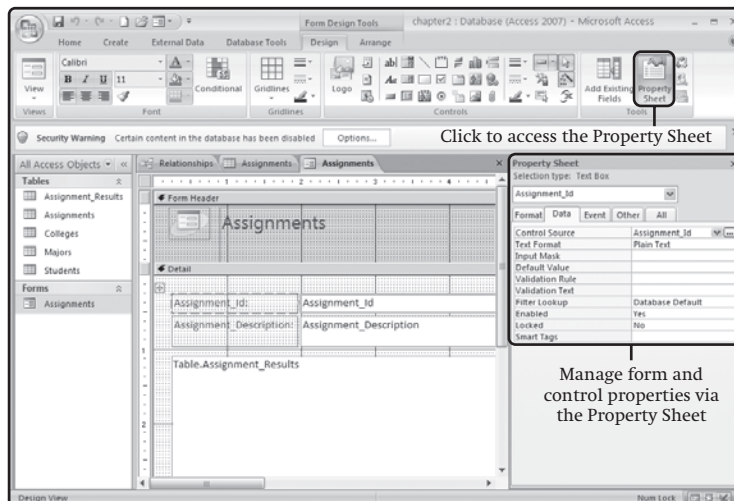
The Form tool automatically creates a new form with all the necessary controls to support data from the selected query or table in Layout view. While in Layout view you can scroll through the data and adjust the controls, such as the placement and size of a text box. If your table has a one-to-many relationship to another table, Access places a datasheet onto the new form as it did for my **Assignments** table, which has a one-to-many relationship with the **Assignment\_Results** table, as revealed in Figure 2.15.

**FIGURE 2.15**

Viewing a new form created by the Form tool.

After creating a form, ensure you save it by clicking the Save icon in the Quick Access toolbar, or from the Office button; you can also leverage the keyboard combination Ctrl + S to save the selected or highlighted objects.

Both forms and controls have properties that describe how they look and behave (attributes). You can manage these attributes in Design view using the Property Sheet window shown in Figure 2.16.

**FIGURE 2.16**

Using the Property Sheet to manage form and control attributes.

Manage form and control properties via the Property Sheet

If you're unable to see the Property Sheet window, ensure you're in Design view and then choose the Property Sheet icon from the Design tab shown in Figure 2.16.

It's important to note, not all control properties are accessible or are available through the Property Sheet in Design view. This means other properties are available only during runtime through VBA statements.

## Common Controls

Controls can be accessed and placed onto your form manually by clicking the control in the Controls section of the Design tab. Access places the control on your form in a predetermined shape and size. You can resize controls using your left mouse button, or by clicking the control and then using the Shift and arrow keys combination to resize the control. Use the Ctrl button and arrow keys simultaneously to move the control.

Just like data fields, controls have properties that determine the control's various attributes or settings. A control's properties can be accessed by right-clicking the control and selecting Properties from the menu.



Both the Field List and Property Sheet windows can be moved in and out of the main Access window. Both windows can also be docked in and around Access by grabbing the top portions of either window and dragging them to the top, bottom, or sides of the main Access window.

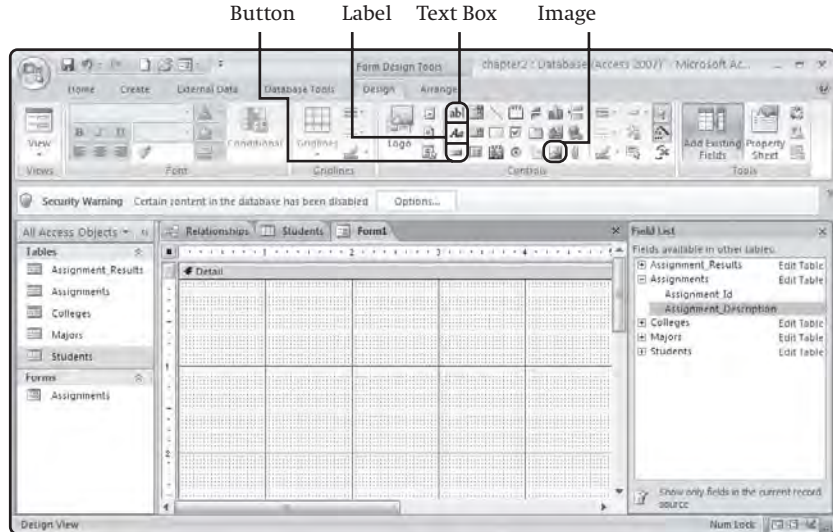
I'll now show you four common controls: text boxes, labels, images, and buttons. You can see in Figure 2.17 a visual depiction of where these common controls are located in the Controls section of the Design tab.

To get started working with these common controls, I will create a new form called *Manage Students* from scratch using the Blank Form option that will allow a user to add, update, and delete students from my database. By default, Access puts you directly in Layout view after creating a new table. When creating a form from scratch, it's better, in my opinion, to switch directly to Design view. Design view provides, among other features, gridlines that will assist you in aligning and sizing your controls for a uniform look.

As shown in Figure 2.18, I have created my new *Manage Students* form from scratch and have switched to Design view revealing the grid lines.

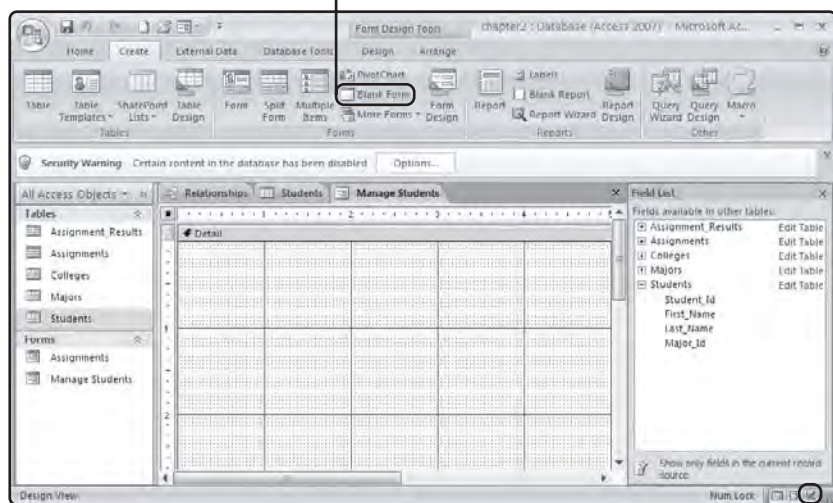
You may have noticed previously that the Form tool automatically bound both form and controls (text boxes) to my *Students* table. I'll now show you how to create bound controls from scratch. First, I need to bind my *Manage Students* form to the *Students* table by setting



**FIGURE 2.17**

Common controls including text box, label, image, and button.

Create a form from scratch using the Blank Form option

**FIGURE 2.18**

Creating a form from scratch using the Blank Form option.

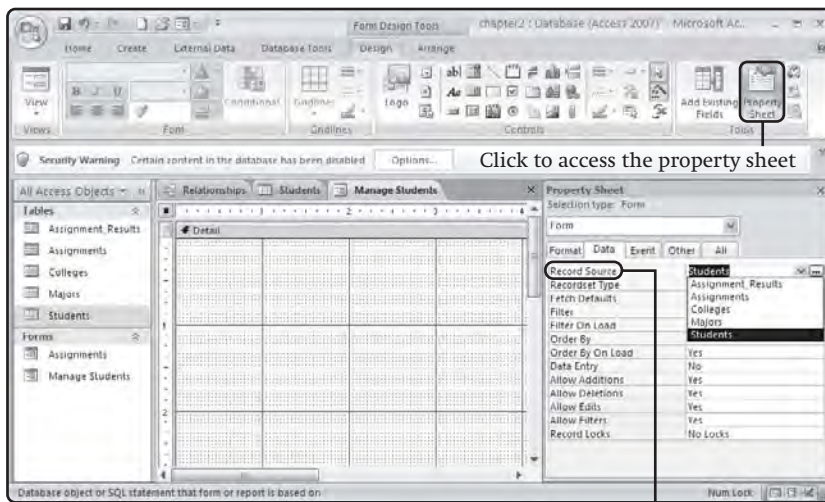
Switch to Design view to reveal the gridlines

the form's Record Source property (by selecting it from the Data tab in the Property Sheet window) to the Students table, as shown in Figure 2.19.

After binding my Manage Students form to the Students table, I now have the ability to bind other controls, such as text boxes, to the fields in the Students table. First I will place one label control at the top of the form by clicking the Label Control icon from the Controls section of

the Ribbon and dropping it onto the form. This label control is for display purposes only and will not be bound to any data field. Next, I will set the following label property values via the All tab of the Property Sheet window:

- Name: lblManageStudents
- Caption: Manage Students
- Font Size: 14
- Text Align: Center



Bind the form to a table by setting the form's Record Source property

**FIGURE 2.19**  
Binding the  
Manage  
Students form to  
the Students  
table.

Remember that the Property Sheet window displays properties for a single control, unless you select multiple controls to view common updatable properties. To switch between control properties, use the drop-down box at the top of the Property Sheet window.

Below the label, I will add three text boxes by clicking on the text box control from the Controls section of the Ribbon and align them on the form, one below the other. Notice after adding a text box control that you actually get two controls for the price of one. Specifically, you get one text box and one label control just to the left. Both controls have their own distinct properties and can be moved and deleted independently of each other.

Assign the following properties to each respective text box via the All tab of the property sheet:

- Name: txtStudentId
- Control Source: Student\_Id



- Locked: Yes
- Name: txtFirstName
- Control Source: First\_Name
- Locked: No
- Name: txtLastName
- Control Source: Last\_Name
- Locked: No

Note that I've set the txtStudentId Locked property value to Yes. I did this because the txtStudentId field is a primary key with a data type of AutoNumber. Since an AutoNumber is generated automatically by the Access system, it is neither necessary nor advisable to let users manage this field manually.

Assign the following properties to each respective text box label via the All tab of the property sheet:

- Name: lblStudentId
- Caption: Student Id:
- Text Align: Right
- Name: lblFirstName
- Caption: First Name:
- Text Align: Right
- Name: lblLastName
- Caption: Last Name:
- Text Align: Right

I'll now add one image control to the upper-right corner of the form and assign the following property values. I've copied the image used in this control to the book's website ([www.courseptr.com/downloads](http://www.courseptr.com/downloads)) for your convenience.

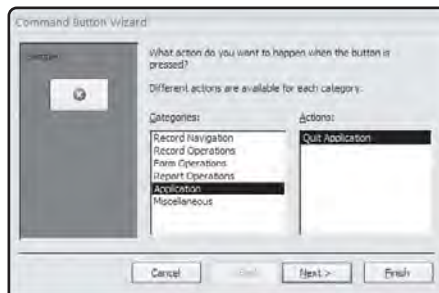
- Name: imgTeacher
- Picture: teacher.JPG

Last but not least, I'll add a button to my new form that exits the user from the application and assigns the following property values:

- Name: cmdQuit
- Caption: Quit Application

When you add a command button to any form, Access launches a wizard that can aid you in assigning various built-in actions to your command button's click event. Though I typically shy away from wizards in this book, this is a good example of when using one is helpful in assigning a simple action to a control.

As seen in Figure 2.20, I have selected the Application Category and Quit Application Action from the Command Button Wizard.

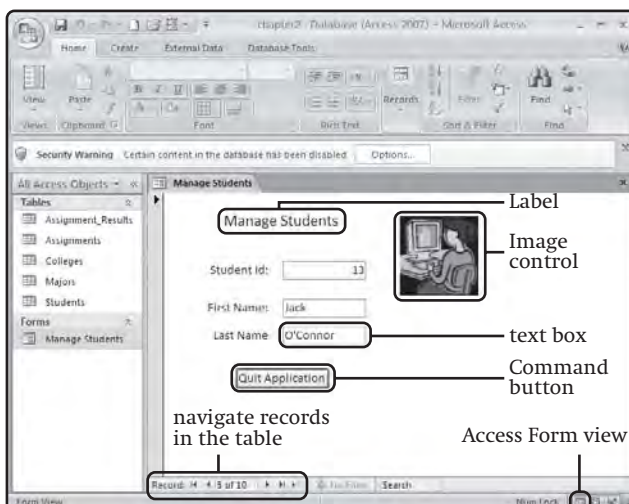


**FIGURE 2.20**

Using the Command Button Wizard to assign an action to a button.

After updating all control property values, I will resize them by left-clicking one of the control's re-sizing boxes (re-sizing boxes are the small boxes in and around a control's outer boundary) and dragging a side or corner, and I will re-position them by left-clicking the control and dragging it to its appropriate location.

Now that the controls have been bound to the *Students* table, users can view, update, and add student records using the form's built-in navigation bar shown in Figure 2.21.



**FIGURE 2.21**

Managing student information with bound controls.

You have now seen how to create a table from scratch and bind controls to data fields by hand. You can of course add bound controls to forms created by scratch by dragging and dropping table fields from the Field List window, but that wouldn't give you the necessary insight into how controls are bound! Nevertheless, the Field List window is available by clicking the Add Existing Fields icon in the Tools section of the Design tab.

In future chapters I will show you how to further control the relationship between controls, tables, and fields by harnessing the power of the Access VBA language.

## Hungarian Notation

You may have noticed the naming convention I used to assign control names in the previous section. I recommend using a naming convention for controls to provide readability and consistency throughout your database and code.

The naming convention I use and recommend is called Hungarian Notation (named after a computer scientist). To apply this notation, simply modify the Name property of each control to use a three-letter prefix (all in lowercase) that indicates the control type followed by a meaningful description. Each word that describes the control (not the prefix) should have its first letter capitalized.

Table 2.9 shows some sample naming conventions for controls discussed throughout this book.

**TABLE 2.9** COMMON CONTROL-NAMING CONVENTIONS

Control	Prefix	Example
Check Box	chk	chkRed
Combo Box	cbo	cboStates
Command Button	cmd	cmdQuit
Form	frm	frmMain
Image	img	imgLogo
Label	lbl	lblFirstName
List Box	lst	lstFruits
Option Button	opt	optMale
Text Box	txt	txtFirstName

## QUERIES

Database queries provide the mechanism by which people can question their data and get responses. Database queries are typically written using a database language called SQL (Structured Query Language), which I discuss in Chapter 9, “Microsoft Access SQL.” Nevertheless, I’ll show you how to create queries using Microsoft’s built-in query designer, which is sufficient for most database queries in Access.

To get started, I will ask my database for the first and last names of all students in my *Students* table. First, I will click the Query Design icon from the Other section in the Create tab of the Ribbon. Next, a Show Table window is opened where I will select and add my *Students* table, as shown in Figure 2.22



**FIGURE 2.22**

Select and add a table to a query.

Access has now created an empty query object for me in Design view with the *Students* table available for field selection. I will now select both last- and first-name fields, displaying them in ascending order. This can be accomplished in one of three ways: Double-clicking one or more fields in the *Students* table, selecting one field at a time in each Fields drop-down box in the matrix at the bottom of the window, or by dragging and dropping a field from the graphical depiction of the table. After selecting the *Last\_Name* and *First\_Name* fields, your query should look like Figure 2.23.

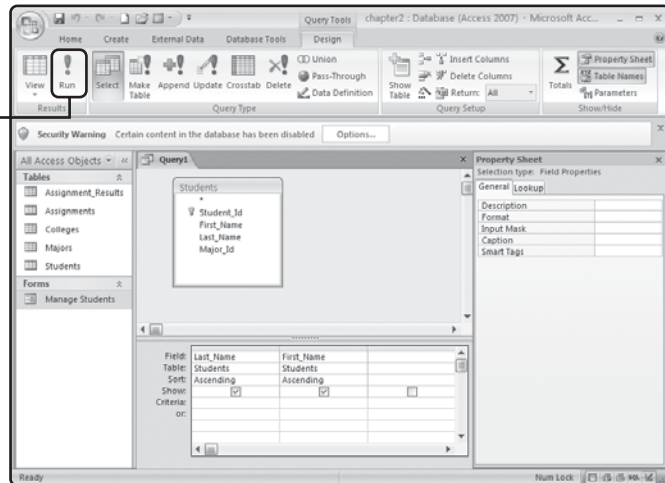


You can easily select all fields in a table for a query by selecting the asterisk (\*) character at the top of a table.

Access allows you to verify the underlying SQL query generated by the Access query designer by selecting the SQL View item from the View menu. The SQL query generated for my *students* query looks like this:

```
SELECT Students.First_Name, Students.Last_Name
FROM Students
ORDER BY Students.First_Name, Students.Last_Name;
```

Execute  
your query

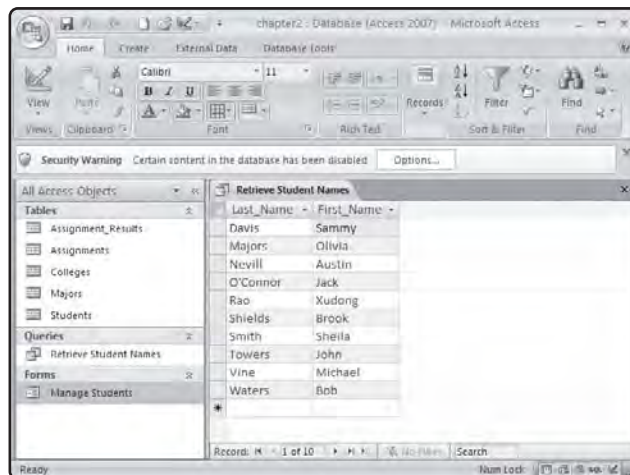


**FIGURE 2.23**

Selecting fields from a table to build a query.

I will now save my query and name it Retrieve Student Names.

To execute or run a query, simply click the exclamation mark (!) icon in the Results area of the Design tab. Results from my Retrieve Student Names query are shown in Figure 2.24.



**FIGURE 2.24**

Viewing the results of the Retrieve Student Names query.

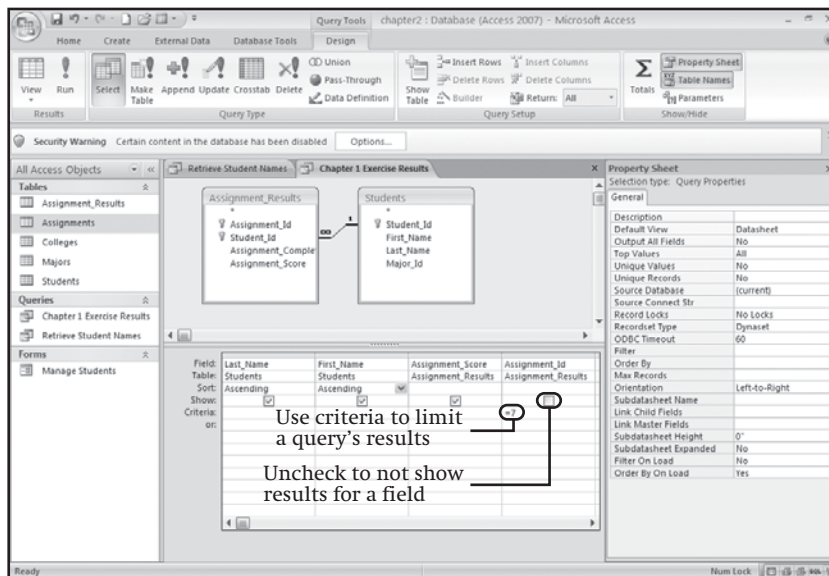
I'll now create a new query that asks a more pointed question of my student data. Specifically, my new query should say, "Give me a list of all student names and their grades for the 'Chapter 1 Exercises - Boolean Algebra' assignment (Assignment\_Id=7)." I will accomplish this by creating a new query called Chapter 1 Exercise Results and selecting both the Students and Assignment\_Results tables.

Notice in Figure 2.25 that Access displays the relationship between both tables by displaying a graphical link between the Student\_Id primary and foreign keys.

To build the query, select the following fields in the following order by double-clicking each field from the corresponding table in the query designer.

- Last\_Name (ascending order)
- First\_Name (ascending order)
- Assignment\_Score
- Assignment\_Id

My query is almost done, but not yet—I still need to tell the query to limit the results to show only assignment scores for assignment number 7. I can accomplish this by specifying criteria (or a condition) of =7. Moreover, since the criteria is applied against the Assignment\_Id field, which I don't want displayed in the query results, I'll tell the query designer to not show the field by unchecking the Show check box. The finished query is depicted in Figure 2.25.



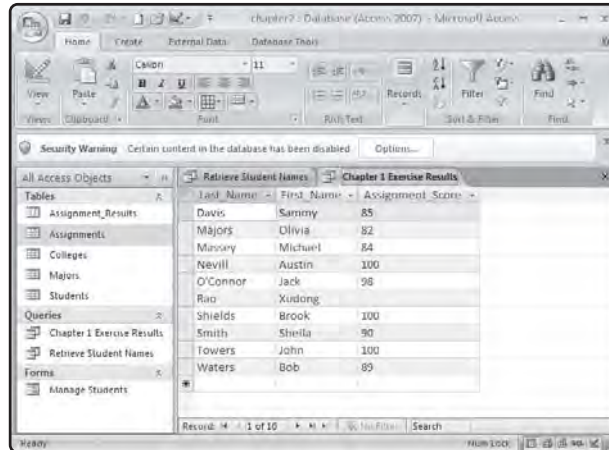
**FIGURE 2.25**

Creating an Access query with multiple tables and a condition.

After saving and naming my query, I run (execute) it with results, as shown in Figure 2.26.

**FIGURE 2.26**

Viewing the  
Chapter 1  
Exercise  
Results query  
results.



Access has limited query results based upon my criteria to show all students and their scores where Assignment\_Id=7.

Though I discuss SQL statements later in the book, I would like you to appreciate what happened behind the scenes in the Chapter 1 Exercise Results query by looking at the following SQL statements that Access created.

```
SELECT Students.Last_Name, Students.First_Name,
Assignment_Results.Assignment_Score
FROM Students INNER JOIN Assignment_Results ON Students.Student_Id =
Assignment_Results.Student_Id
WHERE (((Assignment_Results.Assignment_Id)=7))
ORDER BY Students.Last_Name, Students.First_Name;
```

As you can see from the previous SQL query, Access has done a lot of work behind the scenes in creating the necessary SQL joins and conditions.

## SUMMARY

- Database normalization is the process by which one optimizes table structures to enhance query performance and eliminate data integrity issues by investigating data requirements and their relationships to each other.
- An entity is generally described as the overall description of a set of data to be saved in a table.

- Attributes are the bits of related information that make up an entity.
- A primary key is either a single field or a combined set of fields that uniquely identifies a single row of data in a table.
- The foreign key creates the foundation of a relationship between two tables by inserting the primary key from one table into another table.
- A one-to-one relationship denotes that each row of information for one entity relates to exactly one row of information for another entity.
- A one-to-many relationship is created by adding the primary key from one table into a second table that will hold many occurrences of the primary key.
- To build a many-to-many relationship, you need to create a third table that breaks the many-to-many relationship into two one-to-many relationships.
- Access tables must contain at least one or more fields.
- The `AutoNumber` data type is an autogenerated number whose values can be incremented or randomly generated. It is often used as the table's primary key.
- If you require a number of characters greater than 255 (the maximum size allowed in a Text field), you need to use the `Memo` field type.
- Referential integrity means values entered into the foreign key must match values in the primary key.
- Access forms are graphical controls that act as containers for other graphical controls such as text boxes, labels, and command buttons.
- Controls have properties that determine how the control will look and behave.
- Access allows you to bind controls to tables, fields, and queries.
- Database queries provide the mechanism by which people can ask their data questions and get responses.
- You can limit the result set of a query using criteria, also known as conditions.



## PROGRAMMING CHALLENGES

1. Create a new table called **Teachers** that holds the teacher's name and a teacher id (primary key). Assign appropriate data types to each field.
2. Create a new table called **Courses** that holds a course description, course id (primary key), and another field to hold a teacher id. Assign appropriate data types to each field.
3. Create a one-to-many relationship that links the **Teachers** table to the **Courses** table and remember to enforce referential integrity.
4. Create a new form called **Manage Teachers** that allows a user to manage both the **Teachers** and **Courses** table. Using the form, enter sample records into both the **Teachers** and **Courses** tables. Make sure to leave one or more teacher record that does not have a corresponding entry in the **Courses** table.
5. Using the data entered into the **Teachers** and **Courses** tables from challenge 4, create a new query called **Courses without Teachers** that will list all courses without an assigned teacher.



# INTRODUCTION TO ACCESS VBA

**L**ike many professional RDBMS (Relational Database Management Systems), Microsoft Access comes with its own programming language called VBA or Visual Basic for Applications. Though VBA supports the look and feel of Microsoft's Visual Basic, it is not Visual Basic nor is it Visual Basic .NET. Access VBA is specifically designed for Microsoft Access. This means it has knowledge of and support for the Microsoft Access object model. The concept of an object model is different for each Microsoft Office application. For example, both Microsoft Excel and Microsoft Word support VBA, but each application has its own object model.

## THE EVENT-DRIVEN PARADIGM

The *event-driven paradigm* is a powerful programming model that allows programmers to build applications that respond to actions initiated by the user or system. Access VBA includes a number of events that are categorized by the objects they represent. VBA programmers write code in event procedures to respond to user actions (such as clicking a command button) or system actions (such as a form loading).

To demonstrate the event-driven model, consider a form, which has a corresponding Form object that contains many events such as `Click`, `Load`, and `MouseUp`. As seen next, both `Click` and `MouseUp` events are triggered by the user performing an action with the mouse on the form.

```
Private Sub Form_Click()  
    'write code here to respond to the user clicking the form  
End Sub  
Private Sub Form_MouseUp(Button As Integer, _  
    Shift As Integer, X As Single, Y As Single)  
    'write code here to respond to the user releasing a mouse button  
End Sub
```

You, the VBA programmer, write code in these event procedures to respond to user actions. Moreover, events can be triggered by the system or the program itself. For example, the Load event shown next is triggered when a form's Form object is first loaded into memory.

```
Private Sub Form_Load()  
    'write code here to respond to the form loading into memory  
End Sub
```

If you're new to event-driven programming, this may seem a bit awkward at first. I promise you, however, it is really not that difficult. In fact, VBA does a great job of providing much of the detail for you. By the end of this chapter, you will be writing your first Access VBA event-driven programs with ease.

## OBJECT-BASED PROGRAMMING

The key to programming in VBA is using objects. Objects have properties that describe the object and methods that perform actions. For example, say I have an object called `Person`. The `Person` object contains properties called `HairColor`, `Weight`, `Height`, and `Age` that describe the object. The `Person` object also contains methods that describe an action the object can perform such as `Run`, `Walk`, `Sleep`, and `Eat`. As you can see, understanding the concept of objects is really quite simple!

Many Access VBA objects also contain data structures called collections. In a nutshell, *collections* are groupings of objects.

Access VBA supports many objects, such as the `Form` object, which is simply a window or dialog box. The `Form` object contains many properties such as `Caption`, `Moveable`, and `Visible`. Each of these properties describes the `Form` object and allows VBA programmers to set characteristics of a user's interface. Like the object `Person`, the `Form` object contains methods such as `Move` and `Refresh`.

Many objects share common characteristics such as properties and methods. To demonstrate, the `Label` object (which implements a label control) shares many of the `Form` properties, such as `Caption` and `Visible`.

Properties and methods of objects are accessed using the dot operator (.), as demonstrated in the next two VBA statements.

```
Label1.ForeColor = vbBlue  
Label1.Caption = "Hello World"
```

Realize that properties such as `ForeColor` and `Caption` belong to the `Label1` object, and they are accessed using the dot operator. I discuss this in more detail in sections to come.

## THE VBA IDE

If you've written programs in Visual Basic or VBA before, the Access 2007 VBA *integrated development environment (IDE)* should feel very familiar to you. If not, don't worry—the VBA IDE is user-friendly and easy to learn. For ease of use, I will refer to the VBA integrated development environment simply as the *Visual Basic Editor*, or *VBE*, from now on.

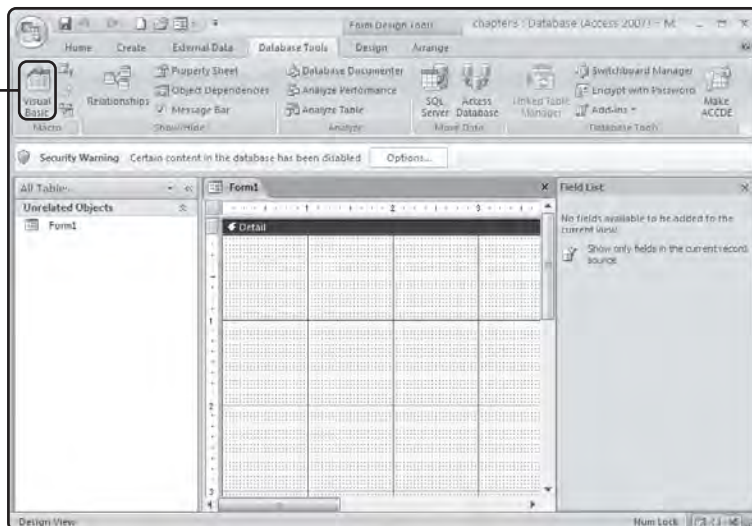
The VBE contains a suite of windows, toolbars, and menu items that provide support for text editing, debugging, file management, and help. Two common ways for accessing the VBE is with forms and code modules.

After adding and saving a form to your database, make sure your form is highlighted (selected) in Design view, and then click the Visual Basic icon in the Macro area of the Database Tools tab of the Ribbon, as shown Figure 3.1.



An easy shortcut to opening the VBE and alternating between Access and the VBE is by pressing **Alt+F11**.

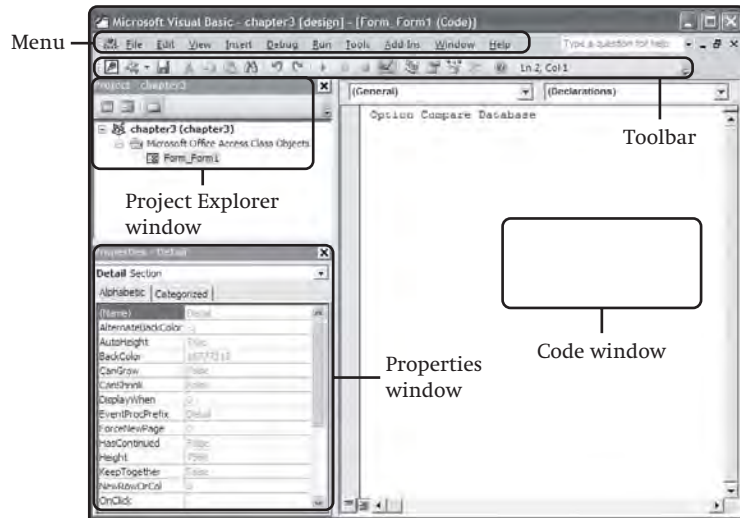
Open the Visual Basic Editor, or VBE



**FIGURE 3.1**

Selecting the Visual Basic Editor or VBE.

After selecting the Visual Basic icon, the VBE should open up in a separate window, similar to the one shown in Figure 3.2.



**FIGURE 3.2**

Opening the Visual Basic Editor or VBE for the first time.

The first time you open the VBE, the Project Explorer and Properties windows may not be visible, but they can be accessed from the View menu.

There are a few VBE components you should familiarize yourself with right away. Each is described here and shown in Figure 3.2.

- **Toolbars:** Toolbars contain shortcuts to many common functions used throughout your VBA development, such as saving, inserting modules, and running your program code. Additional toolbars can be added from the View menu.
- **Menus:** Menus in the VBE provide you with many development features, such as file management, editing, debugging, and help.
- **Project Explorer window:** The Project Explorer window provides you with a bird's-eye view of all files and components that build your Access VBA programming environment. Notice in Figure 3.2 that my form's name (Form\_Form1) appears under the Microsoft Office Access Class Objects folder. If I had multiple forms in my database, there would be multiple form names in this folder. Remember, Microsoft Access stores all components including forms, queries, reports, and modules in a single .accdb file.

- **Properties window:** The Properties window shows all available properties for the object selected in the list box. Most importantly, the Properties window allows you to change the values of an object's property during design-time development.
- **Code window:** The Code window is where you enter your VBA code and find procedures and event procedures for objects using the two list boxes at the top of the window.

If you haven't done so yet, explore each of the previously mentioned components and windows so that you are comfortable navigating the VBE environment.

## Introduction to Event Procedures

Procedures are simply containers for VBA code. Access VBA contains four types of procedures:

- Subprocedures
- Function procedures
- Property procedures
- Event procedures

Each type of procedure is designed to accomplish specific tasks. For example, event procedures are designed to catch and respond to user-initiated events such as a mouse click on a command button or a system-initiated event such as a form loading. In this section, I concentrate on event procedures because they are the foundation for an event-driven language such as VBA. In subsequent chapters, you learn about other types of procedures in detail.

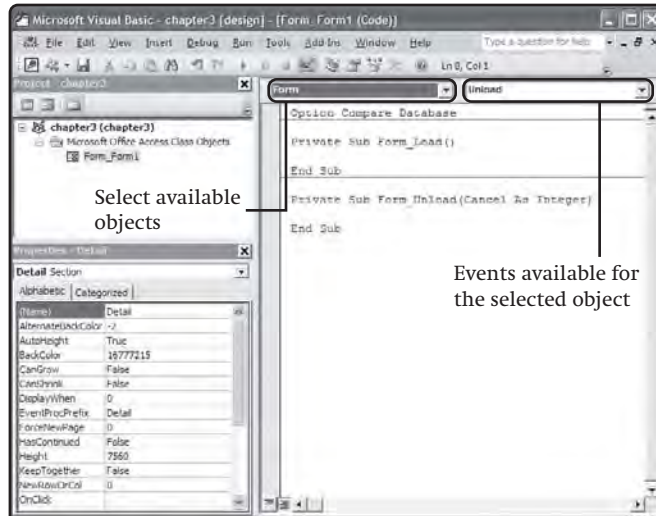
As mentioned, objects such as the `Form` object contain methods and properties. They also contain specialized events that are automatically provided after the object has been added to your database. VBA takes care of naming your object's events for you. The naming convention follows.

`ObjectName_EventName`

For example, a form added to your Access database called `Form1` has a number of events, including the following:

```
Private Sub Form_Load()  
End Sub  
Private Sub Form_Unload(Cancel As Integer)  
End Sub
```

Notice the naming convention used for each event procedure: object name followed by the event name with an underscore in between. The objects and their events in Figure 3.3 are accessed from the VBE Code window.



**FIGURE 3.3**

Accessing an object and its associated events in the VBE.

The leftmost list box in the Code window identifies available objects. The rightmost list box contains all available events for the object selected in the left list box. Each time you select an event, VBA creates the event shell for you automatically. This saves you from having to manually type each event's beginning and ending procedure statements.

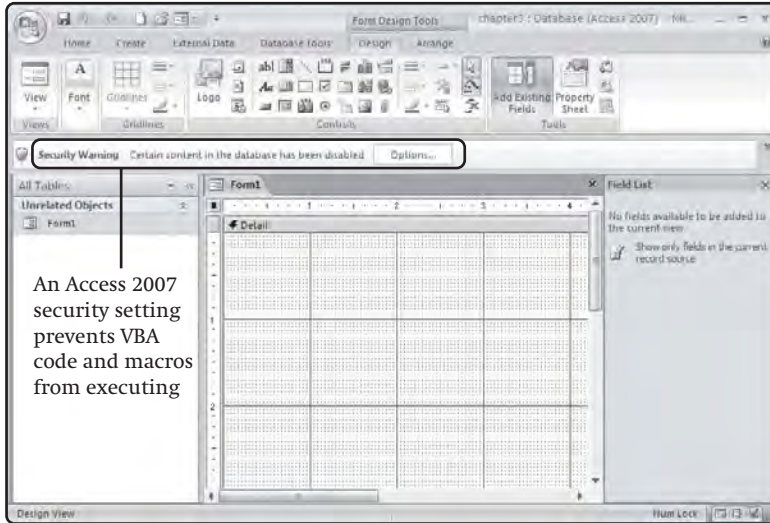


Each procedure in the VBE Code window is separated by a horizontal line.

Keep in mind that empty event procedures serve no purpose until you write code in them using VBA statements. More importantly, VBA procedures will not run if Access has blocked VBA code and macros from executing, which is the default setting after installing Access or Office 2007. You may remember seeing such a security warning in prior screen shots, but I've highlighted it in Figure 3.4.

To run your VBA code, you must first change the security setting to allow VBA Macro content, which is accomplished by clicking the Options button in the Security Warning banner. After which a new Security Options window is displayed where you can change the security setting to enable the VBA Macro content, as shown in Figure 3.5.





An Access 2007 security setting prevents VBA code and macros from executing

**FIGURE 3.4**

Access 2007 security setting that prevents VBA code from running.



**FIGURE 3.5**

Changing a security setting to enable VBA Macro content.

Type the keywords “security” or “trust center” in the Access 2007 Help window for more information about Office 2007 security settings.

## Introduction to VBA Statements

VBA *statements* are comprised of variables, keywords, operators, and expressions that comprise an instruction to the computer. Every VBA statement falls into one of three categories:

- **Declaration statement:** Creates variables, data types, and procedures.
- **Assignment statement:** Assigns data or values to variables or properties.
- **Executable statement:** Initiates an action such as a method or function.



Most VBA statements fit on one line, but sometimes it is appropriate to continue a VBA statement onto a second (or more) line for readability. To split a single VBA statement into multiple lines, VBA programmers use the concatenation character (&) and the line continuation character (\_) separated by a space. To demonstrate, the following assignment statement uses the concatenation and continuation characters to extend a statement across two lines.

```
Private Sub Label3_Click()  
Label3.Caption = "This is a single VBA assignment " & _  
    " statement split onto two lines."  
End Sub
```



The term *concatenation* means to glue or put one or more items together.

One of the best ways to provide understandable VBA statements is with comments. Comments provide you and other programmers a brief description of how and why your program code does something. In Access VBA, comments are created by placing a single quote ('), sometimes called a *tick mark*, to the left side of a statement. Comments are also created by placing the keyword REM (short for remark) at the left side of a statement. The following statements demonstrate both ways of creating VBA comments.

```
' This is a VBA comment using the single quote character.  
REM This is a VBA comment using the REM keyword.
```

When a computer encounters a comment, it is ignored and not processed as a VBA statement.

## ACCESSING OBJECTS AND THEIR PROPERTIES

Besides the Properties window, Microsoft Access provides a number of ways to access objects and their properties. Each way provides a level of intricacy and detail while providing specific performance characteristics. In its simplest form, programmers can directly call the name of an object (such as the Form object) or the name of a control (such as a command button). This is only applicable when accessing objects and controls that belong to the current scope of a code module. For example, the next VBA assignment statement updates the form's Caption property during the form's Load event.

```
Private Sub Form_Load()  
    Form.Caption = "Chapter 1"  
End Sub
```

In addition to forms, controls belonging to the current form and scope can be referenced by simply calling their name.

```
Private Sub Form_Load()  
    lblSalary.Caption = "Enter Salary"  
    txtSalary.Value = "50000.00"  
    cmdIncrease.Caption = "Increase Salary"  
End Sub
```

There are times, however, when you need to go beyond the current scope and access forms and controls that do not belong to the current object. There are a number of other reasons for being more specific about what controls you are referencing, including performance considerations and advanced control access techniques such as enumerating. To accomplish these goals, I will show you how to access forms and controls using common VBA techniques with the `Me` keyword prefix and collections such as the `Forms` collection.

## The Forms Collection

Properties of the `Form` object can be accessed in the VBE Code window by simply supplying the form's Access class name.

```
Form_Form1.Caption = "updating the form's caption property"
```

Notice the naming convention used in the keyword `Form_Form1`. When an Access form is created and saved, Microsoft Access refers to it in the VBE as a Microsoft Office Access Class Object with the name `Form` representing the standard object name with a trailing underscore (`_`) followed by the individual form's name. Moreover, you can use the form's Access class name to not only access its own properties, but controls contained on the form. For example, the following VBA assignment statement uses the Access form class name to modify a label's `Caption` property.

```
Form_Form1.Label1.Caption = "update the label's caption property"
```



If your form name contains spaces, you must surround the Access form class name using brackets.

```
[Form_Light Switch].Label1.Caption = "Light Switch"
```

This approach is convenient when working with small VBA projects. At times, however, you want to use a more advanced feature (such as the `Forms` collection) when working with multiple forms or with multiple controls on a form. Access provides the `Forms` collection for specifying which form's `Caption` property you are referencing.

The `Forms` collection contains all open forms in your Access database. To access individual forms in the `Forms` collection, simply supply the `Forms` collection an index or form name as shown in the next statements.

```
' Using an index to refer to a form in the collection.  
Forms(0).Caption = "Chapter 1"  
  
' Using a form name to reference a form in the collection.  
Forms("Form1").Caption = "Chapter 1"
```



Because form indexes can change, it is considered safer to use the form name when accessing forms in the `Forms` collection.

Notice when passing the name of the form to the `Forms` collection, you must surround the form name in double quotes. If the form's name contains one or more spaces, you must use brackets (`[]`) to surround the name. After specifying a form in the `Forms` collection, you can use the dot operator to reference the individual form's properties, such as `Caption`.

## The Me Keyword

To make things more interesting, Access provides the `Me` keyword, which refers to the current object or form within the scope of the VBE code module. More specifically, I can use the `Me` keyword in place of the Access form class name to access the current form's properties, methods, and controls.

```
Me.Caption = "updating the form's caption property"  
Me.lblSalary.Caption = "updating the label's caption property"
```

The `Me` keyword provides a self-documenting feature for VBA programmers in that it explicitly tells the reader what object, property, or form you are referring to.

In addition to the dot operator (`.`), Microsoft VBA provides the exclamation point (`!`) identifier for identifying what type of item or object immediately follows.

```
Me!lblSalary.Caption = "updating the label's caption property"
```

VBA supports two operators, the dot and exclamation mark, for accessing object properties and collection items. Because the dot and exclamation mark operators can often be interchanged, it can be confusing to remember which serves what purpose and when to use what. As a general rule of thumb, use the exclamation mark operator prior to accessing an item in

a collection, and use the dot operator when referencing a property of a form or control. To keep things simple, however, I use the dot operator to reference both items in collections and properties of forms and controls.

## Assignment Statements

You can assign data to object properties, such as the form's `Caption` property, using an assignment operator in a VBA assignment statement. The *assignment operator* is really a fancy term for the equals (=) sign. However, it's really more important, as you soon see. To demonstrate, evaluate the next lines of VBA code, which assign the text "Ouch!" to the `Caption` property of the `Form1` control.

```
Form.Caption = "Chapter 1"
```

Or

```
Forms("Form1").Caption = "Ouch!"
```

Or

```
Forms(0).Caption = "Chapter 1"
```

Or

```
Me.Caption = "Ouch!"
```

Or

```
Form_Form1.Caption = "Ouch!"
```

A core concept in most programming languages is to understand the difference between data assignment and equality testing. This is especially important in programming languages such as VBA, which use the same operator.

Specifically, the next assignment statement reads: "The `Caption` property takes the literal value `ouch` or the `Caption` property gets the literal value `ouch`."

```
Me.Caption = "Ouch!"
```

Either way, the equals sign in an assignment statement is not testing for equality. In other words, you never want to read the previous assignment as "the `Caption` property equals `Ouch`!"

In the next chapter, I discuss how the equals sign can be used in testing for equality.

## Command and Label Objects

I now show you how to put your knowledge of event procedures, VBA statements, objects, and their properties to work by building two small programs with VBA.

Let's start by building a program that allows a user to turn off and on a light switch. Begin by adding a new form to an Access database and naming it `Light Switch`. Next, add one label control to the form and assign the following property values to it:

- Name: `lblCaption`
- Caption: Lights are on
- Font Size: 12

Now I add three image controls to the form, but only one of them is visible during the form's runtime. (You'll see why shortly.) Add the following property values to the image controls:

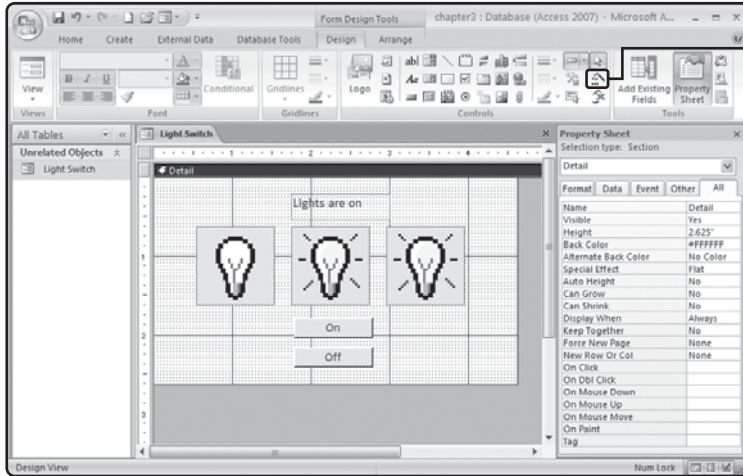
- Name: `imgMain`
- Picture: `LIGHTON.ICO` (Image located on companion website)
- Visible: Yes
- Name: `imgOn`
- Picture: `LIGHTON.ICO` (Image located on companion website)
- Visible: No
- Name: `imgOff`
- Picture: `LIGHTOFF.ICO` (Image located on companion website)
- Visible: No

Now add two command buttons to the form, which allows the user to turn off and on the light switch. Do not use the wizard while adding these command buttons.

- Name: `cmdOn`
- Caption: On
- Name: `cmdOff`
- Caption: Off



You can turn off and on the Control Wizards (command button Control Wizard) by clicking the Control Wizards icon in the Controls section of the Design tab, as shown in Figure 3.6.



Turn off and  
on the Control  
Wizard

**FIGURE 3.6**

Viewing the Light  
Switch program in  
Design view.



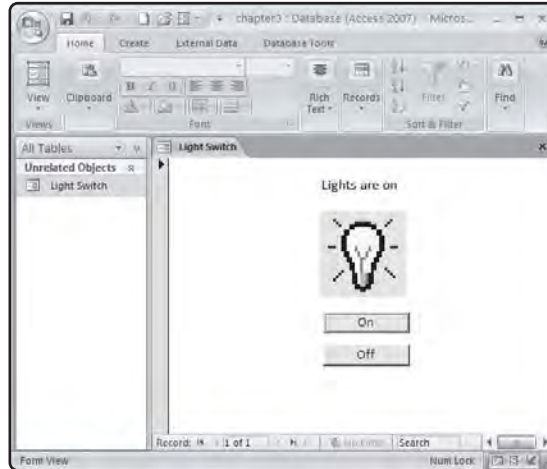
When a graphic's path and filename are assigned to an Image control's Picture property, Microsoft Access does not include the image as part of its .accdb file. To use the Light Switch program located on this book's companion website, you must first change the Picture property's value to a location on your PC. This caution applies to all programs on the companion website that have references to images.

A depiction of my completed form in Design view is revealed in Figure 3.6. Sample code from the Light Switch form is shown next.

```
Private Sub cmdOff_Click()  
    Me.lblCaption.Caption = "Lights are off"  
    Me.imgMain.Picture = Me.imgOff.Picture  
End Sub
```

```
Private Sub cmdOn_Click()  
    Me.lblCaption.Caption = "Lights are on"  
    Me.imgMain.Picture = Me.imgOn.Picture  
End Sub
```

I use only one image control (imgMain) to display one or the other light bulb image. This is why I set the other two image control's Visible properties to No. The final output of my Light Switch form in runtime mode is seen in Figure 3.7.

**FIGURE 3.7**

The completed  
Light Switch  
form in Form view.



You can get rid of the lines and scrollbars on a form in runtime by setting the following form property values to No:

- **DividingLines:** Used to separate sections on a form.
- **NavigationButtons:** Provides access to navigation buttons and a record number box.
- **RecordSelectors:** Record selectors display the unsaved record indicator when a record is being edited in Form view.

I now create a **Colors** program that allows a user to change the color of a label control and exit the Access application without the assistance of a control wizard. First, I create my **Colors** form and set the following form properties:

- **Dividing Lines:** No
- **Navigation Buttons:** No
- **Record Selectors:** No

I add four command buttons (three to change colors and one to exit the application) and one label control that displays the color selected by the user:

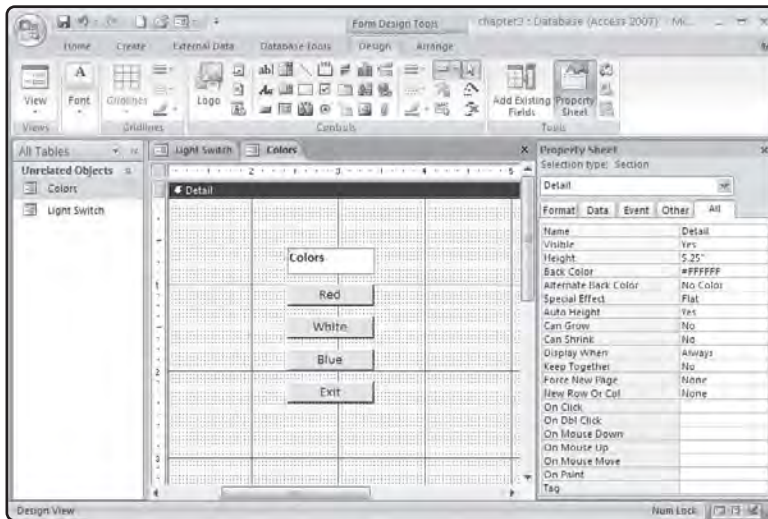
- **Name:** cmdExit
- **Caption:** E&xit
- **Name:** cmdRed
- **Caption:** Red

- Name: cmdWhite
- Caption: White
- Name: cmdBlue
- Caption: Blue
- Name: lblDisplay
- Caption: colors
- Font Weight: Bold
- Back Style: Normal



The label's `BackColor` property cannot be changed unless the corresponding label's `BackStyle` property is set to `Normal`.

A picture of the Colors form in design time should look similar to that in Figure 3.8. The VBA code for each button's Click event is shown next.



**FIGURE 3.8**

The completed Colors form in Design view.

```
Private Sub cmdBlue_Click()
    Me.lblDisplay.BackColor = vbBlue
End Sub
```



```
Private Sub cmdRed_Click()  
    Me.lblDisplay.BackColor = vbRed  
End Sub
```

---

```
Private Sub cmdWhite_Click()  
    Me.lblDisplay.BackColor = vbWhite  
End Sub
```

The way I used the `Me` keyword to access the label and its corresponding properties should not be new to you. What should have caught your attention were the values I used in assigning `BackColor` properties. Specifically, VBA provides you access to eight color constants:

- `vbBlack`
- `vbRed`
- `vbGreen`
- `vbYellow`
- `vbBlue`
- `vbMagenta`
- `vbCyan`
- `vbWhite`

It's important to note that `BackColor` and `ForeColor` properties actually take a number value, which each color constant stores representatively. In addition to using VBA color constants, you can assign numbers representing a multitude of colors using either the `RGB` function or by viewing the `BackColor` or `ForeColor` properties in design time using the Properties window.

To terminate an Access application, use the `DoCmd` object, which runs Microsoft Access functionality from VBA code, and access its built-in `Quit` method as shown in the next command button `Click` event procedure.

```
Private Sub cmdExit_Click()  
    DoCmd.Quit  
End Sub
```

The VBA code in the `cmdExit_Click()` event procedure is similar to the code generated by the Access Control Wizard to quit an Access application using the `DoCmd` object and its `Quit` method.



The ampersand (&) character creates keyboard shortcuts with the Alt key when placed in the `Caption` property of certain controls such as command buttons.

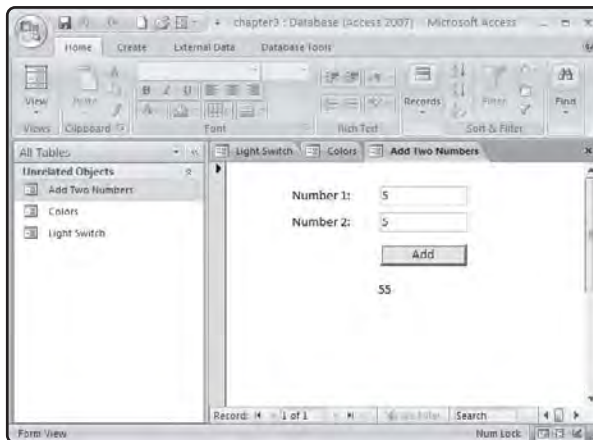
## Getting User Input with Text Boxes

Text box controls receive all types of input from users such as dates, time, text, and numbers. VBA programmers (that's you) write code in procedures to collect the user input and process it. This may seem trivial, but it's not.

Consider a simple application that requests a user to enter two numbers, after which the user clicks a command button to add the two numbers. After adding the numbers, the program should display its output in a label control.

```
Private Sub cmdAdd_Click()  
    Me.lblOutput.Caption = Me.txtNum1.Value + Me.txtNum2.Value  
End Sub
```

I can use a VBA assignment statement to add the value of both text boxes and assign the result to the label control's `Caption` property. Given this fact, why is the output of this VBA statement 55, as revealed in Figure 3.9, instead of 10?



**FIGURE 3.9**

Concatenating two numbers instead of adding them.

This is an excellent question and best answered by examining the `Value` property of a text box. The text box's `Value` property returns or sets the text box's `Text` property (more on this in a moment). Because the `Text` property returns a *string* (a textual description of what's inside the text box), the output seen in Figure 3.9 is generated because I've added two strings together ("5" and "5" makes "55"). In other words, I concatenated them.

To accurately process numbers retrieved from text boxes, you use a built-in VBA function called `Val`. The `Val` function is simple to use. It takes a string as input and returns a numeric value. The next set of VBA code uses the `Val` function to correct the previous program's output.

```
Private Sub cmdAdd_Click()  
    Me.lblOutput.Caption = Val(Me.txtNum1.Value) + _  
        Val(Me.txtNum2.Value)  
End Sub
```

Notice in this example that each `Val` function takes a string as input. Specifically, I use two separate `Val` functions to convert each text box's `Value` property, one at a time, on both sides of the addition operation. The strings contained in the `Value` property are converted to numeric values prior to performing mathematical operations.

Now back to the relationship between the text box's `Value` and `Text` properties. If the `Text` property already contains the contents of the text box, then why use the `Value` property? Another excellent question! Before I answer, look at the following updated code that uses the `Text` property to add two numbers; the output is shown in Figure 3.10.

```
Private Sub cmdAdd_Click()  
    Me.lblOutput.Caption = Val(Me.txtNum1.Text) + _  
        Val(Me.txtNum2.Text)  
End Sub
```

**FIGURE 3.10**

Attempting to use the `Text` property to retrieve user input from text boxes.



As Figure 3.10 depicts, VBA does not like this approach. Why? The `Text` property of a text box is only accessible once the text box has focus. In other words, the `Text` property is only current or valid once the text box has the focus. The `Value` property, however, is the saved value of the text box control regardless of its focus.

To clear the text box of all contents, simply assign an empty string, also known as *empty quotes*, to the text box's `Value` property.

```
Me.Text1.Value = ""
```

In subsequent chapters, I show you how to validate user input with validation programming patterns and text box events.

## VARIABLES AND BEGINNING DATA TYPES

Paramount in any programming language is the concept of variables. In a nutshell, *variables* are pointers to storage locations in memory that contain data. You often hear variables referred to as *containers* for data. In reality, they are pointers that represent a memory address pointing to a memory location.

Though every variable created is unique (unique memory address), all variables share some common characteristics:

- Every variable has a name.
- Each variable has an associated memory address (hidden in modern programming languages such as VBA).
- Variables have a data type such as `String`, `Integer`, or `Boolean`.

Variables in Access VBA must begin with a letter and cannot be longer than 255 characters, nor can they contain periods or spaces. When created, variable names point to a location in memory that can be managed during the execution of your program.

Demonstrated next, VBA programmers use the `Dim` keyword (short for dimension) to declare a new variable in what's called a *declaration statement*:

```
Dim myVariable
```

Once a variable has been declared, VBA reserves space in memory so you can store and retrieve data from its memory location using VBA statements. Simply declaring variables is not the end of the road. It is good programming practice to tell VBA what kind of variable, the *data type*, you are creating. When creating variables, you should ask yourself whether your variable stores numbers, strings, Boolean, dates, or object type data.

VBA provides a number of data types for declaring variables. The more common are listed in Table 3.1.

By default, VBA initializes your declared variables for you. Specifically, all number-based variables are initialized to zero (0), strings are initialized to empty string (""), and Boolean variables are initialized to `False`. This may seem trivial, but it is a nice feature that is not offered in every programming language.

To assign a data type to a variable, simply supply a data type name in the variable declaration using the `As` clause.

**TABLE 3.1 COMMON DATA TYPES IN VBA**

<b>Data Type</b>	<b>Description</b>
Boolean	True (-1) or False (0)
Currency	Useful for money or fixed-point calculations
Date	Holds date and time information
Double	64-bit data type that holds double-precision floating-point numbers
Integer	Stored as a 16-bit (2-byte) number
Long	Stored as a 32-bit number
Single	Stores single-precision floating-point 32-bit (4-byte) numbers
String (variable length)	Stores 1 to 2 billion characters
String (fixed length)	Stores 1 to 64,000 characters
Variant	Can store numeric, string, date/time, Null, or Empty data

```
Dim myName As String
```

With this declaration statement, I've created one variable of `String` data type called `myName`. I can now use the `myName` variable in VBA statements to get and set data inside reserved memory, to which the variable `myName` points. This concept is demonstrated in the following statement.

```
myName = "Emily Elizabeth"
```

Notice when assigning data to string variables that the data on the right side must be enclosed with double quotes. Moreover, VBA programmers can use the concatenation operator (&) to glue two or more strings together. The next few VBA statements reveal VBA string concatenation.

```
Dim myTitle As String
myTitle = "Access VBA " & "Programming for the " & "Absolute Beginner"
Me.Caption = myTitle
```

In the preceding example, I successfully assigned the contents of the `myTitle` variable to the `Caption` property of the form, which works because both the `String` variable and `Caption` property store string data types.

Numbers, however, do not require double quotes when used in assignment statements.

```
Dim mySalary As Double
mySalary = 50000.55
myBalance = -457.23
```



Understanding the difference between string data and string variables is an important concept in beginning programming. Beginning programmers often forget to surround text with double quotes when assigning data to string-based variables or properties. Forgetting to do so can cause compile-time errors.

Study the next program statement and see if anything strikes you as weird.

```
Dim mySalary As Double  
Me.Caption = mySalary
```

It's intriguing that I can assign the variable `mySalary` (a `Double`) to a property such as `Caption`, which holds `String` data types. After executing, the value in the `Caption` property is now "50000.55" and not 50000.55.

Many languages, such as the C language, would not like the preceding assignment statement one bit. This is because many languages require you to convert or cast data of one data type prior to assigning the value to a container of a different data type.

Do not count on VBA to always convert data successfully for you. In fact, it is good programming practice to always use the `Val` function to convert strings to numbers when performing numeric calculations on string variables or properties.

In addition to variables, most programming languages, including VBA, provide support for constants. Unlike variables, constants retain their data values throughout their scope or lifetime.

Constants are useful for declaring and holding data values that will not change during the life of your application. Unless they are declared in a standard code module using the `Public` keyword, constants cannot not be changed once declared.

In VBA, you must use the `Const` statement to declare a constant as revealed in the next statement, which creates a constant to hold the value of `PI`.

```
Const PI = 3.14
```

For readability, I like to capitalize the entire constant name when declaring constants in my VBA code. This way, they really stick out for you and other programmers when seeing their name among other variable names in program code.

## IN THE REAL WORLD

At the lowest computer architecture level, data is represented by electrical states in digital circuits. These states can be translated into binary representations of 0s and 1s, which modern day computing systems can understand as machine language. Understanding how data is converted to and from binary codes is beyond the scope of this book. But, it is worth noting that depending on interpretation, binary codes can represent both a character and an Integer number. To demonstrate this concept, study Table 3.2.

**TABLE 3.2**    **EXAMPLE BINARY REPRESENTATIONS**

Binary Code	Integer Equivalent	Character Equivalent
01100001	97	a
01100010	98	b
01100011	99	c
01100100	100	d

Interesting! The information in Table 3.2 should trigger a question in your head, which goes something like this: “If binary codes can represent both characters and numbers, how do I know what type of data I’m working with?” The notion and application of variables help to answer this question. Variables provide a storage mechanism that accurately manages the binary representations for us. For example, if I store data in an Integer variable, I can feel pretty good VBA will give me back an Integer number. And, if I store data in a String variable, I feel pretty good VBA will give me back characters and not a number. Using built-in VBA functions, it is possible to convert numbers to strings (characters) and strings to numbers.

With the knowledge of how data is represented, consider that data can be stored in varying types of media such as volatile memory (also known as random access memory or RAM) and nonvolatile areas such as disk drives. Programmers can easily manage volatile memory areas using variables with languages like VBA. Nonvolatile memory areas, such as hard drives, are generally managed (stored) in systems such as files or databases like Microsoft Access.

## Variable Naming Conventions

Depending on the programmer and programming language, there are a number of popular naming conventions for variables. I like to use a single prefix that denotes data type, followed

by a meaningful name with first letters capitalized for each word comprising the variable name. For constants, it is advisable to capitalize the entire name so that it stands out from the rest of the code. Table 3.3 lists some common data types with a sample variable name and purpose.

**TABLE 3.3    SAMPLE NAMING CONVENTIONS**

<b>Data Type</b>	<b>Purpose</b>	<b>Sample Variable Name</b>
Boolean	Determines if a user is logged in	bLoggedIn
Currency	Specifies an employee's salary	cSalary
Date	Employee's hire date	dHireDate
Double	Result of calculation	dResult
Integer	Tracks a player's score	iScore
Long	Current temperature	lTemperature
Single	Miles traveled on vacation	sMilesTraveled
String	Employee's last name	sLastName
Const	A constant, which holds the current tax rate	TAXRATE

As you program more and observe more programming, you will notice many other popular naming conventions. The important note is to use a naming convention and stick with it throughout your code!

## Variable Scope

Variable scope is a fancy way of describing how long a variable will hold its data, or in other words, its lifetime. VBA supports three types of variable scope.

- Procedure-level scope
- Module-level scope
- Public scope

To create a variable with procedure-level scope, simply declare a variable inside of a procedure.

```
Private Sub Form_Load()  
    Dim dProfit As Double  
    dProfit = 700.21  
End Sub
```



In the preceding form `Load` event procedure, I declared a `Double` variable called `dProfit` that will hold its value as long as the current scope of execution is inside the procedure. More specifically, once program execution has left the form `Load` procedure, the `dProfit` variable is initialized back to 0.

If you need to maintain the value of `dProfit` for a longer period of time, consider using a module-level or public variable. Module-level variables are only available to the current module from where they are declared, but are available to all procedures contained within the same module. Moreover, module-level variables are considered private and can be declared either with the keyword `Dim` or `Private` in the general declarations area, as demonstrated next.

```
Dim dRunningTotal As Double ' module-level variable
Private iScore As Integer ' module-level variable
```

You can create public variables that are available to the entire project (all code modules) by declaring a variable using the `Public` keyword in the general declarations area of a code module.

```
Public bLoggedIn As Boolean
```



The general declarations area is located at the top of a code module and is considered an area that is outside of any procedure.

Determining variable scope is part of application development—during which you define all needed variables, their storage type, and scope.

## Option Statements

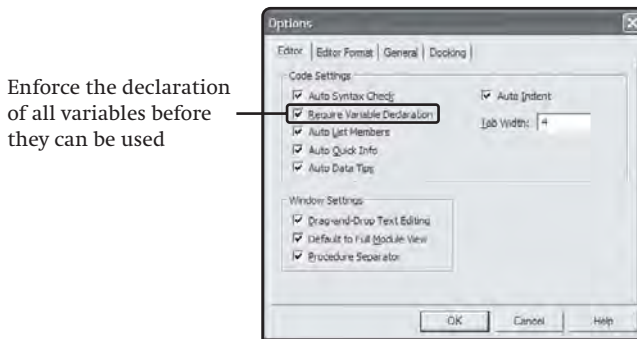
VBA has a few module-level utility statements, known as options, that are used for naming conventions, string comparisons, and other internal settings. First off, you may have already noticed the `Option Compare Database` statement located in the general declarations area of the VBE Code window.

Per Microsoft, the `Option Compare Database` statement “performs string comparisons based on the sort order determined by the locale ID of the database where the string comparisons occur.” This statement can be modified to either `Option Compare Binary` or `Option Compare Text` instead of `Option Compare Database`. If your VBE code module does not include an `Option Compare` statement, VBA will default to `Option Compare Binary`, which results in string comparisons based on a character’s internal binary representation.

The next option statement, `Option Explicit`, is more important to beginning VBA programmers, as it forces you to explicitly declare all variables before you can use them. This is a

huge, I mean HUGE service to even seasoned VBA programmers. By forcing the explicit declaration of variables, you are saved an often painful process of misspelling or misrepresenting variables that ultimately lead to program or compile errors.

Unless you tell Microsoft Access to make it so, the `Option Explicit` statement may not appear by default in your VBE code module. To have this statement provided in each of your code modules, simply access the Options window from the VBE Tools menu and select the Require Variable Declaration setting, as demonstrated in Figure 3.11.

**FIGURE 3.11**

Requiring variable declaration in the Options window.

The next option clause is the `Option Base` statement, which is manually typed into the general declarations area of each code module. In a nutshell, the `Option Base` statement defines the lower bounds for arrays. VBA arrays are by default 0-based arrays, but can start at 1 using an `Option Base` statement as seen next.

```
Option Base 1
```

I discuss arrays and their upper and lower bounds in more detail in subsequent chapters.

## VBA ARITHMETIC AND ORDER OF OPERATIONS

It's no secret, programming in any language involves some level of math. Though it's not necessary to be a mathematical wiz in calculus, algebra, or trigonometry, it is useful to understand the essential arithmetic operators and order of precedence offered in a given programming language. For basic mathematical operations, VBA supports the operators shown in Table 3.4.

In addition to basic math operations, VBA supports what's known as order of operations using parentheses. Without parentheses, VBA determines order of operations in the following order.

1. Exponents
2. Multiplication and division
3. Addition and subtraction

**TABLE 3.4 COMMON MATHEMATICAL OPERATORS**

Operator	Purpose	Example	Result
+	Addition	dSalary = 521.9 + 204	725.9
-	Subtraction	iPoints = 100 - 20	80
*	Multiplication	dResult = 5 * 213.78	1068.9
/	Division	iResult = 21 / 3	7
^	Exponentiation	iResult = 2 ^ 3	8

When VBA encounters a tie between operators, it performs calculations starting from the leftmost operation. To get a better handle of the importance of operator precedence and order of operations, consider the following equation, which calculates a profit.

Profit = (price \* quantity) - (fixed cost + total variable cost)

The next VBA assignment statement implements the preceding equation without parentheses (in other words, without a well-defined order of operations):

```
dProfit = 19.99 * 704 - 406.21 + 203.85
```

The result of this calculation is 13870.6. Now study the next VBA statement which implements the same equation, this time using parentheses to build a well-defined order of operations.

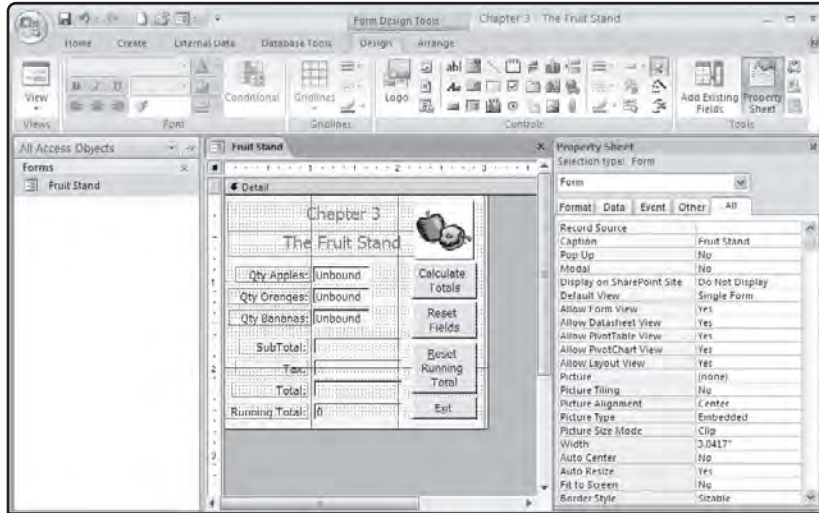
```
dProfit = (19.99 * 704) - (406.21 + 203.85)
```

Using parentheses to guide my order of operations, my new profit is 13462.9. That's a difference of \$407.70 that might have been recorded as inflated profits!

## CHAPTER PROGRAM: FRUIT STAND

The Fruit Stand program is a simplified data entry system for a small fruit vendor. It implements many chapter-based concepts such as variables, constants, and VBA statements.

To build the Fruit Stand program, you'll need to create a form in Design view, as shown in Figure 3.12.

**FIGURE 3.12**

Building the Fruit Stand program in Design view.

Controls and properties of the Fruit Stand program are described in Table 3.5.

**TABLE 3.5 CONTROLS AND PROPERTIES OF THE FRUIT STAND PROGRAM**

Control	Property	Property Value
Form	Name	Fruit Stand
	Caption	Fruit Stand
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No
Label1	Name	lblTitle1
	Caption	Chapter 3
Label1	Name	lblTitle2
	Caption	The Fruit Stand
Label1	Name	lblQtyApples
	Caption	Qty Apples:
Label1	Name	lblQtyOranges
	Caption	Qty Oranges:
Label1	Name	lblBananas
	Caption	Qty Bananas:

Control	Property	Property Value
Label	Name	lblSubTotalCaption
	Caption	Sub Total:
Label	Name	lblTaxCaption
	Caption	Tax:
Label	Name	lblTotalCaption
	Caption	Total:
Label	Name	lblRunningTotalCaption
	Caption	Running Total:
Text Box	Name	txtApples
Text Box	Name	txtOranges
Text Box	Name	txtBananas
Label	Name	lblSubTotal
	Caption	(empty)
	Special Effect	Sunken
Label	Name	lblTax
	Caption	(empty)
	Special Effect	Sunken
Label	Name	lblTotal
	Caption	(empty)
	Special Effect	Sunken
Label	Name	lblRunningTotal
	Caption	(empty)
	Special Effect	Sunken
Command Button	Name	cmdCalculateTotals
	Caption	Calculate Totals
Command Button	Name	cmdResetFields
	Caption	Reset Fields
Command Button	Name	cmdResetRunningTotal
	Caption	Reset Running Total
Command Button	Name	cmdExit
	Caption	E&xit
Image	Name	imgFruit
	Picture	apples.gif
		(located on the companion website)

All of the code required to build the Fruit Stand program is shown here.

Option Compare Database

Option Explicit

' declare module level variable and constants

Dim dRunningTotal As Currency

Const TAXRATE = 0.07

Const dPricePerApple = 0.1

Const dPricePerOrange = 0.2

Const dPricePerBanana = 0.3

---

Private Sub cmdCalculateTotals\_Click()

' declare procedure-level variables

Dim dSubTotal As Currency

Dim dTotal As Currency

Dim dTax As Currency

' calculate and apply sub total

dSubTotal = (dPricePerApple \* Val(Me.txtApples.Value)) + \_  
             (dPricePerOrange \* Val(Me.txtOranges.Value)) + \_  
             (dPricePerBanana \* Val(txtBananas.Value))

Me.lblSubTotal.Caption = "\$" & dSubTotal

' calculate and apply tax

dTax = (TAXRATE \* dSubTotal)

Me.lblTax.Caption = "\$" & dTax

' calculate and apply total cost

dTotal = dTax + dSubTotal

Me.lblTotal.Caption = "\$" & dTotal

' build and apply running total using module-level variable

dRunningTotal = dRunningTotal + dTotal

```
Me.lblRunningTotal.Caption = "$" & dRunningTotal  
End Sub
```

---

```
Private Sub cmdExit_Click()  
    DoCmd.Quit ' terminates the application  
End Sub
```

---

```
Private Sub cmdResetFields_Click()  
    ' reset application fields  
    Me.txtApples.Value = "0"  
    Me.txtOranges.Value = "0"  
    Me.txtBananas.Value = "0"  
    Me.lblSubTotal.Caption = "$0.00"  
    Me.lblTax.Caption = "$0.00"  
    Me.lblTotal.Caption = "$0.00"  
End Sub
```

---

```
Private Sub cmdResetRunningTotal_Click()  
    ' reset running total variable and application field  
    dRunningTotal = 0  
    Me.lblRunningTotal.Caption = "$0.00"  
End Sub
```

---

```
Private Sub Form_Load()  
    ' set focus to first text box  
    Me.txtApples.SetFocus  
    'set default quantities when the form first loads  
    Me.txtApples.Value = 0  
    Me.txtBananas.Value = 0  
    Me.txtOranges.Value = 0  
End Sub
```

## SUMMARY

- The event-driven paradigm allows programmers to build applications that respond to actions initiated by the user or system.
- Access VBA includes a number of events that are categorized by the objects they represent.
- Objects are nouns such as a person, place, or thing.
- Objects have properties that describe the object and methods, which perform actions.
- Properties and methods of objects are accessed using the dot operator (.).
- The VBE (Visual Basic Environment) contains a suite of windows, toolbars, and menu items that provide support for text editing, debugging, file management, and help.
- Procedures are containers for VBA code.
- VBA statements are comprised of variables, keywords, operators, and expressions that build a complete instruction to the computer.
- Comment statements are ignored and not processed as a VBA statement by the computer.
- The `Forms` collection contains all open forms in an Access database.
- The `Me` keyword refers to the current object or form within the scope of the VBE code module.
- The `DoCmd` object runs Microsoft Access functionality from VBA code.
- Use the `Val` function to accurately process numbers retrieved from text boxes.
- Variables are declared using the keyword `Dim`.
- Variables are pointers to storage locations in memory that contain data.
- All number-based variables are initialized to zero (0), string variables are initialized to empty string (""), and Boolean variables are initialized to `False`.
- Constants are useful for declaring and holding data values that will not change during the life of your application.
- The `Option Explicit` statement forces an explicit declaration before a variable can be used.
- VBA supports order of operations using parentheses.



## PROGRAMMING CHALLENGES

1. Create a simple word processor that allows a user to enter text into a large text box. (Hint: Set the Enter Key Behavior property of a text box to New Line in Field.) The user should be able to change the foreground and background colors of the text box using three command buttons representing three different colors. Also, the user should be able to change the font size of the text box using up to three command buttons representing three different font sizes.
2. Build a simple calculator program with an Access form that allows a user to enter numbers in two separate text boxes. The Access form should have four separate command buttons for adding, subtracting, multiplying, and dividing. Write code in each command button's click event to output the result in a label control.
3. Create a discount book program that allows a user to enter an initial book price, a discount rate (e.g., 10% off), and a sales tax percentage. The program should display, in labels, the derived discount price, sales tax amount, and final cost of the book.



# CONDITIONS

**I**n this chapter I show you how to implement *conditions*, which allow programmers to build decision-making abilities into their applications using `If` blocks and `Select Case` structures. In addition, I show you how to leverage VBA's built-in dialog boxes and additional controls to enhance your graphical interface and your system's intelligence.

## IF BLOCKS

A basic component of a high-level language is the ability to construct a condition. Most high-level programming languages offer the `If` block as a way to evaluate an expression. Before proceeding into `If` blocks, I discuss what an expression is in terms of computer programming.

In programming terms, *expressions* are groupings of keywords, operators, and/or variables that produce a variable or object. Expressions are typically used to conduct calculations and to manipulate or test data. Moreover, expressions can be used to build conditions, which return a Boolean value of `True` or `False`. This is an important concept, so I am repeating in italics: *Expressions can be used to build conditions that evaluate to `True` or `False`.*

VBA programmers can use expressions in an If condition.

```
If (number1 = number2) Then
    Me.Label1.Caption = "number1 equals number2"
End If
```

Known as an If block, the preceding code reads “If the variable `number1` equals the variable `number2`, then assign some text to the `Caption` property of `Label1`.” This means the expression inside the parentheses must evaluate to `True` for the VBA statement inside the If block to execute. Note that the parentheses surrounding the expression are not required, but provide readability.

Also note the inclusion of the `Then` keyword at the end of the If statement. The `Then` keyword is required at the end of each If statement.



Always indent VBA statements inside of a condition or loop to provide easy-to-read code. A common convention is to indent two or three spaces or to use a single tab. Doing so implies that the VBA assignment statement belongs inside the If block.

But what if the expression does not evaluate to `True`? To answer this question, VBA includes an `Else` clause, which catches the program’s execution in the event the expression evaluates to `False`. The If/Else block is demonstrated next.

```
If (number1 = number2) Then
    Me.Label1.Caption = "number1 equals number2"
Else
    Me.Label1.Caption = "number1 does not equal number2"
End If
```

Given the preceding examples, you might be asking yourself about possibilities for building simple expressions with operators other than the equals sign. As shown in Table 4.1, VBA supports many common operators to aid in evaluating expressions.

In addition to the `Else` clause, VBA provides the `ElseIf` clause as part of a larger expression. The `ElseIf` clause is one word in VBA and is used for building conditions that may have more than two possible outcomes.

```
If (number1 = number2) Then
    Me.Label1.Caption = "number1 equals number2"
ElseIf (number1 > number2) Then
    Me.Label1.Caption = "number1 is greater than number2"
```

**TABLE 4.1** COMMON OPERATORS USED IN EXPRESSIONS

Operator	Description
=	Equals
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

```
ElseIf (number1 < number2) Then
    Me.Label1.Caption = "number1 is less than number2"
End If
```

Notice in the preceding example that the `ElseIf` clause must include an expression followed by the keyword `Then`, just like an `If` condition. In addition, you can use the `Else` clause to act as a concluding clause in the event that none of the conditions evaluates to `True`, as seen next.

```
If (sColor = "red") Then
    Me.Label1.Caption = "The color is red"
ElseIf (sColor = "white") Then
    Me.Label1.Caption = "The color is white"
ElseIf (sColor = "blue") Then
    Me.Label1.Caption = "The color is blue"
Else
    Me.Label1.Caption = "The color is not red, white or blue"
End If
```

## Nested If Blocks

There are times when you may need to provide one or more conditions inside of another condition. This concept is known as *nested conditions* and can often require much thought regarding the flow of the program.

To exhibit the concept of nested conditions, I build a nested `If` block, which implements a simple payroll system.

```
If (sEmployeeType = "salary") Then
    ' Employee is paid a salary.
    cPay = cSalary
Else
    ' Employee paid hourly wages and has worked 40 or less hours
    If (iHoursWorked <= 40) Then
        cPay = cHourlyRate * iHoursWorked
    Else
        ' Employee earned overtime, which is time and a half
        cOverTime = (iHoursWorked - 40) * (cHourlyRate * 1.5)
        cPay = (cHourlyRate * 40) + cOverTime
    End If
End If
```

Because I used indenting techniques, you can easily see I have a nested If block inside of the Else block. This nested If block is executed only if the first If condition evaluates to False. If the first, or outer, If condition evaluates to True, the employee wage is calculated as a salary, after which program control is sent to the outer, or last, End If statement.

Without indentation, the preceding nested program code is very difficult to read. Always indent program statements that include nested If blocks inside conditions.

## Compound If Blocks

So far, you've seen how to build simple and nested conditions using If blocks. There is, however, much more to consider if you plan to build more complex decision-making capabilities, such as compound conditions, into your VBA applications. To build compound expressions, VBA programmers can use the conditional operators And, Or, and Not.



Conditional operators such as And, Or, and Not are considered reserved keywords and must be used in an expression. Otherwise, VBA will generate a compile error.

To get a better understanding of the conditional operators And, Or, and Not, I use what's known as truth tables to demonstrate possible scenarios and results for each operator. A *truth table* must include inputs and their possible results. Each input can evaluate to either True or False. Using one or more inputs and a corresponding operator, you can build all possible results in a truth table. Regardless of the number of inputs and type of operator, a compound expression ultimately results in either True or False.

Truth tables are commonly used in mathematic and logic circles such as quantitative analysis, discrete mathematics, and Boolean algebra. Using logical operators, truth tables allow one to evaluate all possible results to prove an outcome.

Table 4.2 demonstrates the truth table for the And operator. The And operator uses two inputs to determine the result for the entire expression.

**TABLE 4.2 TRUTH TABLE FOR THE AND OPERATOR**

Input X	Input Y	Result
True	True	True
True	False	False
False	True	False
False	False	False

You can see from the truth table that there is only one occasion when the And operator generates a True result in an expression—when both inputs are True.

The next program block implements a compound condition in VBA using the And operator.

```
If (sEmpType = "salary" And sEmpEvalResult <> "poor") Then
    ' Employee is given a 20% bonus.
    cBonusPay = cSalary * .20
End If
```

In the preceding example, the employee is given a 20% bonus only if both conditions within the parentheses are True. If either condition is False, the entire compound condition evaluates to False, and the employee is not awarded the bonus.

The Or operator in Table 4.3 has a much different effect based on its inputs. More specifically, the Or operator always generates a True value, provided at least one input is True. The only time a compound condition using the Or keyword results in a False result is when both inputs on each side of the Or operator are False.

The next block of code demonstrates a compound condition in VBA using the Or operator.

**TABLE 4.3 TRUTH TABLE FOR THE OR OPERATOR**

Input X	Input Y	Result
True	True	True
True	False	True
False	True	True
False	False	False

```
If (sMonth = "June" Or sMonth = "July") Then
    sSeason = "Summer"
End If
```

As long as the variable `sMonth` is either June or July, the variable `sSeason` is set to Summer. Only one side of the expression needs to be True for the entire condition to be True.

The truth table for the Not operator (seen in Table 4.4) contains only one input. In a nutshell, the Not operator reverses the value of its input value such that Not true results in False and Not false results in True.

**TABLE 4.4 TRUTH TABLE FOR NOT OPERATOR**

Input X	Result
True	False
False	True

The Not operator is implemented next in VBA, as seen in the next program block.

```
If Not(5 = 5) Then
    Me.lblResult.Caption = "true"
Else
    Me.lblResult.Caption = "false"
End If
```

Given the preceding code, what do you think the value of the label's `Caption` property will be? If you said False, you would be correct. Why? To understand, you must look at the result of the inner expression (`5=5`) first, which evaluates to True. The outer expression, `Not(True)` or

`Not(5=5)`, evaluates to `False`, which means the statement inside the `If` condition does not execute. Instead, the statement inside the `Else` condition executes.

## SELECT CASE STRUCTURES

The `Select Case` structure is another tool for VBA programmers to use to build conditions. Specifically, the `Select Case` structure evaluates an expression only once. It's useful for comparing a single expression to multiple values.

```
Select Case sDay
    Case "Monday"
        Me.lblDay.Caption = "Weekday"
    Case "Tuesday"
        Me.lblDay.Caption = "Weekday"
    Case "Wednesday"
        Me.lblDay.Caption = "Weekday"
    Case "Thursday"
        Me.lblDay.Caption = "Weekday"
    Case "Friday"
        Me.lblDay.Caption = "Weekday"
    Case Else
        Me.lblDay.Caption = "Weekend!"
End Select
```

In this case (excuse the pun), the `Select Case` structure evaluates a string-based variable and uses five `Case` statements to define possible expression values. The `Case Else` statement catches a value in the top expression that is not defined in a `Case` statement.

The `Case Else` statement is not required in a `Select Case` structure. After code within a `Case` or `Case Else` block is executed, program control is then moved to the `End Select` statement, which is required.

The `Select Case` structure is very flexible. For example, I can simplify the preceding structure by using `Select Case`'s ability to place multiple items in a single statement separated by commas.

```
Select Case sDay
    Case "Monday", "Tuesday", "Wednesday", "Thursday", "Friday"
        Me.lblDay.Caption = "Weekday"
    Case Else
        Me.lblDay.Caption = "Weekend!"
End Select
```



In the following code, the `Select Case` structure also allows you to check for a range of values using the `Is` and `To` keywords.

```
Select Case dTemperature
    Case Is < 32
        Me.lblTemperature.Caption = "Freezing"
    Case 32 To 45
        Me.lblTemperature.Caption = "Cold"
    Case 46 To 69
        Me.lblTemperature.Caption = "Cool"
    Case 70 To 89
        Me.lblTemperature.Caption = "Warm"
    Case Is > 89
        Me.lblTemperature.Caption = "Hot"
End Select
```

Using ranges of values and comparison operators, I can easily build logic into my `Case` statements to determine temperature ranges.

## DIALOG BOXES

*Dialog boxes* are generally small windows that prompt the user for a response. Dialog boxes can be configured to include one to three command buttons, which provide the user with various options for interaction. In this section, you learn about two common VBA dialog boxes: the message box and the input box.

### Message Box

VBA's `MsgBox` function is a built-in function that can generate a dialog box. The `MsgBox` function takes five parameters, separated by commas, as input:

```
MsgBox Prompt, Buttons, Title, HelpFile, Context
```

The only argument required by the `MsgBox` function is the `Prompt` parameter, which is displayed on the dialog box to the user. Though not required, the `Buttons` parameter is very useful. You can specify various VBA constants to customize the available buttons. The most common of these constants are shown in Table 4.5.

Another useful but not required parameter is the `Title` argument, which displays text in the title bar area of the message box. Using these parameters, I can create and display a simple dialog box in the `Click` event of a command button:

```
Private Sub Command1_Click()
    MsgBox "I created a dialog box.", vbInformation, "Chapter 4"
End Sub
```

**TABLE 4.5** BUTTON SETTINGS

Constant	Value
vbOKOnly (default)	0
vbOKCancel	1
vbAbortRetryIgnore	2
vbYesNoCancel	3
vbYesNo	4
vbRetryCancel	5
vbCritical	16
vbQuestion	32
vbExclamation	48
vbInformation	64

To successfully use the `Buttons` parameter of the `MsgBox` function, you work with variables and conditions. Specifically, you need to create a variable that holds the user's response when the user selects a button on the dialog box. The variable gets its value from the result of the `MsgBox` function. That's right; the `MsgBox` function not only creates a dialog box, but also returns a value. This is how VBA programmers determine which button on the dialog box was clicked. The possible return values are described in Table 4.6.

**TABLE 4.6** MSGBOX FUNCTION RETURN VALUES

Constant	Value
vbOK	1
vbCancel	2
vbAbort	3
vbRetry	4
vbIgnore	5
vbYes	6
vbNo	7

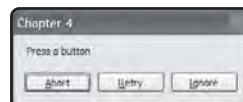
Remember from Chapter 3, “Introduction to Access VBA,” that constants are containers for data that cannot be changed. The built-in VBA constants, such as the ones seen in Tables 4.5 and 4.6, hold integer values. This means you can use either the constant name or its value directly. To see how this works, examine the next program. This program uses the `MsgBox` function, one variable, and a `Select Case` structure to determine what button the user has pressed.

```
Private Sub Command1_Click()  
  
    Dim iResponse As Integer  
  
    ' Display a message box to the user  
    iResponse = MsgBox("Press a button", _  
        vbAbortRetryIgnore, "Chapter 4")  
  
    ' Determine which button was selected.  
    Select Case iResponse  
        Case vbAbort  
            Me.lblResponse.Caption = "You pressed abort."  
        Case vbRetry  
            Me.lblResponse.Caption = "You pressed retry."  
        Case vbIgnore  
            Me.lblResponse.Caption = "You pressed ignore."  
    End Select  
  
End Sub
```

Figure 4.1 demonstrates the message display to the user from the preceding code.

**FIGURE 4.1**

A multibutton message box.



Linefeed characters can be added in a message box prompt using the `Chr(10)` function call.

```
MsgBox "This prompt demonstrates how to add a" & _  
    " line feed character" & Chr(10) & "in a message box."
```

When using the message box function in an expression such as the following, realize that parentheses are required to surround parameters, which are values passed to the function (in this case the MsgBox function).

```
iResponse = MsgBox("Press a button", _  
    vbAbortRetryIgnore, "Chapter 4")
```

Without parentheses, the VBA compiler complains and prevents further execution. On the other hand, the VBA compiler does not like the use of parentheses when the MsgBox function is used by itself.

```
MsgBox "I created a dialog box.", vbInformation, "Chapter 4"
```

This is standard operating procedure when working with VBA functions, so it's worth repeating again in *italics*: *Functions in expressions require the use of parentheses for their parameters, whereas functions outside of expressions or by themselves do not.*

## Input Box

The *input box* also displays a dialog box, but allows a user to input information (hence the name). Like the message box, the input box is created with a function call but takes seven parameters.

```
InputBox Prompt, Title, Default, XPos, YPos, HelpFile, Context
```

The most common InputBox parameters are Prompt, Title, and Default, where Prompt and Title behave similarly to the same parameters of the message box. The Default argument displays default text in the text box area of the input box. Also note that the InputBox function does not have a Buttons parameter. The only required parameter is Prompt.

The InputBox function returns a string data type, so you need to declare a String variable to capture its return value.

In Figure 4.2, I use an input box to prompt a user with a question.



**FIGURE 4.2**

Using an input box to prompt a user for information.

Sample VBA code for Figure 4.2 would look like the following.

```
Private Sub cmdAskQuestion_Click()

    Dim sResponse As String

    sResponse = InputBox("What is the Capitol of Florida?", _
        "Chapter 4")

    If sResponse = "Tallahassee" Then
        Me.lblResponse.Caption = "That is right!"
    Else
        Me.lblResponse.Caption = "Sorry, that is not correct."
    End If

End Sub
```

I will now enhance the previous code to ensure the user has clicked the default OK button on the input box prior to validating the user's response. More specifically, if the user clicks the Cancel button, a zero-length string is returned by the InputBox function. To check for this, I can use an outer If block.

```
Private Sub cmdAskQuestion_Click()

    Dim sResponse As String

    sResponse = InputBox("What is the Capitol of Florida?", _
        "Chapter 4")

    ' Check to see if the user clicked Cancel.
    If sResponse <> "" Then

        If sResponse = "Tallahassee" Then
            Me.lblResponse.Caption = "That is right!"
        Else
            Me.lblResponse.Caption = "Sorry, that is not correct."
        End If

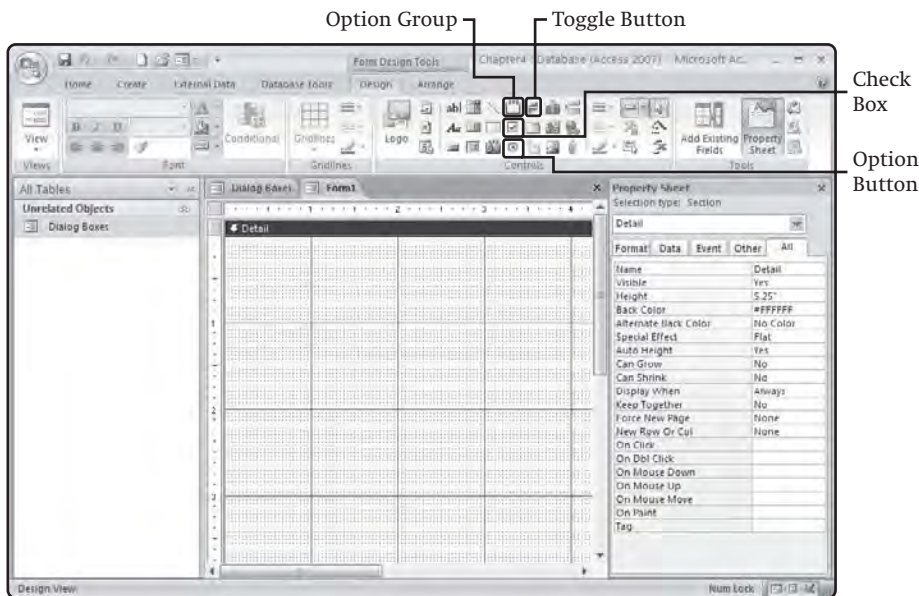
    End If

End Sub
```

## COMMON CONTROLS CONTINUED

Beyond what you've already seen from Chapter 3, there are a number of more common controls available to you in Access. In this chapter, you learn about a few more that can require the use of conditions. Specifically, I discuss the following common controls, and you can view them in Figure 4.3:

- Option groups
- Option buttons
- Check boxes
- Toggle buttons



**FIGURE 4.3**

Common controls  
continued.

## Option Group

The option group control is a container that logically groups controls such as option buttons, check boxes, and toggle buttons. Though not required, the option group provides a very effective wizard for grouping your controls inside the option group's frame.

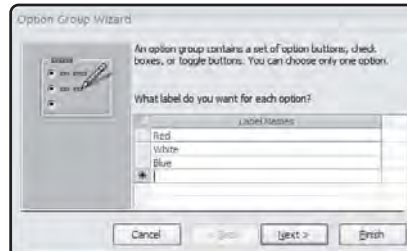
When first adding an option group to your form, Access initiates the Option Group Wizard. As shown in Figure 4.4, the first step in the wizard is to add label names for each item in your group. At this stage, it doesn't matter what control you're using—you're only adding textual descriptions for each item in the group.



The Option Group Wizard does not activate if you've turned off the Control Wizards item in the Access toolbar.

**FIGURE 4.4**

Adding label names for each control in the option group.



After you've added all label names and clicked Next, the wizard asks you to choose a default control if one is desired. In Figure 4.5, I've asked the wizard to make my item, called Red, the default control.

**FIGURE 4.5**

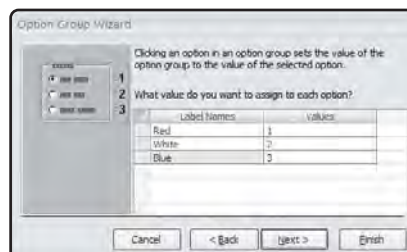
Selecting a default control.



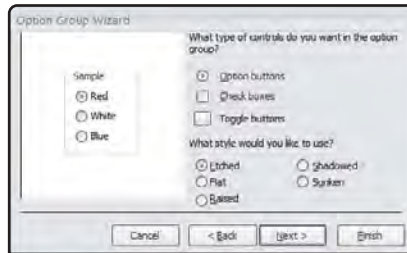
Figure 4.6 depicts the next step in the wizard, where you set values for each option in the group. Option values allow VBA programmers to tell which option the user has selected in the group. The wizard's default values for each option are acceptable.

**FIGURE 4.6**

Providing values for each option.



The next wizard screen, displayed in Figure 4.7, allows you to select what type of option controls are displayed in your option group.



**FIGURE 4.7**

Choosing an option control type for the option group.

The last screen in the wizard (see Figure 4.8) prompts you to enter a caption for your option group frame. This caption is actually a property of a label control that automatically sits at the top of the frame.



**FIGURE 4.8**

Entering a caption for the option group's label.

In the next three sections I show you specific implementations of option groups.

## Option Buttons

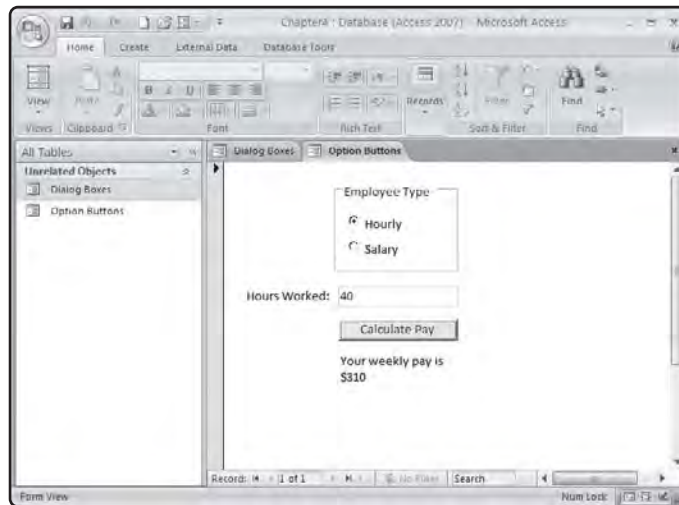
Often referred to as *radio buttons*, *option buttons* provide a user with a list of selectable choices. Specifically, the user can select only one option at a time. Individual option buttons comprise two controls, a label, and an option button. Each has its own properties and can be managed during design time or runtime via VBA.



After creating an option group either manually or with the Option Group Wizard, you should change the name of each option control to one that contains a meaningful description. This greatly reduces confusion when working with VBA code.



To determine which option button has been selected in a group, you use the option group's `Value` property. For this to work, each option button must have been assigned a valid and unique number in its `OptionValue` property (set by default in the Option Group Wizard). When a user clicks an option button, the `Value` property is set to the same number as the option button's `OptionValue` property. These concepts are demonstrated in the next program code, which implements the *graphical user interface (GUI)* in Figure 4.9.



**FIGURE 4.9**

Using option buttons to determine an employee's pay type.

Option Compare Database  
Option Explicit

Const SALARY As Double = 350.25  
Const HOURLYRATE = 7.75

---

```
Private Sub cmdCalculatePay_Click()
```

```
Dim dOverTime As Double  
Dim dNormalPay As Double
```

```
If Me.fraEmployeeType.Value = 2 Then
```

```
    ' Employee is paid a salary
```

```
Me.lblPay.Caption = "Your weekly salary is $" & SALARY

Else

    ' Employee is paid by the hour
    ' Find out if the employee has worked overtime
    If Val(Me.txtHoursWorked.Value) > 40 Then

        dOverTime = (Val(Me.txtHoursWorked.Value) - 40) _
            * (HOURLYRATE * 1.5)
        dNormalPay = HOURLYRATE * 40
        Me.lblPay.Caption = "Your weekly pay is $" & _
            dNormalPay + dOverTime

    Else

        Me.lblPay.Caption = "Your weekly pay is $" & _
            HOURLYRATE * Val(Me.txtHoursWorked.Value)

    End If

End If

End Sub
```

---

```
Private Sub optHourly_GotFocus()
    Me.txtHoursWorked.Enabled = True
    Me.lblPay.Caption = ""
End Sub
```

---

```
Private Sub optSalary_GotFocus()
    Me.txtHoursWorked.Enabled = False
    Me.lblPay.Caption = ""
End Sub
```

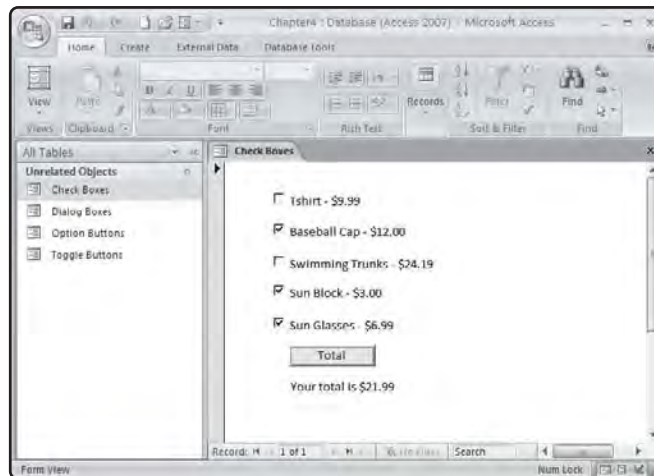
The option group is really defined as the name of the frame. (An option group is really a frame control.) Notice that I used the `GotFocus` method of each option button to disable the `Hours Worked` text box. The `GotFocus` event is triggered whenever the option button receives focus.

## Check Boxes

When used in an option group, *check boxes* behave much like option buttons. If you have experience in other graphical languages, you might be surprised to learn that a user can select only one check box at a time when it is located in an option group. Remember that an option group provides a single selection for any option-based control such as check boxes, option buttons, and toggle buttons.

To use check boxes in a multiselection facility, you need to add them manually, outside of an option group. In addition, you need to set each check box's `DefaultValue` property to a unique number during design time.

Implemented in Figure 4.10, the following code demonstrates how one might use check boxes in a multiselection capacity without an option group.



**FIGURE 4.10**

Selecting more than one check box at a time.

```
Option Compare Database
Option Explicit
```

```
Dim dRunningTotal As Currency
```

```
Private Sub cmdTotal_Click()

    dRunningTotal = 0

    If Me.chkTShirt.Value = True Then
        dRunningTotal = dRunningTotal + 9.99
    End If

    If Me.chkBaseballCap.Value = True Then
        dRunningTotal = dRunningTotal + 12#
    End If

    If Me.chkSwimmingTrunks.Value = True Then
        dRunningTotal = dRunningTotal + 24.19
    End If

    If Me.chkSunBlock.Value = True Then
        dRunningTotal = dRunningTotal + 3#
    End If

    If Me.chkSunGlasses.Value = True Then
        dRunningTotal = dRunningTotal + 6.99
    End If

    Me.lblTotal.Caption = "Your total is $" & _
        dRunningTotal

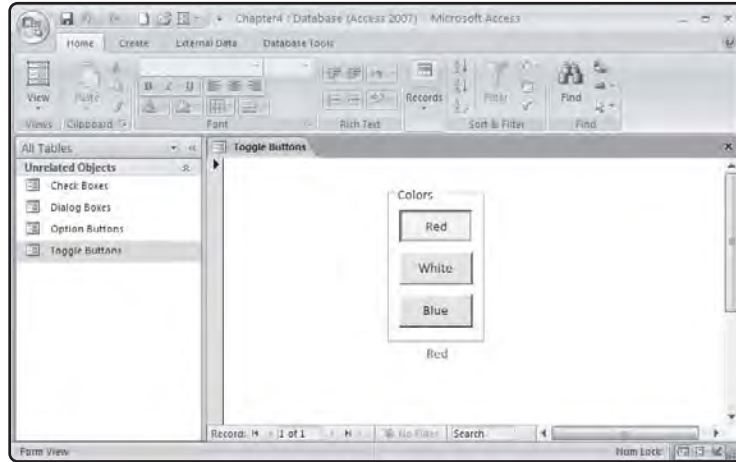
End Sub
```

I can use the `Value` property of each check box to determine whether the user has selected it. If the check box has been checked, the `Value` property is set to `True`; if not, it is set to `False`.

## Toggle Buttons

When used in an option group, *toggle buttons* serve the same purpose as option buttons and check boxes, which allow a user to select one item at a time.

In the next example (seen in Figure 4.11), I use an option group of three toggle buttons to change label properties.

**FIGURE 4.11**

Using toggle buttons in an option group.

Option Compare Database  
Option Explicit

---

```
Private Sub tglRed_GotFocus()
    lblOutput.ForeColor = vbRed
    lblOutput.Caption = "Red"
End Sub
```

---

```
Private Sub tglWhite_GotFocus()
    lblOutput.ForeColor = vbWhite
    lblOutput.Caption = "White"
End Sub
```

---

```
Private Sub tglBlue_GotFocus()
    lblOutput.ForeColor = vbBlue
    lblOutput.Caption = "Blue"
End Sub
```

Toggle buttons, check boxes, and option buttons behave similarly when used in an option group. To use one or the other is simply a preference on your part.

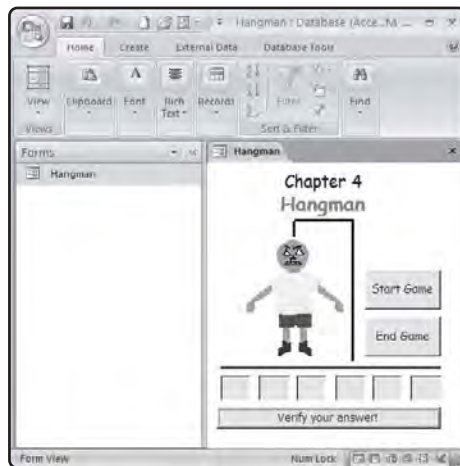


If your VBA code is not executing for an option group control, ensure the control's `On Got Focus` property has the `[Event Procedure]` value assigned.

## CHAPTER PROGRAM: HANGMAN

Hangman is a game common among school-aged children where a player tries to guess a word or phrase before a figure of a man (in this case a monster) is hanged. Each time the player guesses incorrectly, a portion of a body is shown until the body is complete, at which time the game is over. The player wins by guessing the word or phrase before all body parts are shown.

To build the Hangman program, simply construct the graphical interface, as seen in Figure 4.12. The graphic of the monster is really six different graphics files, all of which can be found on the companion website.



**FIGURE 4.12**

Using chapter-based concepts to build the Hangman program.

Controls and properties to build the Hangman program are described in Table 4.7.

All of the code required to build the Hangman program is seen next.

Option Compare Database  
Option Explicit

```
' Form level variables to track game results.
Dim iCounter As Integer
Dim letter1 As String
```

**TABLE 4.7 CONTROLS AND PROPERTIES OF THE HANGMAN PROGRAM**

Control	Property	Property Value
Form	Name	Hangman
	Caption	Hangman
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No
Label	Name	lblTitle1
	Caption	Chapter 4
Label	Name	lblTitle2
	Caption	Hangman
Line	Name	Line0
	BorderStyle	Solid
	Border Color	0
	Border Width	2 pt
Line	Name	Line1
	Border Style	Solid
	Border Color	0
	Border Width	2 pt
Line	Name	Line2
	Border Style	Solid
	Border Color	0
	Border Width	2 pt
Line	Name	Line3
	Border Style	Solid
	Border Color	0
	Border Width	2 pt
Command Button	Name	cmdStart
	Caption	Start Game
Command Button	Name	cmdQuit
	Caption	End Game
Command Button	Name	cmdVerify
	Caption	Verify your answer!
Text Box	Name	txtA
Text Box	Name	txtB
Text Box	Name	txtC
Text Box	Name	txtD
Text Box	Name	txtE
Text Box	Name	txtF
Image	Name	imgHead
	Picture	head.gif

Control	Property	Property Value
Image	Size Mode	Stretch
	Name	imgBody
	Picture	body.gif
Image	Size Mode	Stretch
	Name	imgLeftArm
	Picture	left_arm.gif
Image	Size Mode	Stretch
	Name	imgRightArm
	Picture	right_arm.gif
Image	Size Mode	Stretch
	Name	imgLeftLeg
	Picture	left_leg.gif
Image	Size Mode	Stretch
	Name	imgRightLeg
	Picture	right_leg.gif
	Size Mode	Stretch

```

Dim letter2 As String
Dim letter3 As String
Dim letter4 As String
Dim letter5 As String
Dim letter6 As String

```

---

```

Private Sub cmdStart_Click()
    MsgBox "A five letter word for database.", , "Hangman"

    'Reset the game board
    iCounter = 0

    Me.cmdVerify.Enabled = True

    Me.imgHead.Visible = False
    Me.imgBody.Visible = False
    Me.imgLeftArm.Visible = False
    Me.imgRightArm.Visible = False
    Me.imgLeftLeg.Visible = False

```



```
Me.imgRightLeg.Visible = False
```

```
Me.txtA.Enabled = True
```

```
Me.txtB.Enabled = True
```

```
Me.txtC.Enabled = True
```

```
Me.txtD.Enabled = True
```

```
Me.txtE.Enabled = True
```

```
Me.txtF.Enabled = True
```

```
Me.txtA.Value = ""
```

```
Me.txtB.Value = ""
```

```
Me.txtC.Value = ""
```

```
Me.txtD.Value = ""
```

```
Me.txtE.Value = ""
```

```
Me.txtF.Value = ""
```

```
End Sub
```

---

```
Private Sub cmdVerify_Click()
```

```
    ' Did the user win?
```

```
    If (Me.txtA.Value & Me.txtB.Value & Me.txtC.Value & _  
        Me.txtD.Value & Me.txtE.Value & Me.txtF.Value) _  
        = "Access" Then
```

```
        MsgBox "You won!", , "Hangman"
```

```
        Me.cmdStart.SetFocus
```

```
        Me.cmdVerify.Enabled = False
```

```
    Else
```

```
        ' User did not guess the correct letter.
```

```
        ' Find an available body part to display.
```

```
        If Me.imgLeftLeg.Visible = False Then
```

```
            Me.imgLeftLeg.Visible = True
```

```
            iCounter = iCounter + 1
```

```
        ElseIf Me.imgRightLeg.Visible = False Then
```

```
            Me.imgRightLeg.Visible = True
```

```
            iCounter = iCounter + 1
```

```
        ElseIf Me.imgBody.Visible = False Then
```

```

        Me.imgBody.Visible = True
        iCounter = iCounter + 1
    ElseIf Me.imgLeftArm.Visible = False Then
        Me.imgLeftArm.Visible = True
        iCounter = iCounter + 1
    ElseIf Me.imgRightArm.Visible = False Then
        Me.imgRightArm.Visible = True
        iCounter = iCounter + 1
    ElseIf Me.imgHead.Visible = False Then
        Me.imgHead.Visible = True
        iCounter = iCounter + 1
    End If
    ' Find out if the user has lost.
    If iCounter = 6 Then
        MsgBox "Sorry, you lost.", , "Hangman"
        Me.txtA.Enabled = False
        Me.txtB.Enabled = False
        Me.txtC.Enabled = False
        Me.txtD.Enabled = False
        Me.txtE.Enabled = False
        Me.txtF.Enabled = False
    Else
        MsgBox "You have " & 6 - iCounter & _
            " chances left!", , "Hangman"
    End If

```

```
End If
```

```
End Sub
```

---

```
Private Sub Form_Load()
```

```

    ' Start the game by calling an event procedure
    cmdStart_Click

```

```
End Sub
```

---

```
Private Sub txtA_LostFocus()  
    ' Ensure correct case  
    If Me.txtA.Value = "a" Then  
        Me.txtA.Value = "A"  
    End If  
End Sub
```

---

```
Private Sub txtB_LostFocus()  
    ' Ensure correct case  
    If Me.txtB.Value = "C" Then  
        Me.txtB.Value = "c"  
    End If  
End Sub
```

---

```
Private Sub txtC_LostFocus()  
    ' Ensure correct case  
    If Me.txtC.Value = "C" Then  
        Me.txtC.Value = "c"  
    End If  
End Sub
```

---

```
Private Sub txtD_LostFocus()  
    ' Ensure correct case  
    If Me.txtD.Value = "E" Then  
        Me.txtD.Value = "e"  
    End If  
End Sub
```

---

```
Private Sub txtE_LostFocus()  
    ' Ensure correct case  
    If Me.txtE.Value = "S" Then  
        Me.txtE.Value = "s"  
    End If  
End Sub
```

---

```
Private Sub txtF_LostFocus()  
    ' Ensure correct case  
    If Me.txtF.Value = "S" Then  
        Me.txtF.Value = "s"  
    End If  
End Sub
```

---

```
Private Sub cmdQuit_Click()  
    DoCmd.Quit  
End Sub
```

## SUMMARY

- Expressions can be used to build conditions that evaluate to True or False.
- VBA conditions are built with If blocks and Select Case structures.
- Compound conditions have two or more conditions and are built using the operators And, Or, and Not.
- The Select Case structure is useful for checking an expression against a list of values.
- The Case statements in a Select Case structure can check a single value, multiple values, or a range of values.
- VBA contains the built-in functions MsgBox and InputBox for building dialog boxes.
- The MsgBox function returns an integer value, whereas the InputBox function returns a string.
- The option group control contains a useful wizard for building groups of option buttons, check boxes, and toggle buttons.
- In an option group, a user can select only one check box, toggle button, or option button at a time.

## PROGRAMMING CHALLENGES

1. **Construct a simple math quiz that asks a user to answer a math problem of your choice. On the form, place one text box (txtAnswer) and two command buttons (cmdAskQuestion and cmdVerifyAnswer). Store the correct answer as a module-level constant and assign the user's answer in a local or procedure-level variable. Write code in the Click event of one command button to display a math question to the user with a message box. Write code in the other command button's Click event to compare the user's response to the module-level constant and inform the user of the results (correct or incorrect) also using a message box.**
2. **Construct another quiz program, this time using an input box to ask the question and return the user's answer. Reveal the user's result in the form of a message box. Remember to check for an empty string (user clicks the Cancel button) before checking the user's response.**
3. **Enhance the Hangman program to allow the player multiple chances to win. More specifically, display a message box that gives the player a Yes or No option to restart the game only if the game were lost.**
4. **Create a simple word processor that allows a user to enter text into a large text box. (Hint: Set the Enter Key Behavior property of a text box to New Line in Field.) The user should be able to change the foreground and background colors of the text box using option buttons in an option frame. Also, the user should be able to change the font size of the text box using option buttons in another option frame.**



# LOOPING STRUCTURES

**I**n this chapter I show you how to build iteration into your programs using VBA looping structures such as `Do` and `For` loops. In addition, you will learn some new VBA controls for managing groups of items and how to build random numbers into your programs.

## INTRODUCTION TO LOOPING STRUCTURES

To *loop*, or iterate, computers need instructions known as *looping structures*, which determine such things as how many times a loop's statements will execute and by what condition the loop exits. Each programming language implements its own version of looping structures, but most languages, including VBA, support some variation of `Do` and `For` loops. Although the syntax of looping structures varies from language to language, looping structures share similar characteristics:

- Loops are logical blocks that contain other programming statements.
- Loops can increment a counter.
- Loops implement a condition by which the loop exits.
- Many looping structures support conditions at either the top or bottom of the loop.
- Special statements can cause the loop to exit prematurely.

Before looking at specific VBA implementations, I discuss some possibilities for looping, some of which may not be so apparent at first. Consider the following list of programming scenarios, each of which requires the use of looping structures:

- Displaying a menu
- Running an autopilot system for a jumbo jet
- Finding a person's name in an electronic phone book
- Controlling a laser-guided missile
- Applying a 5% raise to all employees in a company
- Spinning the wheels in an electronic slot machine

All of the preceding scenarios have already been implemented by programmers using techniques and concepts similar to the ones I show you in this chapter.

Some scenarios require a predefined number of iterations. For example, if I write a software program to apply a 5% raise to all employees in a company, I can be sure there are a limited number of iterations, or so at least the CEO hopes! In other words, the number of times the loop executes is directly related to the number of employees in the company. Displaying a menu, however, can be a much different scenario. Take an ATM (automated teller machine) menu, for example. After a customer withdraws money from the ATM, should the ATM menu display again for the next customer? You know the answer is yes, but then how many times should that same menu display and for how many customers? The answer is indefinitely. It doesn't happen often, but there are times when a loop needs to be infinite.



Infinite loops are created when a loop's terminating condition is never met:

```
Do While 5 = 5
```

```
    MsgBox "Infinite loop"
```

```
Loop
```

In the previous code example, the loop will never terminate because the expression `5 = 5` will always be `True`. To break out of an endless loop in VBA, try pressing the `Esc` key or `Ctrl+Break` keys simultaneously.

To ensure loops are not endless, each loop has a condition that must be met for the loop to stop iterating. It's important to note that loops use expressions to build conditions, just as an `If` block or `Select Case` structure does. Moreover, each loop's condition evaluates to either `True` or `False`.

Many times, a loop's exiting condition is determined by a counter that is either incremented or decremented. In VBA, numbers are incremented and decremented using VBA assignment statements. In a nutshell, you must reassign a variable to itself with either an increment or decrement expression.

```
' Increment x by 1  
x = x + 1
```

```
' Decrement y by 1  
y = y - 1
```

After this cursory overview on looping structures, you're now ready to look at some specific VBA implementations. Specifically, you learn about the following VBA looping structures:

- Do While
- Do Until
- Loop While
- Loop Until
- For

## Do While

The `Do While` loop uses a condition at the top of the loop to determine how many times statements inside the loop will execute. Because the loop's condition is checked first, it is possible that the statements inside the loop never execute.

```
Dim x As Integer  
Dim y As Integer
```

```
Do While x < y
```

```
    MsgBox "x is less than y"
```

```
Loop
```

In this loop, it's possible that `x` is less than `y`, preventing the statement inside the loop from ever executing.

In the `Do While` loop, the statements are executed so long as the condition evaluates to `True`. In other words, the loop stops when the condition is `False`.



In the next example, the `Do While` loop uses an increment statement to satisfy the condition, which allows the loop to iterate five times.

```
Dim x As Integer
Dim y As Integer

x = 0
y = 5

Do While x < y

    MsgBox "The value of x is " & x
    x = x + 1
```

Loop

Knowing that the loop executes five times, what do you think the value of `x` is after the last iteration? The value of `x` is 4 after the last iteration. If you're having trouble seeing this, try placing this code in the `Click` event of a command button so you can step through it one iteration at a time, with each click of the command button.

## Do Until

Similarly to the `Do While` loop, the `Do Until` loop uses reverse logic to determine how many times a loop iterates.

```
Dim x As Integer
Dim y As Integer

Do Until x > y

    MsgBox "Infinite loop!"
```

Loop

In the preceding example, the statement inside the loop executes until `x` is greater than `y`. Since there is no incrementing of `x` or decrementing of `y`, the loop is infinite.

Note that `x` being equal to `y` does not satisfy the condition. In other words, the statement still executes inside the loop. Only when `x` is greater than `y` does the looping process stop.

Now study the following code and determine how many times the loop iterates and what the value of `x` is after the loop terminates.

```
Dim x As Integer
Dim y As Integer

x = 0
y = 5

Do Until x > y

    MsgBox "The value of x is " & x
    x = x + 1

Loop
```

This `Do Until` loop executes six times, and the value of `x` after the last iteration is 6.

## Loop While

Say I wanted to make sure the statements inside a loop execute at least once despite the loop's condition. The `Loop While` loop solves this dilemma by placing the loop's condition at the bottom of the loop:

```
Dim x As Integer
Dim y As Integer

x = 5
y = 2

Do

    MsgBox "Guaranteed to execute once."

Loop While x < y
```

Because the preceding loop's condition executes last, the statement inside the loop is guaranteed to execute at least once, and in this case, only once.

## Loop Until

Using logic combined from the `Do Until` and `Loop While` loops, the `Loop Until` loop uses a reverse logic condition at the end of its looping structure.

```
Dim x As Integer
Dim y As Integer

x = 5
y = 2

Do

    MsgBox "How many times will this execute?"
    y = y + 1

Loop Until x < y
```

Using an increment statement, this loop's statements execute four times. More specifically, the loop iterates until `y` becomes 6, which meets the loop's exit condition of `True`.

## For

The next and last type of looping structure this chapter investigates is `For`. The `For` loop is very common for iterating through a list. It uses a range of numbers to determine how many times the loop iterates.

```
Dim x As Integer

For x = 1 To 5

    MsgBox "The value of x is " & x

Next x
```

Notice that you are not required to increment the counting variable (in this case variable `x`) yourself. The `For` loop takes care of this for you with the `Next` keyword.



Although it is common to specify the counting variable after the `Next` keyword, it is not required. When used by itself, the `Next` keyword automatically increments the variable used on the left side of the assignment in the `For` loop.

Using the `For` loop, you can dictate a predetermined range for the number of iterations:

```
Dim x As Integer

For x = 10 To 20

    MsgBox "The value of x is " & x

Next x
```

You can determine how the `For` loop increments your counting variable using the `Step` keyword. By default, the `Step` value is 1.

```
Dim x As Integer

For x = 1 To 10 Step 2

    MsgBox "The value of x is " & x

Next x
```

The preceding `For` loop, using the `Step` keyword with a value of 2, iterates five times, with the last value of `x` being 9.

Though it is common to use number literals on both sides of the `To` keyword, you can also use variables or property values:

```
Dim x As Integer

For x = Val(Text1.Value) To Val(Text2.Value)

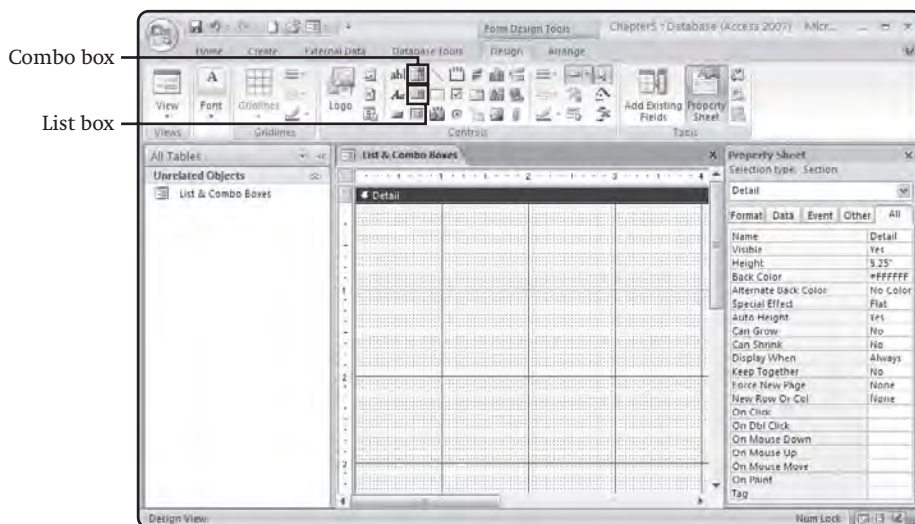
    MsgBox "The value of x is " & x

Next x
```

Using the value of two text boxes, I can build a dynamic `For` loop that obtains its looping range from a user.

## LIST AND COMBO BOXES

Both list and combo boxes store a list of items defined in design time, in runtime with VBA, or through a linked database form such as Access. Shown in Figure 5.1, list and combo boxes can be added to your forms using the Toolbox.



**FIGURE 5.1**

Viewing the  
combo box and list  
box from the VBA  
Toolbox.

Whether you are building list and combo boxes manually or through a wizard, you must take into account many properties. The most common properties are listed in Table 5.1.

**TABLE 5.1 COMMON LIST AND COMBO BOX PROPERTIES**

Property	Description
ColumnCount	Specifies the number of columns
ColumnHeads	Determines whether the list or combo box has column headings
ColumnWidths	Determines the width of each column, separated by semicolons
ListCount	Determines the number of rows in the list or combo box
ListIndex	Identifies the item selected in the list or combo box
ListRows	Specifies the maximum number of rows to display
MultiSelect	Specifies whether the user can select more than one row at a time
RowSource	Determines the entries in the list, separated by semicolons
RowSourceType	Specifies the type of row: Table/Query, Value List, Field List, or a Visual Basic function

In addition to common properties, both list and combo boxes share two important methods for adding and removing items. In VBA, they are appropriately called `AddItem` and `RemoveItem`.

In the next subsections, you learn the most common approaches for managing items in both list and combo boxes.

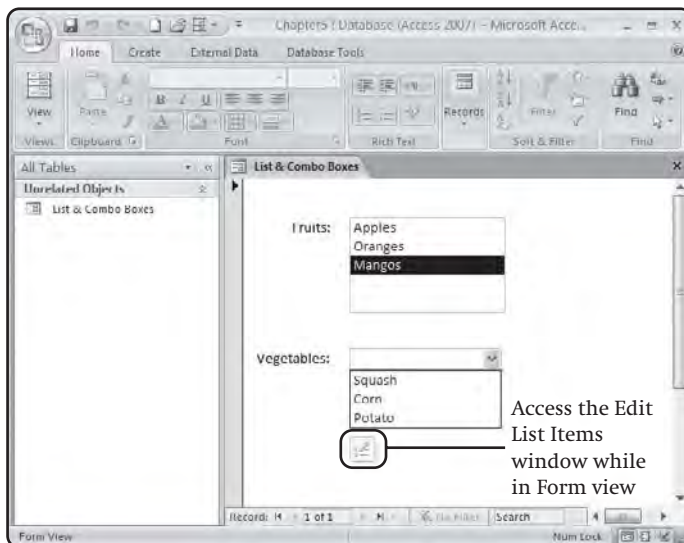
## Adding Items

Depending on the source, adding items with VBA can be a bit different between list and combo boxes. When used in a straightforward manner, however, both list and combo boxes support the `AddItem` method. Before using the `AddItem` method, you must set your combo or list box's `RowSourceType` property to `Value List`.



**CAUTION** Forgetting to set your list or combo box's `RowSourceType` property to `Value List` causes a runtime error when using the `AddItem` method.

The next program, seen in Figure 5.2, uses the form `Load` event to execute multiple `AddItem` methods for both a combo and a list box.



**FIGURE 5.2**

Using the `AddItem` method to populate a list box with fruit and a combo box with vegetables.

```
Private Sub Form_Load()  
  
    lstFruits.AddItem "Apples"  
    lstFruits.AddItem "Oranges"  
    lstFruits.AddItem "Mangos"  
  
    cboVegetables.AddItem "Squash"  
    cboVegetables.AddItem "Corn"  
    cboVegetables.AddItem "Potato"  
  
End Sub
```



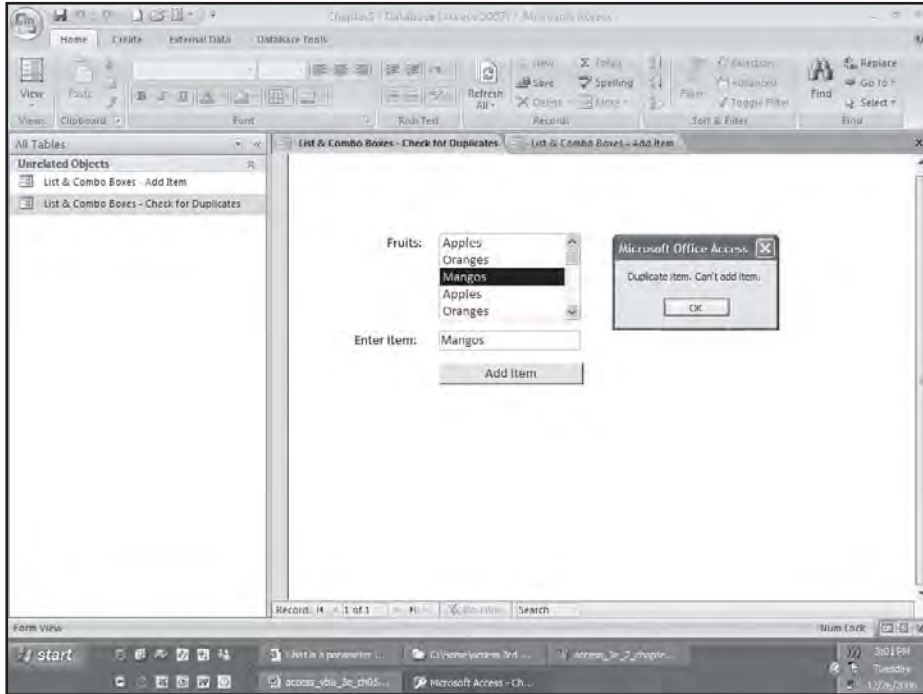
New to Access 2007 is the Edit List Items icon (seen in Figure 5.2) that appears when a cursor enters the list or combo box. When clicked, an Edit List Items window appears that enables you to add, remove, and set the default value of a list or combo box without VBA while in Form view. This feature can be turned off by setting the list or combo box's `Allow Value List Edits` property to `No`.

The `AddItem` method takes two parameters (`Item` and `Index`), the first of which is required.

Many times, loops populate list and combo boxes with static data or table information from a database.

```
Private Sub Form_Load()  
  
    Dim x As Integer  
  
    ' Add 25 items to the list box.  
    For x = 1 To 25  
  
        lstFruits.AddItem "The value of x is " & x  
  
    Next x  
  
End Sub
```

So far, adding items has been fairly static. To make things more interesting, you can add items to your list box or combo box based on user input. Before doing so, however, it's fairly common to check for duplicates before adding the item, which the following program and its output in Figure 5.3 demonstrate.

**FIGURE 5.3**

Checking for a duplicate item before adding the item to a list box.

Option Compare Database  
Option Explicit

---

```
Private Sub Form_Load()
```

```
    ' Add preliminary items to the list box.
    lstFruits.AddItem "Apples"
    lstFruits.AddItem "Oranges"
    lstFruits.AddItem "Mangos"
```

```
End Sub
```

---

```
Private Sub cmdAddItem_Click()
```

```
    Dim iCounter As Integer
```



```
' Search for a duplicate item.
' If none is found, add the item.
For iCounter = 0 To (lstFruits.ListCount - 1)

    If lstFruits.ItemData(iCounter) = txtInput.Value Then

        MsgBox "Duplicate item. Can't add item."
        Exit Sub ' A duplicate was found, exit this procedure.

    End If

Next iCounter

' No duplicate found, adding the item.
lstFruits.AddItem txtInput.Value

End Sub
```

A few statements may appear new to you in the preceding program code. First, note the use of the `ListCount` property in the `For` loop. The `ListCount` property contains the number of items in a list or combo box. This number starts at 1, but the lowest number in a list box starts with 0. This is why I subtract 1 from the `ListCount` property in the `For` loop statement.

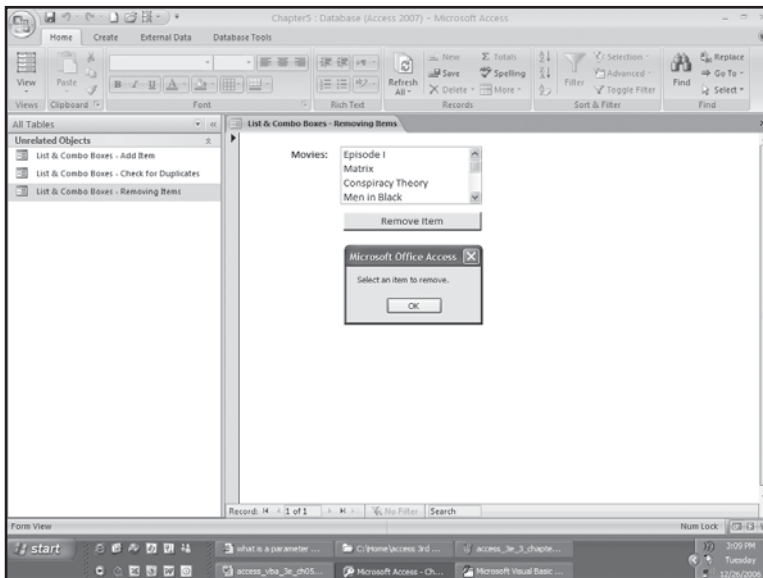
To compare what's in the text box to each item in the list box, I can use the list box's `ItemData` property, which takes an index (in this case the looping variable) as a parameter and passes back the item's value.

Last but not least is the presence of the `Exit Sub` statement. This statement is very common with Visual Basic and VBA programmers when needing to exit a procedure prematurely. In my case, I want to exit the procedure prematurely (of course, after letting the user know) if a duplicate item is found.

If I choose to use a combo box when accepting input from a user, an additional text box control is not needed because the combo box already contains a text box. In reality a combo box is really two controls: a list box and text box. To accept new input from a user with a combo box, your combo box's `LimitToList` property must be set to `No` (the default). When retrieving user input from the combo box, work with its `Value` or `Text` properties (which is similar to working with a text box).

## Removing Items

Removing items from a list or combo box is quite easy. Specifically, you use the `RemoveItem` method, which takes a single parameter called `Index` as a value. Generally speaking, items are removed based on a user's selection. Before removing items, however, it is always a good idea to ensure that a user has selected an item first. To do so, simply check the list or combo box's `ListIndex` property, as the next program demonstrates. Output is seen in Figure 5.4.



**FIGURE 5.4**

Checking that an item has been selected before using the `RemoveItem` method.

Option Compare Database  
Option Explicit

```
Private Sub Form_Load()

    ' Add preliminary items to the list box.
    lstMovies.AddItem "Episode I"
    lstMovies.AddItem "Matrix"
    lstMovies.AddItem "Conspiracy Theory"
    lstMovies.AddItem "Men in Black"

End Sub
```

```
Private Sub cmdRemoveItem_Click()  
  
    ' Has the user selected an item first?  
    If lstMovies.ListIndex = -1 Then  
  
        ' The user has not selected an item.  
        MsgBox "Select an item to remove."  
  
    Else  
  
        ' The user selected an item, so remove it.  
        lstMovies.RemoveItem lstMovies.ListIndex  
  
    End If  
  
End Sub
```

If no items in a list or combo box have been selected, the `ListIndex` is set to `-1`. Otherwise, the `ListIndex` contains the index of the currently selected item (starting with index 0 for the first item). This means you can pass the `ListIndex` property value to the `RemoveItem` method. This is an efficient and dynamic means of using property values to pass parameters to methods.

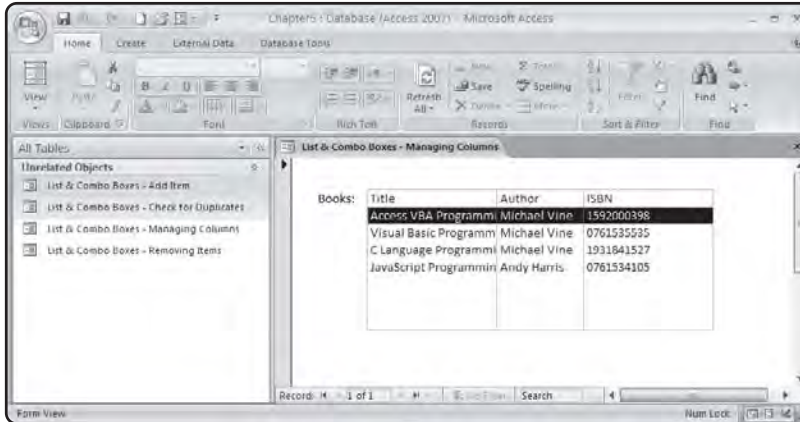
## Managing Columns

Adding and managing columns with your list and combo boxes is really quite easy. The important rule to remember is that a column header is considered a row or item (index 0) in a list or combo box. This means the header must be treated as an extra item when deleting items or searching for items.

To add and manage columns, you work with the following properties either in design time or through VBA code in runtime:

- **ColumnCount:** Specifies the number of columns to display
- **ColumnHeads:** Determines whether the list or combo box has a column header
- **ColumnWidths:** Specifies the width of each column (separated by semicolons) in inches or centimeters

Remembering that a column header is treated like another item, a column header is added using the list or combo box's `AddItem` method. Figure 5.5 depicts the visual appearance of columns and column headers.

**FIGURE 5.5**

Adding columns  
and column  
headers to a list  
box.

Option Compare Database  
Option Explicit

Private Sub Form\_Load()

```
1stBooks.ColumnCount = 3
1stBooks.ColumnHeads = True
1stBooks.ColumnWidths = "1.5in;1in;1in"

1stBooks.AddItem "Title;Author;ISBN"
1stBooks.AddItem "Access VBA Programming..." & _
    "Michael Vine;1592000398"
1stBooks.AddItem "Visual Basic Programming..." & _
    "Michael Vine;0761535535"
1stBooks.AddItem "C Language Programming..." & _
    "Michael Vine;1931841527"
1stBooks.AddItem "JavaScript Programming..." & _
    "Andy Harris;0761534105"
```

End Sub

When the `ColumnHeads` property is set to `True`, the first `AddItem` method encountered by VBA is used to populate the column headers. Note that when working with columns, you have to remember to separate each column or column data with semicolons.



Do not confuse the `ColumnHeads` property with the singular version, `ColumnHead`. They are different properties belonging to completely different controls.

## RANDOM NUMBERS

One of my favorite beginning programming concepts is random numbers. Random numbers allow you to build a wide variety of applications ranging from encryption to games. In this section, I show you how to build and use random numbers in your programs using two VBA functions called `Randomize` and `Rnd`.

The `Randomize` function initializes VBA's internal random number generator. It is only necessary to call this function (no argument required) once during the lifetime of your program. Most often, the `Randomize` function is executed during startup routines such as a form `Load` event. Without the `Randomize` function, the `Rnd` function generates random numbers in a consistent pattern, which of course is not really random at all.

The `Rnd` function takes a number as an argument and returns a `Single` data type. When used in conjunction with the `Randomize` function, the `Rnd` function can generate random numbers. To create a range of Integer-based random numbers, use the following VBA statements:

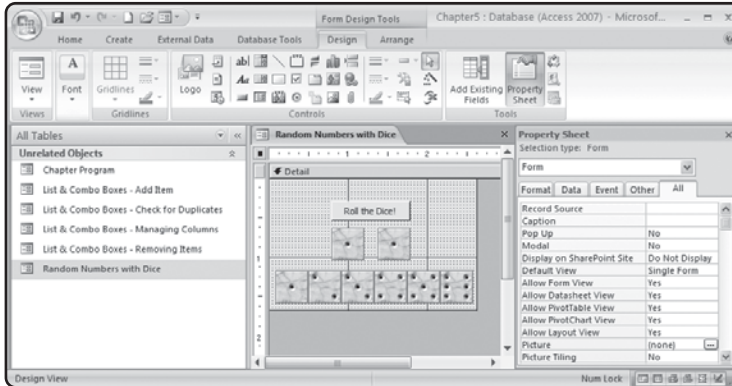
```
Dim x as Integer  
x = Int((10 * Rnd) + 1)
```

The `Int` function takes a number as argument and converts it to an integer value (whole number). Remember that the `Rnd` function returns a `Single` number type, so the `Int` function is used to convert a decimal number into a whole number. Adding 1 to the result of  $(10 * \text{Rnd})$  creates a random number between 1 and 10. Removing the addition of 1 causes the random number range to be 0 through 9.

One way of utilizing random numbers is through a simulated dice roll. Figure 5.6 reveals the design-time form I used to simulate this. Note that there are eight images (can be found on the companion website), six of which have their `Visible` property set to `False`. This way only two dice are visible to the user during runtime.



Set the `Size Mode` property of the image control to `Stretch`, which stretches the image to fit the size of the image control.

**FIGURE 5.6**

Using random numbers and image-swapping techniques to emulate rolling of the dice.

Option Compare Database  
Option Explicit

```
Private Sub Form_Load()  
    Randomize  
End Sub
```

```
Private Sub cmdRoll_Click()  
  
    Dim iRandomNumber As Integer  
  
    ' Generate random number (die) for die 1.  
    iRandomNumber = Int((6 * Rnd) + 1)  
  
    Select Case iRandomNumber  
  
        Case 1  
            imgDie1.Picture = Image1.Picture  
        Case 2  
            imgDie1.Picture = Image2.Picture  
        Case 3  
            imgDie1.Picture = Image3.Picture  
        Case 4  
            imgDie1.Picture = Image4.Picture  
        Case 5
```

```
        imgDie1.Picture = Image5.Picture
    Case 6
        imgDie1.Picture = Image6.Picture

End Select

' Generate random number (die) for die 2.
iRandomNumber = Int((6 * Rnd) + 1)

Select Case iRandomNumber

    Case 1
        imgDie2.Picture = Image1.Picture
    Case 2
        imgDie2.Picture = Image2.Picture
    Case 3
        imgDie2.Picture = Image3.Picture
    Case 4
        imgDie2.Picture = Image4.Picture
    Case 5
        imgDie2.Picture = Image5.Picture
    Case 6
        imgDie2.Picture = Image6.Picture

End Select

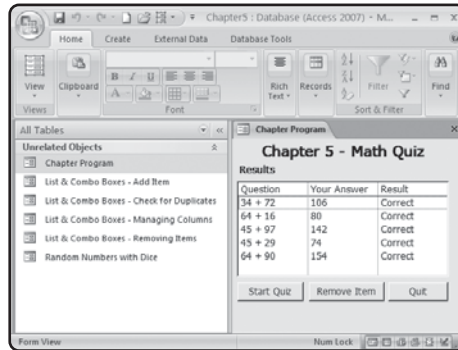
End Sub
```

To simulate the dice roll, I use the `Click` event of a command button to create a random number ranging from 1 to 6 for each die. After that, I perform a bit of image swapping based on a `Select Case` structure. Image swapping in this case is performed by assigning one `Picture` property to another. Remember, to work with programs that use images found on the website, you will need to change the path of the image's `Picture` property to a path on your local computer.

## CHAPTER PROGRAM: MATH QUIZ

The Math Quiz program in Figure 5.7 is a fun way of learning how to incorporate chapter-based concepts, such as loops, random numbers, and list boxes, into your VBA applications.

The program prompts a user for the number of math questions she would like to answer. Then, Math Quiz prompts the user with a predetermined number of addition questions using random numbers between 1 and 100. A list box with columns stores each question, the user's response, and the result.

**FIGURE 5.7**

Using chapter-based concepts to build the Math Quiz program.

Controls and properties to build the Math Quiz program are described in Table 5.2.

**TABLE 5.2 CONTROLS AND PROPERTIES OF THE MATH QUIZ PROGRAM**

Control	Property	Property Value
Form	Name	Chapter Program
	Caption	Chapter Program
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No
Label	Name	lblTitle
	Caption	Chapter 5 - Math Quiz
Label	Name	lblResults
	Caption	Results
List Box	Name	lstResults
	Row Source Type	Value List
	Column Count	3
	Column Heads	Yes
Command Button	Name	cmdStart
	Caption	Start Quiz
Command Button	Name	cmdRemoveItem
	Caption	Remove Item
Command Button	Name	cmdQuit
	Caption	Quit



All of the code required to build Math Quiz is seen next.

Option Compare Database  
Option Explicit

---

```
Private Sub cmdQuit_Click()  
    DoCmd.Quit  
End Sub
```

---

```
Private Sub Form_Load()  
    Randomize  
End Sub
```

---

```
Private Sub cmdRemoveItem_Click()  
  
    ' Determine if an item has been selected first  
    If lstResults.ListIndex = -1 Then  
        MsgBox "Select an item to remove."  
    Else  
        lstResults.RemoveItem lstResults.ListIndex + 1  
    End If  
  
End Sub
```

---

```
Private Sub cmdStart_Click()  
  
    Dim sResponse As String  
    Dim sUserAnswer As String  
    Dim iCounter As Integer  
    Dim iOperand1 As Integer  
    Dim iOperand2 As Integer  
  
    ' Determine how many math questions to ask.  
    sResponse = InputBox("How many math questions would you like?")  
  
    If sResponse <> "" Then  
  
        ' Add header to each column in the list box if one  
        ' hasn't already been added.  
        If lstResults.ListCount = 0 Then
```

```

    lstResults.AddItem "Question;Your Answer;Result"
End If

' Ask predetermined number of math questions.
For iCounter = 1 To Val(sResponse)

    ' Generate random numbers between 0 and 100.
    iOperand1 = Int(100 * Rnd)
    iOperand2 = Int(100 * Rnd)

    ' Generate question.
    sUserAnswer = InputBox("What is " & iOperand1 & _
        " + " & iOperand2)

    ' Determine if user's answer was correct and add an
    ' appropriate item to the multi-column list box.
    If Val(sUserAnswer) = iOperand1 + iOperand2 Then
        lstResults.AddItem iOperand1 & " + " & _
            iOperand2 & ";" & sUserAnswer & ";Correct"
    Else
        lstResults.AddItem iOperand1 & " + " & _
            iOperand2 & ";" & sUserAnswer & ";Incorrect"
    End If

Next iCounter

End If

End Sub

```

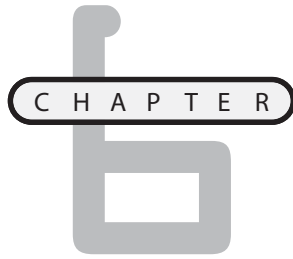
## SUMMARY

- VBA supports the Do While, Do Until, Loop While, Loop Until, and For loop structures.
- Looping structures use conditions to determine the number of iterations the loop will execute.
- An infinite, or endless, loop is caused when the loop's condition is never met.
- Generally, a loop's exiting condition is determined by a counter that is either incremented or decremented.
- The For loop uses a range of numbers to determine how many times the loop iterates.
- Loops are often used to populate list and combo boxes with static data or table information from a database.

- To use the `AddItem` method of a list or combo box, the `RowSourceType` property value must be set to `Value List`.
- The `ListIndex` property of a list and combo box can be used to determine which item a user has selected.
- If no items are selected in a list or combo box, the `ListIndex` property is set to `-1`.
- Columns can be added to list and combo boxes by setting the `ColumnCount`, `ColumnHeads`, and `ColumnWidths` properties.
- Columns are managed through runtime with VBA or through design time by separating individual columns with semicolons.
- VBA uses the `Randomize` and `Rnd` functions to generate random numbers.
- The `Randomize` function initializes VBA's internal random-number generator.
- The `Rnd` function takes a number as an argument and returns a `Single` data type.

### PROGRAMMING CHALLENGES

1. **Place a single command button on a form. Write code in the Click event of the command button to display a message box five times using a Do While loop. Remember to use a counting variable in the loop's condition, which increments each time the loop iterates.**
2. **Modify Challenge 1 to use a For loop that iterates 20 times with a Step value of 3.**
3. **Add a combo box and command button to a form. In the form's Load event, add three items to the combo box using the AddItem method. In the Click event of the command button, add input from the combo box's Value property (input from the user). Remember to check for duplicate items and turn off the Edit List Items functionality by setting the combo box's Allow Value List Edits property to No.**
4. **Enhance the Math Quiz program to randomize not only numbers but the type of math problem. More specifically, use an additional variable to hold a random number between 1 and 4, where each number represents addition, subtraction, multiplication, or division.**



# COMMON FORMATTING AND CONVERSION FUNCTIONS

So far you have learned to use some built-in VBA functions such as `InputBox` and `MsgBox`, which provide interactive dialog boxes to the user. What you might not know is that VBA provides many more intrinsic functions for you to use in your programming efforts. Learning how to leverage the power of these built-in functions is all-important in VBA programming and is certainly the key to saving you from unproductive programming time.

To facilitate your learning of VBA functions, this chapter introduces a number of commonly used functions for formatting strings, dates, time, and for converting data.

## STRING-BASED FUNCTIONS

Someone famous once asked, “What’s in a name?” Someone less famous (yours truly) once asked, “What’s in a string?” So what is in a string? Well, lots. *Strings* are key building blocks in any high-level programming language. More specifically, they are data structures that contain one or more characters. Note that it is also possible for strings to be `Null` (undefined).

Groupings of characters and numbers comprise strings. These groupings of characters can mean different things depending on their use. Many languages, including VBA, provide popular means for parsing, searching, and managing the individual pieces (characters and numbers) that make up strings.

In this section, I show you how to parse, search, and manage strings using some very popular built-in VBA functions, which are described in Table 6.1.

**TABLE 6.1    COMMON STRING-BASED FUNCTIONS**

Function Name	Description
UCase	Converts a string to uppercase
LCase	Converts a string to lowercase
Len	Returns the number of characters in a string
StrCom	Compares two strings and determines whether they are equal to, less than, or greater than each other
Right	Determines the specified number of characters from the right side of a string
Left	Determines the specified number of characters from the left side of a string
Mid	Determines the specified number of characters in a string
InStr	Finds the first occurrence of a string within another
Format	Formats a string based on specified instructions

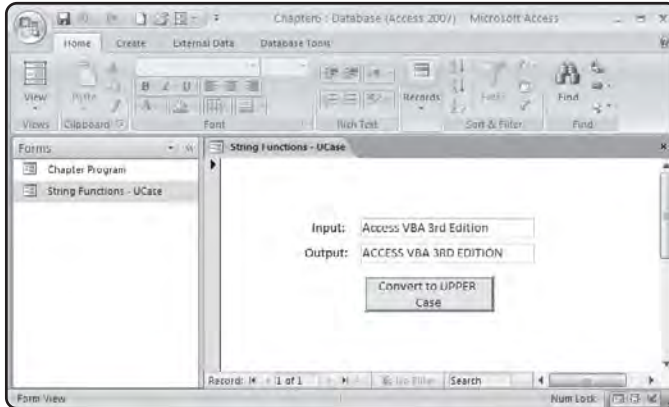
## UCase

The **UCase** function is an easy function to use. It takes a string as a parameter and returns the string in uppercase letters.

```
Private Sub cmdConvert_Click()  
  
    txtOutput.Value = UCase(txtInput.Value)
```

```
End Sub
```

The **UCase** function can take a string literal ("This is a string literal"), string variable (`Dim sFirstName As String`), or string type property (`txtFirstName.Value`) and provide output like that in Figure 6.1.

**FIGURE 6.1**

Converting a string to uppercase using the `UCase` function.

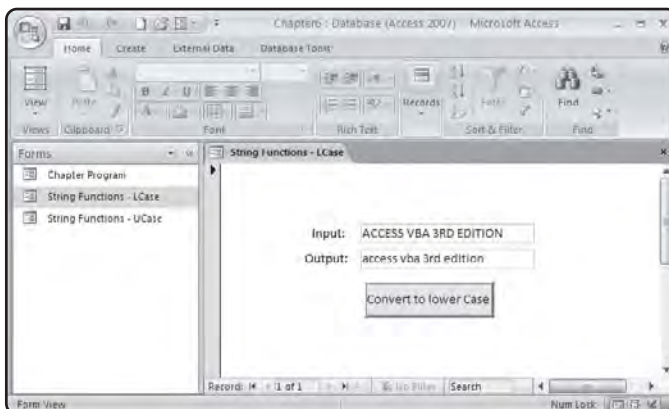
## LCase

The inverse of `UCase`, the `LCase` function takes a string parameter and outputs the string in lowercase. Sample code is demonstrated next, with output seen in Figure 6.2.

```
Private Sub cmdConvert_Click()
```

```
    txtOutput.Value = LCase(txtInput.Value)
```

```
End Sub
```

**FIGURE 6.2**

Converting a string to lowercase using the `LCase` function.

## Len

The `Len` function is a useful tool for determining the length of a string. It takes a string as input and returns a number of `Long` data type. The `Len` function's return value indicates the

number of characters present in the string parameter. To demonstrate, the next event procedure determines the number of characters in a person's name. Output can be seen in Figure 6.3.

```
Private Sub cmdNumberOfCharacters_Click()

    Dim iNumberOfCharacters As Integer

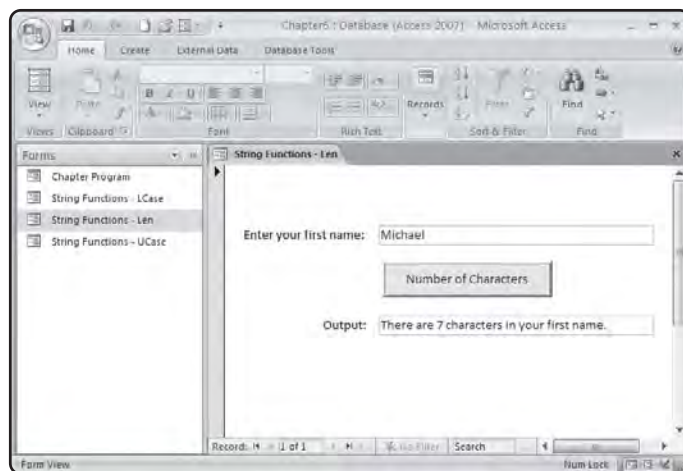
    iNumberOfCharacters = Len(txtFirstName.Value)

    txtOutput.Value = "There are " & iNumberOfCharacters & _
        " characters in your first name."

End Sub
```

**FIGURE 6.3**

Using the `Len` function to determine the number of characters in a string.



## StrComp

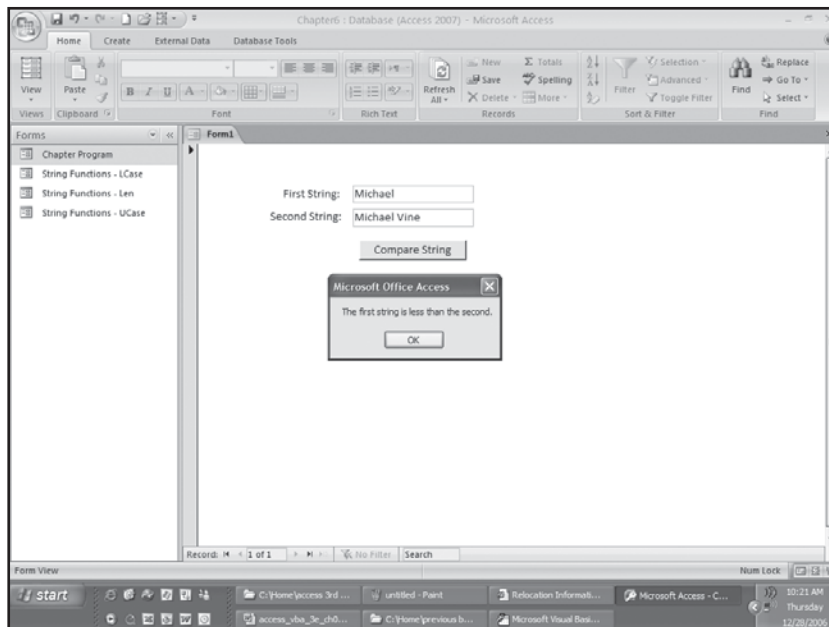
The `StrComp` function is useful when comparing the sequence of characters in two strings. The `Option Compare` statement is used to determine whether binary or textual comparison is done. If a binary comparison is done, characters are treated with case sensitivity. Textual comparisons are not case sensitive.

The `StrComp` function takes two string parameters and returns one of four values, as explained in Table 6.2.

**TABLE 6.2** OUTPUT VALUES FOR THE **StrComp** FUNCTION

Return Value	Description
-1	String 1 is less than string 2
0	Both strings are equal
1	String 1 is greater than string 2
Null	One of the strings is Null (undefined)

The next Click event procedure uses the **StrComp** function to determine the equality of two strings. Figure 6.4 examines sample output from the event procedure.

**FIGURE 6.4**

Using the **StrComp** function to compare two strings for equality.

```
Private Sub cmdCompareStrings_Click()

    Dim iResult As Integer

    iResult = StrComp(txtFirstString.Value, txtSecondString.Value)

    Select Case iResult
```



Case -1

```
MsgBox "The first string is less than the second."
```

Case 0

```
MsgBox "Both strings are equal."
```

Case 1

```
MsgBox "The first string is greater than the second."
```

Case Else

```
MsgBox "One or more strings are Null."
```

End Select

End Sub

## Right

The **Right** function takes two parameters and returns a string containing the number of characters from the right side of a string. The first parameter is the evaluated string. The second parameter is a number that indicates how many characters to return from the right side of the string. Sample output is shown in Figure 6.5.

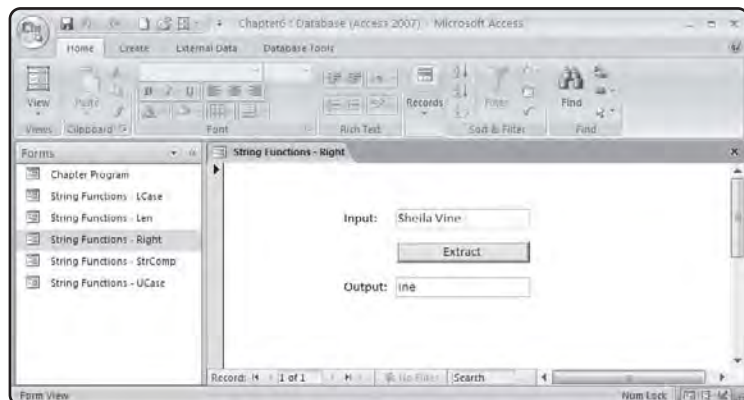
```
Private Sub cmdExtract_Click()
```

```
txtOutput.Value = Right(txtInput.Value, 3)
```

```
End Sub
```

**FIGURE 6.5**

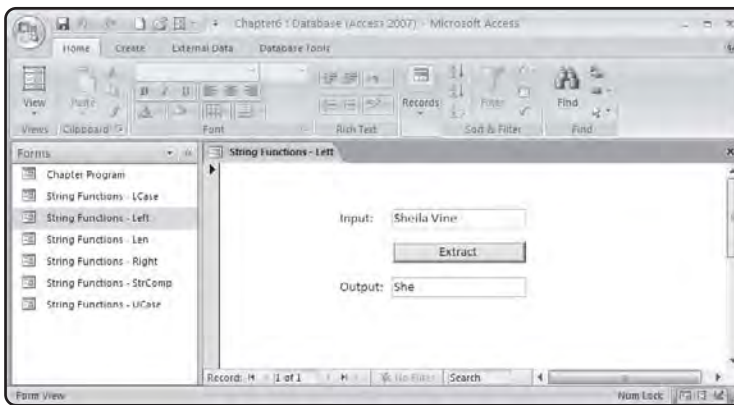
Extracting three characters from the right side of a string.



## Left

Working in the opposite direction of the **Right** function, the **Left** function extracts a predetermined number of characters from the left side of a string. Like the **Right** function, the **Left** function takes two parameters. The first parameter is the evaluated string. The second parameter is a number that indicates how many characters to return from the left side of the string. Sample output is seen in Figure 6.6.

```
Private Sub cmdExtract_Click()  
    txtOutput.Value = Left(txtInput.Value, 3)  
End Sub
```



**FIGURE 6.6**

Extracting three characters from the left side of a string.

## Mid

The **Mid** function returns a string containing a predetermined number of characters. It takes three parameters, the first two of which are required. The first parameter is the evaluated string. The next parameter is the starting position from which characters should be taken. The last parameter, which is optional, is the number of characters to be returned. If the last parameter is omitted, all characters from the starting position to the end of the string are returned.

```
Dim sString As String  
Dim sMiddleWord As String
```

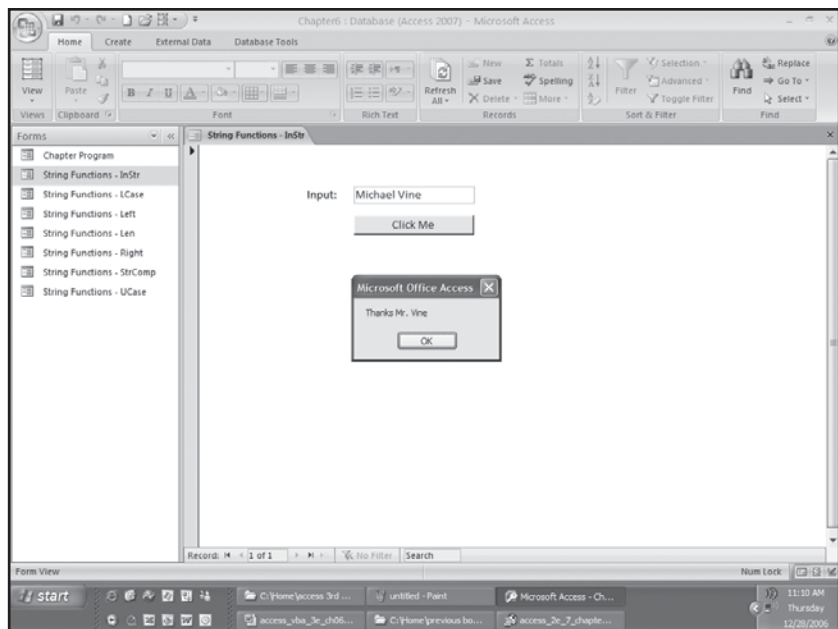
```
sString = "Access VBA Programming"
```

```
sMiddleWord = Mid(sString, 8, 3) ' Returns "VBA"
```

## InStr

The **InStr** function can take up to four parameters and returns a number specifying the starting position of a string's first occurrence within another string. The required parameters are two strings, where the first string is being searched and the second parameter is sought after. The optional parameters determine the starting position of the search and the type of string comparison made.

In the next code example, I use both the **Mid** and **InStr** functions to extract a person's last name from a string expression. Output is seen in Figure 6.7.



**FIGURE 6.7**

Using **Mid** and **InStr** functions to extract one string from another.

```
Private Sub cmdClickMe_Click()
```

```
    Dim sLastName As String
```

```
    Dim startPosition
```

```
    ' Search the input string for a space character.
    startPosition = InStr(txtInput.Value, " ")
```

```
    ' Extract the last name from the string starting
    ' after the space character.
```

```
sLastName = Mid(txtInput.Value, startPosition + 1)
```

```
MsgBox "Thanks Mr. " & sLastName
```

```
End Sub
```

## DATE AND TIME FUNCTIONS

Access VBA contains numerous date and time functions, such as **Date**, **Time**, and **Now**, for accessing your system's date and time. Specifically, I show you how to use the following VBA date/time functions:

- **Date**
- **Day**
- **WeekDay**
- **Month**
- **Year**
- **Time**
- **Second**
- **Minute**
- **Hour**
- **Now**

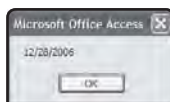
With these functions, you can create date/time stamps, stopwatches, clocks, or custom timer functions.

### Date

The **Date** function requires no parameter when executed and returns a **Variant** data type containing your system's current date. This is that code:

```
MsgBox Date
```

Figure 6.8 demonstrates sample output from the **Date** function.



**FIGURE 6.8**

Displaying the current system's date with the **Date** function.

## Day

The **Day** function takes a required date argument, such as the output from the **Date** function, and returns a whole number between 1 and 31, which represents a day within the current month. This is that code:

```
Day(Date) ' Returns a number between 1 and 31.
```

## WeekDay

The **WeekDay** function takes two parameters and returns a whole number containing the current day of the week. The first parameter is the date (**Date** function output), which is required. The second, optional parameter determines the first day of the week. This is that code:

```
WeekDay(Date) ' Returns a number between 1 and 7.
```



The default first day of the week is Sunday.

## Month

The **Month** function takes a single parameter, which signifies the current date and returns a whole number representing the current month in the year. This is that code:

```
Month(Date) ' Returns a number between 1 and 12.
```

## Year

Much like the preceding date-based functions, the **Year** function takes a date parameter and returns a whole number representing the current year. Here is that code:

```
Year(Date)
```

## Time

The **Time** function is another easy-to-use function; it requires no parameters as input and returns a **Variant** data type with your system's current time. That code is shown here:

```
MsgBox Time
```

When used with other functions and events, the **Time** function can be quite useful in building many applications. For example, I can use a form's **Timer** event and **TimerInterval** property to display the current time updated automatically every second:

```
Private Sub cmdStop_Click()

    ' Stop the Timer event.
    Me.TimerInterval = 0

End Sub

Private Sub Form_Load()

    Me.TimerInterval = 1000 ' 1000 milliseconds = 1 second

End Sub

Private Sub Form_Timer()

    ' Update the time every 1 second.
    lblTime.Caption = Time

End Sub
```

Note that setting the form's `TimerInterval` property to 0 stops the `Timer` event from executing.

## Second

The `Second` function requires a time parameter (the output from the `Time` function) and returns a whole number from 0 to 59 indicating the current second in the current minute. That code looks like this:

```
Second(Time) ' Returns a number from 0 to 59.
```

## Minute

Much like the `Second` function, the `Minute` function requires a time parameter (the output from the `Time` function) and returns a whole number from 0 to 59, which indicates the current minute in the current hour. That code looks like this:

```
Minute(Time) ' Returns a number from 0 to 59.
```

## Hour

The `Hour` function takes a required time parameter and returns a whole number between 0 and 23, which represents the current hour according to your system's time. Here is that code:

Hour(Time) ' Returns a number from 0 to 23.

## Now

The **Now** function incorporates results from both **Date** and **Time** functions. It takes no parameters and returns a **Variant** data type indicating the system's current date and time, respectively.

MsgBox Now

## CONVERSION FUNCTIONS

Conversion functions are very powerful; they allow programmers to convert data from one type to another. Access VBA supports many types of conversion functions. Many common uses for data conversion involve converting strings to numbers and numbers to strings. To explore the application of data conversion, I discuss the conversion functions found in Table 6.3.

**TABLE 6.3** COMMON VBA CONVERSION FUNCTIONS

Function	Description
Val	Converts recognized numeric characters in a string as numbers
Str	Converts a recognized number to a string equivalent
Chr	Converts a character code to its corresponding character
Asc	Converts a character to its corresponding character code

## Val

The **Val** function takes a string as input and converts recognizable numeric characters to a **number** data type. More specifically, the **Val** function stops reading the string when it encounters a nonnumeric character. The following are some sample return values:

- Val("123") ' Returns 123
- Val("a123") ' Returns 0
- Val("123a") ' Returns 123

## Str

The **Str** function takes a number as argument and converts it to a string representation with a leading space for its sign (positive or negative). An error occurs in the **Str** function if a nonnumeric value is passed as a parameter. The following are some sample return values:

- `Str(123)` ' Returns " 123"
- `Str(-123)` ' Returns "-123"

## Chr

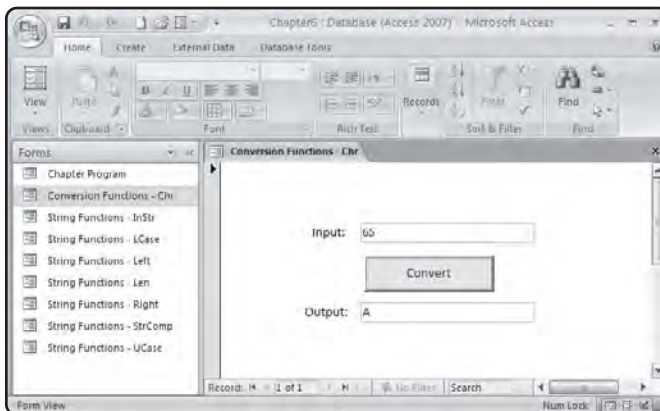
You may remember from earlier chapters that data can take many forms. Specifically, numbers can represent both numbers and characters. This means it is up to the programmer to determine how data is stored—variables and data types—and presented—formatting and conversion functions.

Many programming languages, including VBA, support the concept of character codes. *Character codes* are numbers that represent a single character. For example, the character A is represented by the character code 65, and the character a (lowercase letter A) is represented by the character code 97. Appendix A, “Common Character Codes,” contains a table of VBA’s most common character codes.

To convert a character code to its corresponding character, VBA programmers use the `Chr` function. The `Chr` function takes a single character code as a parameter and returns the corresponding character.

Figure 6.9 demonstrates a simple program (given here) that can convert a character code to its corresponding character.

```
Private Sub cmdConvert_Click()  
  
    txtOutput.Value = Chr(txtInput.Value)  
  
End Sub
```



**FIGURE 6.9**

Converting character codes to characters with the `Chr` function.



Note that character codes also represent numeric characters and nonprintable characters such as space, tab, and linefeed.

## Asc

The **Asc** function works as the inverse of the **Chr** function. It takes a single character as input and converts it to its corresponding character code.

```
Private Sub cmdConvert_Click()  
  
    txtOutput.Value = Asc(txtInput.Value)
```

```
End Sub
```

## FORMATTING

It is often necessary to format data to a specific need. For example, you may want to display a date in long format, or a number with a thousandths separator, or numbers as a currency or percentage. Each of these scenarios, and many more, can be accomplished with a single VBA function called **Format**.

The **Format** function takes up to four parameters:

```
Format(expression, format, firstDayOfWeek, firstWeekOfYear)
```

Table 6.4 describes each of the **Format** function's parameters in detail.

**TABLE 6.4    FORMAT FUNCTION PARAMETERS**

Parameter	Description
<b>expression</b>	An expression to format. Required.
<b>format</b>	A valid user-defined or named expression format. Optional.
<b>firstDayOfWeek</b>	A VBA constant that specifies the first day of the week. Optional.
<b>firstWeekOfYear</b>	A VBA constant that specifies the first week of the year. Optional.

In the next three sections, I show you how to use the **Format** function to format strings, numbers, dates, and times.

## Formatting Strings

You have five characters with which to build user-defined strings using the `Format` function. Each format character seen in Table 6.5 must be enclosed in quotation marks when passed as a format expression argument in the `Format` function.

**TABLE 6.5 STRING FORMATS**

Format Character	Description
@	Displays a character or space as a placeholder
&	Displays a character or nothing as a placeholder
<	Formats all characters in lowercase
>	Formats all characters in uppercase
!	Placeholders are filled left to right

Note that placeholders are displayed from right to left unless an exclamation point character is present in the format expression. Consult Microsoft Visual Basic Reference for more information on the `Format` function and format expressions.

In the next code segment, I use the `Format` function to change a string literal to all uppercase:

```
' Returns "HI THERE"
Format("hi there", ">")

' Returns "access vba programming"
Format("Access VBA Programming", "<")
```

## Formatting Numbers

Numbers can be displayed with user-defined formatting expressions. The `Format` function supports a multitude of formatting characters, which are used in the format argument of the `Format` function, for numbers. Table 6.6 reveals these number formats.

The next three statements demonstrate how the `Format` function can be used to format numbers as money, with decimal precision, and as percentage:

- `s = Format(12345.6, "$###,##00.00")` ' Returns \$12,345.60
- `s = Format("12345.6", "00.0")` ' Returns 12345.6
- `s = Format("10", "0.0%")` ' Returns 1000.0%

TABLE 6.6 NUMBER FORMATS

Format Character	Description
0	Displays a digit or 0 as a placeholder
#	Displays a digit or nothing as a placeholder
.	A placeholder that determines how many digits are displayed to the left and right of the decimal
%	Places the percentage character at the location where it appears in the format expression; multiplies the number by 100
,	Separates thousandths from hundreds using the comma character
E- E+ e- e+	Specifies scientific formatting
- + \$ ( )	Specifies a literal character
\\	Displays a single backslash

## Formatting Date and Time

One of the most common reasons to format data is to display dates and times. The **Format** function supports many named (VBA-defined) and user-defined formatting expressions for display customization.

Table 6.7 describes many date and time formatting options.

TABLE 6.7 DATE AND TIME FORMATS

Format Character	Description
:	Separates time in hours, minutes, and seconds
/	Separates dates in month, day, and year
d	Day displayed as a number without a leading zero
dd	Day displayed as a number with a leading zero
ddd	Day displayed as an abbreviation
dddd	Day displayed with the full name
dddddd	Complete date displayed in short format (m/d/yy)
dddddd	Complete date displayed in long format (mmm dd, yyyy)
w	Day of the week displayed as a number (1 starts on Sunday)
ww	Week of the year displayed as a number

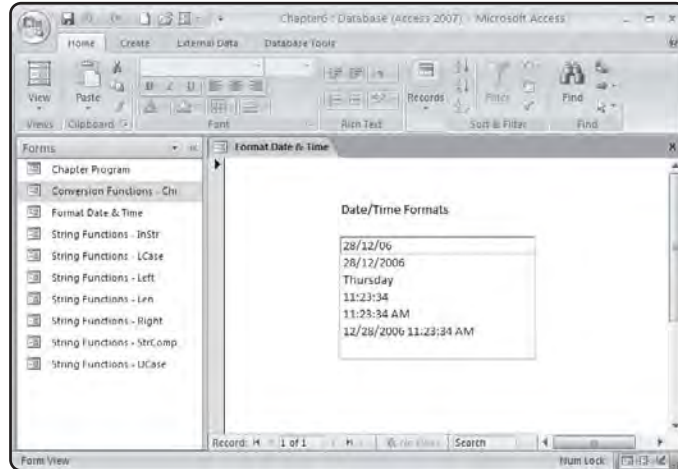
Format Character	Description
m	Month displayed as a number without a leading zero
mm	Month displayed as a number with a leading zero
mmm	Month displayed as an abbreviation
mmm	Month displayed with full name
q	Quarters in year displayed as a number
y	Day of the year displayed as a number
yy	Year displayed in two-digit format
yyyy	Year displayed in four-digit format
h	Hour displayed without leading zeros
hh	Hour in two digits (00 to 23)
n	Minute displayed in one or two digits
nn	Minute in two digits
s	Second displayed without leading zeros
ss	Second displayed with leading zeros
ttttt	Time displayed with hour, minute, and second
AM/PM	Displays uppercase AM or PM using 12-hour clock
am/pm	Displays lowercase am or pm using 12-hour clock
A/P	Displays an uppercase A or P using 12-hour clock
a/p	Displays a lowercase a or p using 12-hour clock
p	Displays the date as dddd and time as tttt
c	Same as the general predefined date format

In Figure 6.10, sample outputs are shown in a list box from formatting dates and time with the `Format` function, as follows.

```
Private Sub Form_Load()

    1stFormatDateTime.AddItem Format(Date, "d/m/yy")
    1stFormatDateTime.AddItem Format(Date, "dd/mm/yyyy")
    1stFormatDateTime.AddItem Format(Date, "dddd")
    1stFormatDateTime.AddItem Format(Time, "h:m:s")
    1stFormatDateTime.AddItem Format(Time, "hh:mm:ss AM/PM")
    1stFormatDateTime.AddItem Format(Now, "c")

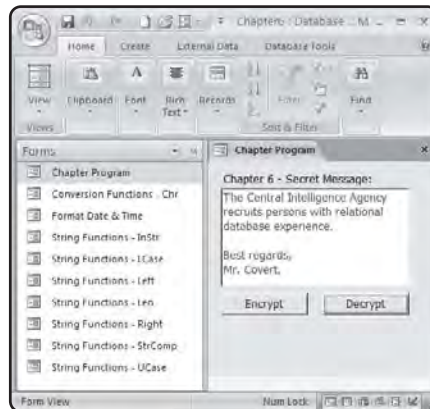
End Sub
```

**FIGURE 6.10**

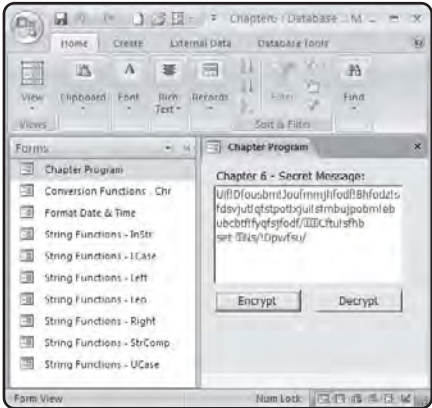
Using the **Format** function and user-defined expressions to format date and time.

## CHAPTER PROGRAM: SECRET MESSAGE

**Secret Message** uses built-in VBA functions to build a fun encryption program. Moreover, the **Secret Message** program uses string-based functions, such as **Len** and **Mid**, and conversion functions **Asc** and **Chr** to encrypt and decrypt messages. Figures 6.11 and 6.12 depict sample input and output from the **Secret Message** program.

**FIGURE 6.11**

Using chapter-based concepts to build the **Secret Message** program.



**FIGURE 6.12**  
Using chapter-based concepts to encrypt a message with the Secret Message program.

Controls and properties to build the Secret Message program are described in Table 6.8.

**TABLE 6.8    CONTROLS AND PROPERTIES OF THE SECRET MESSAGE PROGRAM**

Control	Property	Property Value
Form	Name	Chapter Program
	Caption	Chapter Program
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No
Label	Name	lblTitle
	Caption	Chapter 6 - Secret Message:
	Font Size	10
Text Box	Name	txtMessage
	Enter Key Behavior	New Line in Field
Command Button	Name	cmdEncrypt
	Caption	Encrypt
Command Button	Name	cmdDecrypt
	Caption	Decrypt

All of the code required to build the Secret Message program is shown next.

Option Compare Database  
Option Explicit

```
Private Sub cmdDecrypt_Click()

    Dim sDecryptedMessage As String
    Dim sDecryptedCharacter As String
    Dim iCounter As Integer

    If txtMessage.Value <> "" Then

        ' Iterate through each encrypted character in the message.
        For iCounter = 1 To Len(txtMessage.Value)

            ' Convert one encrypted character at a time to its
            ' equivalent character code.
            sDecryptedCharacter = Asc(Mid(txtMessage.Value, iCounter, 1))

            ' Convert the character code (shifted by -1) back
            ' to a character.
            sDecryptedCharacter = Chr(sDecryptedCharacter - 1)

            ' Add the decrypted character to the new decrypted message.
            sDecryptedMessage = sDecryptedMessage + sDecryptedCharacter

        Next iCounter

        ' Display the decrypted message.
        txtMessage.Value = sDecryptedMessage

    End If

End Sub
```

---

```
Private Sub cmdEncrypt_Click()

    Dim sEncryptedMessage As String
    Dim sEncryptedCharacter As String
    Dim iCounter As Integer
```

```
If txtMessage.Value <> "" Then

    ' Iterate through each character in the message.
    For iCounter = 1 To Len(txtMessage.Value)

        ' Convert one character at a time to its equivalent
        ' character code.
        sEncryptedCharacter = Asc(Mid(txtMessage.Value, iCounter, 1))

        ' Convert the character code (shifted by 1) back
        ' to a character.
        sEncryptedCharacter = Chr(sEncryptedCharacter + 1)

        ' Add the encrypted character to the new encrypted message.
        sEncryptedMessage = sEncryptedMessage + sEncryptedCharacter

    Next iCounter

    ' Display the encrypted message.
    txtMessage.Value = sEncryptedMessage

End If

End Sub
```

## SUMMARY

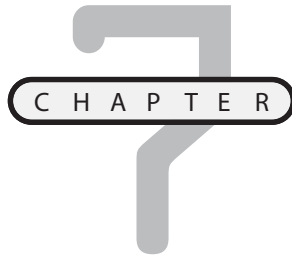
- String case can be managed with **UCase** and **LCase** functions.
- Strings can be extracted from other strings using functions such as **Left**, **Right**, and **Mid**.
- Strings can be searched and compared with VBA functions **InStr** and **StrComp**, respectively.
- VBA supports a multitude of functions, such as **Date**, **Time**, and **Now**, for displaying dates and times.
- Forms have a **Timer** event, which can be triggered automatically and regularly using the form's **TimerInterval** property.



- Data is represented by both numbers and characters using character codes. VBA uses the `Chr` and `Asc` functions to convert between character codes and characters.
- Data such as strings, numbers, and date/time can easily be formatted using VBA's `Format` function.

### PROGRAMMING CHALLENGES

1. Using the `Right` function, code the `Click` event of a command button to output the last seven characters in the string `Access VBA Programming`.
2. Using a form's `Timer` event and `TimerInterval` property, build a stopwatch with one label control and two command buttons. Use the `Format` function, `Time` function, and format expression `Ss` to display seconds only.
3. Create a word search game that allows a user to view a string of characters for a predetermined amount of time (say 5 to 10 seconds). Build a timer to accomplish this. After time is up, hide the string of characters and prompt the user to enter one or more words he saw in the string. For example, the string of characters `ke01xakaccessqc1nmsboxeam1z` contains the words `access` and `box`. Use the `InStr` function to determine whether the user's guess is contained in the word search string.
4. Build a form with one text box and one command button. Allow the user to enter multiple lines into the text box. In the `Click` event of the command button, use a `For` loop and the `Len` function to iterate through each character in the text box. Every time a space character is found, increment a procedure-level variable by 1. After the loop has completed, output the number of spaces found in a message box.



# CODE REUSE AND DATA STRUCTURES

**I**n this chapter, I show you how to increase your programming productivity by building your own procedures for reuse throughout an Access program. I also show you how to build collections of related information using data structures such as arrays and user-defined types.

## CODE REUSE

Remember that Visual Basic and VBA are *event-driven* programming languages. This means VBA programmers could easily duplicate work when writing code in two or more event procedures. For example, consider a bookstore program that contains three separate graphical interfaces (forms or windows) a user could search for a book by entering a book title and clicking a command button. As a VBA programmer, you could easily write the same code in three separate control events. This approach is demonstrated in the next three event procedures.

```
Private Sub cmdSearchFromMainWindow_Click(BookTitle As String)
```

```
    ' Common code to search for a book based on book title.
```

End Sub

---

```
Private Sub cmdSearchFromHelpWindow_Click(BookTitle As String)
```

```
    ' Common code to search for a book based on book title.
```

End Sub

---

```
Private Sub cmdSearchFromBookWindow_Click(BookTitle As String)
```

```
    ' Common code to search for a book based on book title.
```

End Sub

The program statements required to search for a book could be many lines long and needlessly duplicated in each event procedure. To solve this problem, you could build your own user-defined procedure called `SearchForBook`, which implements all the required code only once to search for a book. Then each event procedure need only call `SearchForBook` and pass in a book title as a parameter.

To remove duplicate code, I must first build the `SearchForBook` user-defined procedure.

```
Public Sub SearchForBook(sBookTitle As String)
```

```
    ' Search for a book based on book title.
```

End Sub

Instead of duplicating the `Search` statements in each `Click` event, I simply call the `SearchForBook` subprocedure, passing it a book title.

```
Private Sub cmdSearchFromMainWindow_Click()
```

```
    SearchForBook(txtBookTitle.Value)
```

End Sub

---

```
Private Sub cmdSearchFromHelpWindow_Click()
```

```
SearchForBook(txtBookTitle.Value)
```

```
End Sub
```

---

```
Private Sub cmdSearchFromBookWindow_Click()
```

```
    SearchForBook(txtBookTitle.Value)
```

```
End Sub
```

This new approach eliminates duplicate code and logic by creating what's known as code-reuse. Specifically, *code reuse* is the process by which programmers pull out commonly used statements and put them into unique procedures or functions, which can be referenced from anywhere in the application.

Code reuse makes your life as a programmer much easier and more enjoyable. It is an easy concept to grasp and is really more applied than theoretical. In the world of VBA, code reusability is implemented as subprocedures and function procedures. Programmers create user-defined procedures for problems that need frequently used solutions.

## Introduction to User-Defined Procedures

In previous chapters, you learned how to use built-in VBA functions (also known as *procedures*) such as `MsgBox` and `InputBox`. You may have wondered how those functions were implemented. In this section, you learn how to build your own functions using user-defined procedures.

Access VBA supports three types of procedures: subprocedures, function procedures, and property procedures. I specifically discuss subprocedures and function procedures in this chapter and save property procedures for Chapter 11, “Object-Oriented Programming with Access VBA,” when I discuss object-oriented programming, also known as OOP!

**The main difference between subprocedures and function procedures is that subprocedures do not return values. Many other programming languages, such as C or Java, simply refer to a procedure that returns no value as a *void function*.**

Though different in implementation and use, both subprocedures and function procedures share some characteristics, such as beginning and ending statements, executable statements, and incoming arguments. The main difference between the two revolves around a return value. Specifically, subprocedures do not return a value, whereas function procedures do.

User-defined procedures are added to your Visual Basic code modules manually or with a little help from the Add Procedure dialog box. To access the Add Procedure dialog box, open the VBE and make sure the Code window portion has the focus. Select Insert, Procedure from the menu.



The Procedure menu item appears unavailable (disabled) if the Code window in the VBE does not have the focus.

The Add Procedure dialog box in Figure 7.1 allows you to name your procedure and select a procedure type and scope.

**FIGURE 7.1**

Adding a procedure with the Add Procedure dialog box.



If you select the All Local Variables as Statics check box, your procedure-level variables maintain their values through your program's execution.

After creating your procedure, the Add Procedure dialog box tells VBA to create a procedure shell with Public Sub and End Sub statements, as shown in Figure 7.2.

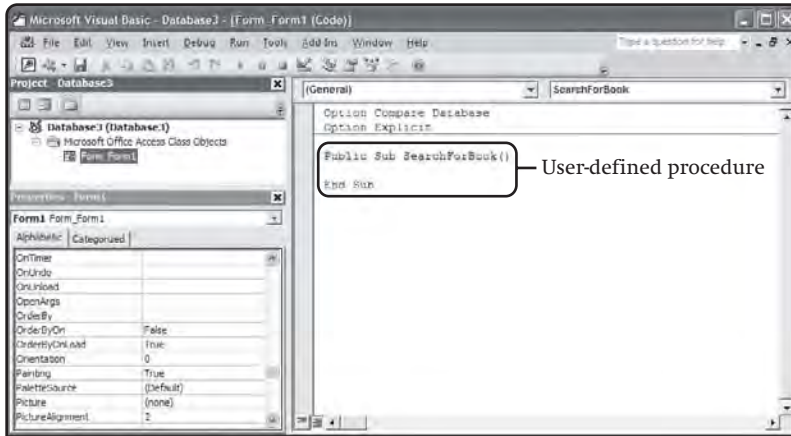
## Subprocedures

Subprocedures must have a Sub statement and corresponding End Sub statement. They can contain executable Visual Basic statements such as declaration and assignment statements. Subprocedures can take arguments such as variables, constants, and expressions. If no arguments are provided, the beginning Sub statement must contain an empty set of parentheses:

```
Public Sub DisplayCurrentTime()

    MsgBox "The time is " & Time

End Sub
```



**FIGURE 7.2**  
An empty procedure created with the Add Procedure dialog box.

The next procedure implements adding two numbers, which are passed in as arguments.

```
Public Sub AddTwoNumbers(iNumber1 As Integer, iNumber2 As Integer)

    MsgBox "The result of " & iNumber1 & " and " & iNumber2 & _
        " is " & iNumber1 + iNumber2

End Sub
```

When executed by itself, the `AddTwoNumbers` procedure requires no parentheses to surround its parameter list:

```
AddTwoNumbers 4, 6
```

When used in an assignment statement, however, the comma-separated parameter list must be enclosed in parentheses:

```
lblOutput.Caption = AddTwoNumbers(4, 6)
```

Note again that subprocedures only execute statements and do not return a value to the calling procedure. If a return value is required, consider using a function procedure (discussed next).

## Function Procedures

Function procedures are very much like subprocedures in that they consist of Visual Basic statements and take arguments. Unlike subprocedures, function procedures begin with a `Function` statement and end with an `End Function` statement. Function procedures return values to the calling procedure by assigning a value to the function name:

```
Public Function MultiplyTwoNumbers(dNumber1 As Double, dNumber2 As Double)
```

```
    MultiplyTwoNumbers = dNumber1 * dNumber2
```

```
End Function
```

The `MultiplyTwoNumbers` function procedure takes two arguments and assigns the result of their multiplication to the function name, thereby returning the result to the calling function.

```
lblResult.Caption = MultiplyTwoNumbers(6, 9)
```

To be more dynamic, I could pass `Value` properties of two text boxes directly in as arguments.

```
lblResult.Caption = MultiplyTwoNumbers(Val(txtNumber1.Value), _  
Val(txtNumber2.Value))
```

To ensure that the `MultiplyTwoNumbers` function receives numbers (doubles) as arguments, I use the `Val` function inside the parameter list to convert strings to numbers.

## Arguments and Parameters

The words *arguments* and *parameters* are often used in the same context. They differ in purpose and definition. *Arguments* are constants, variables, or expressions that are passed to a procedure, whereas *parameters* are variables that hold the arguments and can be used in the procedure that it was passed to. Confusing, I know, but know that there is technically a difference between the two words.

Many programming languages, including VBA, allow arguments to be passed either by value or by reference. When arguments are passed by value, VBA makes a copy of the original variable's contents and passes the copy to the procedure. This means the procedure can't modify the original contents of the argument, only the copy.

To pass arguments by value, you need to preface the parameter name with the `ByVal` keyword as shown in the `Increment` procedure.

```
Private Sub cmdProcess_Click()  
  
    Dim iNumber As Integer  
  
    iNumber = 1  
  
    Increment iNumber  
  
    MsgBox "The value of iNumber is " & iNumber  
  
End Sub
```

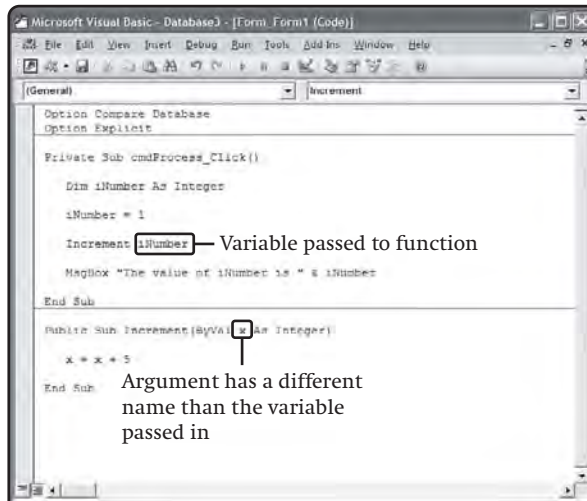
---

```
Public Sub Increment(ByVal x As Integer)
```

```
    x = x + 5
```

```
End Sub
```

Looking at Figure 7.3, you can see that it is not required to give the argument the same name as the variable passed in.



**FIGURE 7.3**

Argument names need not be the same as the variable passed in.



**CAUTION**

When not used in assignment statements, argument lists can't be enclosed in parentheses. Here is an example:

```
Increment iNumber
```

Keep in mind that Visual Basic does not always produce a runtime error when parentheses are used and yet not required. Instead, Visual Basic may simply pass the argument incorrectly, producing unexpected results.

Arguments passed by reference send the procedure a reference to the argument's memory location. In a nutshell, a memory address is sent to the procedure when arguments are passed by reference. This means the procedure is able to modify the original data. Passing arguments by reference is the default argument behavior in VBA. Moreover, passing arguments by reference is the most efficient means of passing arguments to procedures because only a reference (memory address) to the argument is passed, not the data itself.

To pass arguments by reference, simply preface the argument name using the **ByRef** keyword or use no preface keyword at all.

```
Private Sub cmdProcess_Click()
```

```
    Dim iNumber As Integer
```

```
    iNumber = 1
```

```
    Increment iNumber
```

```
    MsgBox "The value of iNumber is " & iNumber
```

```
End Sub
```

---

```
Public Sub Increment(ByRef x As Integer)
```

```
    x = x + 5
```

```
End Sub
```



Arguments are passed by reference automatically. It is not necessary to preface the argument name with the **ByRef** keyword.

Passing the `iNumber` variable by reference allows the `Increment` procedure to modify the argument's value directly.

## STANDARD MODULES

Access VBA supports two types of modules: class and standard. Class modules are directly related to an object, such as a form or report. Form class modules contain event procedures for the associated controls and objects. Standard modules, however, have no association with an object. They store a collection of variables and user-defined procedures, which can be shared among your Access programs.

You can add a standard module from the Visual Basic environment by selecting `Insert, Module` from the menu.

To see how you could utilize a standard module, I've revised the `Secret Message` program from Chapter 6, "Common Formatting and Conversion Functions." Specifically, I added one standard module and two public functions called `Encrypt` and `Decrypt`. Using public functions allows me to reuse the code in these functions from anywhere in my application.

To move the `Encrypt` and `Decrypt` functionality from event procedures to functions, I first create the function shells using the `Add Procedure` dialog box. Next, I add a string parameter to both functions. This argument is passed into each function when called. Moreover, the parameter called `sMessage` replaces the hard-coded text-box value from the previous version of `Secret Message`. All occurrences of the text-box name are replaced with the parameter name. This is truly code reuse, as I can now call these functions and pass my message from anywhere in my Access application. Last but not least, I assign the function's output to the function's name.

The new standard module code from the enhanced `Secret Message` program is shown next.

`Option Compare Database`  
`Option Explicit`

---

```
Public Function Decrypt(sMessage As String)
```

```
    Dim sDecryptedMessage As String  
    Dim sDecryptedCharacter As String  
    Dim iCounter As Integer
```

```
    For iCounter = 1 To Len(sMessage)
```

```
sDecryptedCharacter = Asc(Mid(sMessage, iCounter, 1))  
sDecryptedCharacter = Chr(sDecryptedCharacter - 1)  
sDecryptedMessage = sDecryptedMessage + sDecryptedCharacter
```

```
Next iCounter
```

```
' Assign decrypted message to function name.  
Decrypt = sDecryptedMessage
```

```
End Function
```

---

```
Public Function Encrypt(sMessage As String)
```

```
Dim sEncryptedMessage As String  
Dim sEncryptedCharacter As String  
Dim iCounter As Integer
```

```
For iCounter = 1 To Len(sMessage)
```

```
    sEncryptedCharacter = Asc(Mid(sMessage, iCounter, 1))  
    sEncryptedCharacter = Chr(sEncryptedCharacter + 1)  
    sEncryptedMessage = sEncryptedMessage + sEncryptedCharacter
```

```
Next iCounter
```

```
' Assign encrypted message to function name.  
Encrypt = sEncryptedMessage
```

```
End Function
```

With my `Encrypt` and `Decrypt` functions implemented in a standard module, I simply need to call them and pass the `Value` property from the text box. After the function call is executed, the function's return value is assigned back to the text box's `Value` property.

```
Option Compare Database  
Option Explicit
```

---

```
Private Sub cmdDecrypt_Click()

    ' Call the Decrypt function passing the encrypted
    ' message as an argument. Assign function's result
    ' to the text box's Value property.
    If txtMessage.Value <> "" Then
        txtMessage.Value = Decrypt(txtMessage.Value)
    End If

End Sub
```

---

```
Private Sub cmdEncrypt_Click()

    ' Call the Encrypt function passing the plain text
    ' message as an argument. Assign function's result
    ' to the text box's Value property.
    If txtMessage.Value <> "" Then
        txtMessage.Value = Encrypt(txtMessage.Value)
    End If

End Sub
```

You should understand that the changes made to the **Secret Message** program are transparent to the user. In other words, the use of user-defined functions and standard modules does not change the way the user interacts with the program, nor does it change the program's functionality. The important concept is that the changes were made to provide a more modular program, which implements code reuse through user-defined procedures and modules. The enhanced **Secret Message** program can be found on the companion website.

## ARRAYS

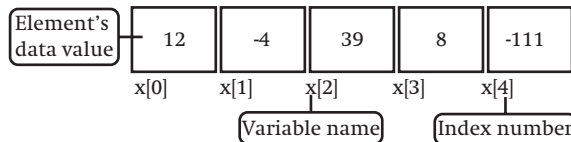
Arrays are one of the first data structures learned by beginning programmers. Not only common as a teaching tool, arrays are frequently used by professional programmers to store like data types as one variable. In short, *arrays* can be thought of as a single variable that contains many elements. Moreover, VBA arrays share many common characteristics:

- Elements in an array share the same variable name.
- Elements in an array share the same data type.
- Elements in an array are accessed with an index number.

As noted, elements in an array share the same variable name and data type. Individual members in an array are called *elements* and are accessed via an index. Just like any other variable, arrays occupy memory space. To explain further, an array is a grouping of contiguous memory segments, as demonstrated in Figure 7.4.

**FIGURE 7.4**

A five-element array.



Notice the five-element array in Figure 7.4 starts with **index 0**. This is an important concept to remember, so it's worth repeating in italics: *Unless otherwise stated, elements in an array begin with index number zero*. With that said, there are five array elements in Figure 7.4, starting with index 0 and ending with index 4.



A common programming error is not accounting for the zero-based index in arrays. This programming error is often called the *off-by-one error*. Errors like this are generally not caught during compile time, but rather at runtime when a user or your program attempts to access an element number in an array that does not exist. For example, if you have a five-element array and your program tries to access the fifth element with index number 5, a runtime program error ensues. This is because the last index in a five-element array is **index 4**!

## Single-Dimension Arrays

Using the keywords `Dim`, `Static`, `Public`, and `Private`, arrays are created just like any other variable.



Unless `Option Base 1` is specified, or *dimensioned*, with an explicit range, arrays by default begin with a zero base index.

- `Dim myIntegerArray(5) As Integer` ' creates six Integer elements.
- `Dim myVariantArray(10)` ' creates eleven Variant elements.
- `Dim myStringArray(1 to 7) As String` ' creates 7 String elements.

In the preceding declarations, the number of elements in an array is determined during array declaration using either a number or a range of numbers surrounded by parentheses.

A nice feature of VBA is its ability to initialize variables for use. Specifically, VBA initializes number-based array elements to 0 and string-based array elements to "" (indicating an empty string).

Individual elements in an array are accessed via an index:

```
lblArrayValue.Caption = myStringArray(3)
```

The next Click event procedure initializes a String array using a For loop and adds the array contents to a list box.

```
Private Sub cmdPopulateListBox_Click()  
  
    ' Declare a seven element String array.  
    Dim myStringArray(1 To 7) As String  
    Dim x As Integer  
  
    ' Initialize array elements.  
    For x = 1 To 7  
  
        myStringArray(x) = "The value of myStringArray is " & x  
  
    Next x  
  
    ' Add array contents to a list box.  
    For x = 1 To 7  
  
        lstMyListBox.AddItem myStringArray(x)  
  
    Next x  
  
End Sub
```

VBA provides two array-based functions called `LBound` and `UBound` for determining an array's lower and upper bounds. The `LBound` function takes an array name and returns the array's lower bound. Conversely, the `UBound` function takes an array name and returns the array's upper bound. These functions are demonstrated in this Click event procedure.

```

Private Sub cmdPopulateListBox_Click()

    ' Declare an eleven element Integer array.
    Dim myIntegerArray(10) As Integer
    Dim x As Integer

    ' Initialize array elements using LBound and UBound functions
    ' to determine lower and upper bounds.
    For x = LBound(myIntegerArray) To UBound(myIntegerArray)

        myIntegerArray(x) = x

    Next x

    ' Add array contents to a list box.
    For x = LBound(myIntegerArray) To UBound(myIntegerArray)

        lstMyListBox.AddItem myIntegerArray(x)

    Next x

End Sub

```

## Two-Dimensional Arrays

Two-dimensional arrays are most often thought of in terms of a table or matrix. For example, a two-dimensional array containing four rows and five columns creates 20 elements, as shown in Figure 7.5.

```
Dim x(3, 4) As Integer ' Two dimensional array with 20 elements.
```

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	x(0,0)	x(0,1)	x(0,2)	x(0,3)	x(0,4)
Row 1	x(1,0)	x(1,1)	x(1,2)	x(1,3)	x(1,4)
Row 2	x(2,0)	x(2,1)	x(2,2)	x(2,3)	x(2,4)
Row 3	x(3,0)	x(3,1)	x(3,2)	x(3,3)	x(3,4)

Row Index
Column Index

**FIGURE 7.5**

A two-dimensional array with 20 elements.

The first index (also known as a *subscript*) in a two-dimensional array represents the row in a table. The second index represents the table's column. Together, both subscripts specify a single element within an array.

A nested looping structure is required to iterate through all elements in a two-dimensional array.

```
Private Sub cmdInitializeArray_Click()  
  
    ' Create a 20 element two dimensional array.  
    Dim x(3, 4) As Integer  
    Dim iRow As Integer  
    Dim iColumn As Integer  
  
    ' Loop through one row at a time.  
    For iRow = 0 To 3  
  
        ' Loop through each column in the row.  
        For iColumn = 0 To 4  
  
            ' Populate each element with the result of  
            ' multiplying the row and column.  
            x(iRow, iColumn) = iRow * iColumn  
  
        Next iColumn  
  
    Next iRow  
  
End Sub
```

As shown in the previous Click event, the outer For loop iterates through one column at a time. Each time the outer loop is executed, a nested For loop is executed five times. The inner loop represents each column (in this case five columns) in a row. After each column in a row has been referenced, the outer loop executes again, which moves the array position to the next row and the inner loop to the next set of columns.

## Dynamic Arrays

Arrays are useful when you know how many elements you need. What if you don't know how many array elements your program requires? One way to circumvent this problem is by creating a huge array that most definitely holds any number of elements you throw at it. I don't



recommend this, however. When arrays are *declared* (created), VBA reserves enough memory to hold data for each element. If you're guessing on the number of elements required, you're most certainly wasting memory! A more professional way of solving this dilemma is with dynamic arrays.

If you've worked in other programming languages, such as C, you might be cringing about the thought of dynamic arrays implemented with linked lists. You will be relieved to learn that VBA makes building and working with dynamic arrays very easy.

When your program logic uses dynamic arrays, it can size and resize your array while the application is running. To create a dynamic array, simply eliminate any references to subscripts or indexes in the array declaration.

Option Compare Database

Option Explicit

```
Dim iDynamicArray() As Integer ' Dynamic array.
```

Leaving the parentheses empty tells VBA that your array is dynamic. I will be able to use my dynamic array in all subsequent form-level procedures by dimensioning the dynamic array in the general declaration area. To set the number of elements in a dynamic array, use the **ReDim** keyword.

```
Private Sub cmdDynamicArray_Click()
```

```
    Dim sUserResponse As String
```

```
    sUserResponse = InputBox("Enter number of elements:")
```

```
    ' Set number of array elements dynamically.
```

```
    ReDim iDynamicArray(sUserResponse)
```

```
    MsgBox "Number of elements in iDynamicArray is " _  
        & UBound(iDynamicArray) + 1
```

```
End Sub
```

Using the **ReDim** keyword, I can set my array size after the program is running. The only problem with this approach is that each time the **ReDim** statement is executed, all previous element data is lost. To correct this, use the **Preserve** keyword in the **ReDim** statement, as follows.

```

Private Sub cmdIncreaseDynamicArray_Click()

    Dim sUserResponse As String

    sUserResponse = InputBox("Increase number of elements by:")

    ' Set number of array elements dynamically, while
    ' preserving existing elements.
    ReDim Preserve iDynamicArray(UBound(iDynamicArray) _
        + sUserResponse)

    MsgBox "Number of elements in iDynamicArray is now " _
        & UBound(iDynamicArray) + 1

End Sub

```

To preserve current elements while increasing a dynamic array, you must tell VBA to add elements to the array's existing upper bound. This can be accomplished using the `UBound` function, as demonstrated in the previous `Click` event procedure `cmdIncreaseDynamicArray`.



The `Preserve` keyword allows you to change a dynamic array's upper bound only. You cannot change a dynamic array's lower bound with the `Preserve` keyword.

## Passing Arrays as Arguments

Passing an array to a function or subprocedure is not as difficult in VBA as one might think. You must follow a couple of rules, however, to ensure a valid argument pass.

To pass all elements in an array to a procedure, simply pass the array name with no parentheses. Next, you must define the parameter name with an empty set of parentheses, as the next two procedures demonstrate.

```

Private Sub cmdPassEntireArray_Click()

    Dim myArray(5) As Integer

    HowMany myArray

```

End Sub

---

```
Private Sub HowMany(x() As Integer)
```

```
    MsgBox "There are " & UBound(x) & " elements in this array."
```

```
End Sub
```

To pass a single element in an array, it is not necessary to define the parameter name as an array. Rather, simply pass one array element as a normal variable argument:

```
Private Sub cmdPassArrayElement_Click()
```

```
    Dim myArray(5) As Integer
```

```
    CheckItOut myArray(3)
```

```
End Sub
```

---

```
Private Sub CheckItOut(x As Integer)
```

```
    MsgBox "The parameter's value is " & x & "."
```

```
End Sub
```

Passing arrays and elements of arrays as arguments is that easy!

## USER-DEFINED TYPES

In other programming languages, such as C, user-defined types are commonly referred to as *structures*. *User-defined types* are collections of one or more related elements, which can be of different data types. User-defined types must be declared at the module level (also known as the *general declarations area*) in a standard module. Programmers can leverage user-defined types to group like variables as one, much as a record in a database does.

## Type and End Type Statements

User-defined types are created with the `Type` and `End Type` statements at the module level. More specifically, user-defined types must be declared outside of any procedure in a standard module. To demonstrate, I created a user-defined type called `EmployeeData`.

Type EmployeeData

```
EmployeeLastName As String
EmployeeFirstName As String
EmployeeID As Integer
EmployeeSalary As Currency
EmployeeHireDate As Date
```

End Type

' ...is the same as

Public Type EmployeeData

```
Dim EmployeeLastName As String
Dim EmployeeFirstName As String
Dim EmployeeID As Integer
Dim EmployeeSalary As Currency
Dim EmployeeHireDate As Date
```

End Type



It is not necessary to use the `Dim` keyword when declaring variables (members) inside a user-defined type.

Note that declaring a user-defined type does not instantiate a variable, nor does it reserve any space in memory. The declaration of a user-defined type simply provides VBA with a blueprint when variables of your user-defined type are created.

By default, user-defined types are public, though they can be declared using the keyword `Private`, which makes them available only to the current module from where they are created.

' Available only in the current module.

Private Type BookData

```
Title As String
ISBN As String
Author As String
Publisher As String
```

```
PublishDate As Date
Price As Currency
```

```
End Type
```

## Declaring Variables of User-Defined Type

As mentioned, declaring a user-defined type does not create a variable, but rather defines a template for VBA programmers to use later. To create variables of your user-defined types, define a user-defined type in a standard module.

```
Option Compare Database
Option Explicit
```

```
' Define user defined type in a standard module.
Type BookData
```

```
Title As String
ISBN As String
Author As String
Publisher As String
PublishDate As Date
Price As Currency
```

```
End Type
```

Then you can create variables of your user-defined type at a module level.

```
' Declare 5 element array of BookData Type
Dim myFavoriteBook As BookData
```

Because user-defined types are public by default, you can create type variables in other modules, such as form class modules:

```
Private Sub cmdEnterBookData_Click()

' Declare one variable of BookData Type
Dim myCookingBook As BookData

End Sub
```

## Managing Elements

Once a variable has been declared as a user-defined type, you can use it much like any other variable. To access elements within type variables, simply use the dot notation to assign and retrieve data, as the next program demonstrates.

```
Private Sub cmdEnterBookData_Click()  
  
    Dim myBook As BookData ' Declare one variable of BookData Type  
  
    myBook.Title = txtTitle.Value  
    myBook.ISBN = txtISBN.Value  
    myBook.Author = txtAuthor.Value  
    myBook.Publisher = txtPublisher.Value  
    myBook.PublishDate = txtPublishDate.Value  
    myBook.Price = txtPrice.Value  
  
    MsgBox myBook.Title & " has been entered."  
  
End Sub
```

Note that a public user-defined type must have already been created in a standard module.

Remember that user-defined types can be thought of as rows in a database table: Both table rows and user-defined types maintain a grouping of like elements of one or more data types.

So far, you have only seen how to create a single variable of user-defined type (analogous to a single row in a database). To create multiple variables of the same user-defined type (much like multiple rows in a database), simply create an array of user-defined type, as shown in the next program.

```
Option Compare Database  
Option Explicit  
  
Dim myBooks() As BookData ' Declare dynamic array of BookData Type  
Dim currentIndex As Integer
```

---

```
Private Sub cmdAddNewBook_Click()  
  
    ' Add one array element to the dynamic array.
```

```
ReDim Preserve myBooks(UBound(myBooks) + 1)
```

```
' Clear text boxes
txtTitle.Value = ""
txtISBN.Value = ""
txtAuthor.Value = ""
txtPublisher.Value = ""
txtPublishDate.Value = ""
txtPrice.Value = ""
```

```
End Sub
```

---

```
Private Sub cmdEnterBookData_Click()
```

```
myBooks(UBound(myBooks)).Title = txtTitle.Value
myBooks(UBound(myBooks)).ISBN = txtISBN.Value
myBooks(UBound(myBooks)).Author = txtAuthor.Value
myBooks(UBound(myBooks)).Publisher = txtPublisher.Value
myBooks(UBound(myBooks)).PublishDate = txtPublishDate.Value
myBooks(UBound(myBooks)).Price = txtPrice.Value
```

```
MsgBox myBooks(UBound(myBooks)).Title & " has been entered."
```

```
End Sub
```

---

```
Private Sub cmdNext_Click()
```

```
If currentIndex <= UBound(myBooks) Then
```

```
    If currentIndex < UBound(myBooks) Then
        ' Increment index.
        currentIndex = currentIndex + 1
    End If
```

```
txtTitle.Value = myBooks(currentIndex).Title
txtAuthor.Value = myBooks(currentIndex).Author
txtISBN.Value = myBooks(currentIndex).ISBN
```

```
txtPublisher.Value = myBooks(currentIndex).Publisher
txtPublishDate.Value = myBooks(currentIndex).PublishDate
txtPrice.Value = myBooks(currentIndex).Price
```

```
End If
```

```
End Sub
```

---

```
Private Sub cmdPrevious_Click()
```

```
If currentIndex >= 1 Then
```

```
    If currentIndex > 1 Then
        ' Decrement index.
        currentIndex = currentIndex - 1
    End If
```

```
    txtTitle.Value = myBooks(currentIndex).Title
    txtAuthor.Value = myBooks(currentIndex).Author
    txtISBN.Value = myBooks(currentIndex).ISBN
    txtPublisher.Value = myBooks(currentIndex).Publisher
    txtPublishDate.Value = myBooks(currentIndex).PublishDate
    txtPrice.Value = myBooks(currentIndex).Price
```

```
End If
```

```
End Sub
```

---

```
Private Sub Form_Load()
```

```
    ' Add one array element to the dynamic array.
    ReDim myBooks(1)
```

```
    currentIndex = 1
```

```
End Sub
```



The **Integer** variable—called `currentIndex` in the previous `Form_Load` event procedure—was declared in the general declarations section. It therefore can be used throughout the form's class module. I use this variable to maintain the current index of the array as I navigate through the elements in the array. Also note the use of dynamic array techniques to add elements of `BookData` type to my array variable in the `cmdAddNewBook_Click` event procedure.

## CHAPTER PROGRAM: DICE

The chapter program `Dice` in Figure 7.6 is an easy-to-build, fun game. Mimicking basic poker rules, the player rolls the dice by clicking a command button and hopes for either three of a kind (worth 10 points) or, better yet, four of a kind (worth 25 points).

**FIGURE 7.6**

Using chapter-based concepts to build the `Dice` program.



The `Dice` program implements code reuse by leveraging chapter-based concepts such as arrays, user-defined procedures, and standard code modules. In addition to chapter-based concepts, the `Dice` program uses random number techniques to simulate a roll. (This is discussed in Chapter 5, “Looping Structures.”)

Controls and properties that build the `Dice` program are described in Table 7.1.

**TABLE 7.1 CONTROLS AND PROPERTIES OF THE DICE PROGRAM**

Control	Property	Property Value
Form	Name	Chapter Program
	Caption	Chapter Program
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No

Control	Property	Property Value
Frame	Name	fraGameBoard
Command Button	Name	cmdHowToPlay
	Caption	How to Play
Command Button	Name	cmdQuit
	Caption	End Game
Command Button	Name	cmdRoll
	Caption	Roll the Dice!
Label	Name	lblTitle
	Caption	Chapter 7 - Dice!
Image	Name	imgSlot1
	Picture	blank.jpg
	Size Mode	Stretch
Image	Name	imgSlot2
	Picture	blank.jpg
	Size Mode	Stretch
Image	Name	imgSlot3
	Picture	blank.jpg
	Size Mode	Stretch
Image	Name	imgSlot4
	Picture	blank.jpg
	Size Mode	Stretch
Image	Name	imgDice1
	Picture	die1.jpg
	Size Mode	Stretch
Image	Name	imgDice2
	Picture	die2.jpg
	Size Mode	Stretch
Image	Name	imgDice3
	Picture	die3.jpg
	Size Mode	Stretch
Image	Name	imgDice4
	Picture	die4.jpg
	Size Mode	Stretch
Image	Name	imgDice5
	Picture	die5.jpg
	Size Mode	Stretch
Image	Name	imgDice6
	Picture	die6.jpg
	Size Mode	Stretch

All of the code required to implement the form class module in the Dice program is seen next.

Option Compare Database

Option Explicit

---

```
Private Sub cmdQuit_Click()
```

```
    DoCmd.Quit
```

```
End Sub
```

---

```
Private Sub cmdRoll_Click()
```

```
    ' Roll the Dice!
```

```
    RollTheDice
```

```
    ' Check the player's hand.
```

```
    DetermineCurrentHand iCurrentHand
```

```
End Sub
```

---

```
Private Sub cmdHowToPlay_Click()
```

```
    MsgBox "Dice! Version 1.0" & Chr(13) & _
```

```
    "Developed by Michael Vine." & Chr(13) & Chr(13) & _
```

```
    "Roll the dice and win points with four of a kind (25 points), " & _
```

```
    & Chr(13) & _
```

```
    "and three of a kind (10 points).", , "Chapter 7 - Dice!"
```

```
End Sub
```

---

```
Private Sub Form_Load()
```

```
    Randomize
```

```
    lblScore.Caption = "Your score is " & iScore
```

```
End Sub
```

---

The algorithm to roll the dice and check the player's hand is implemented in a standard code module, which is shown next.

Option Compare Database

Option Explicit

' Public variables available to all procedures in

' all modules

Public iScore As Integer

Public iCurrentHand(3) As Integer

---

Public Sub DetermineCurrentHand(a() As Integer)

' Look for valid hands worth points.

' Valid hands with points are:

' 3 of a kind - 10 points

' 4 of a kind - 25 points

Dim iCounter As Integer

' Holds possibilities of a win

Dim iNumbers(1 To 6) As Integer

' Count the number of occurrences for each die

For iCounter = 0 To 3

    Select Case a(iCounter)

        Case 1

            iNumbers(1) = iNumbers(1) + 1

        Case 2

            iNumbers(2) = iNumbers(2) + 1

        Case 3

            iNumbers(3) = iNumbers(3) + 1

        Case 4

            iNumbers(4) = iNumbers(4) + 1

        Case 5

            iNumbers(5) = iNumbers(5) + 1

        Case 6

            iNumbers(6) = iNumbers(6) + 1

    End Select

Next iCounter

```

' Determine if player has four of a kind
If iNumbers(1) = 4 Or iNumbers(2) = 4 Or iNumbers(3) = 4 Or _
    iNumbers(4) = 4 Or iNumbers(5) = 4 Or iNumbers(6) = 4 Then

    MsgBox "Four of a kind! 25 points!"
    iScore = iScore + 25
    Forms("ChapterProgram").lblScore.Caption = _
        "Your score is " & iScore
    Exit Sub

End If

```

```

' Player did not have a four of a kind, see if they
' have three of a kind.
If (iNumbers(1) = 3 Or iNumbers(2) = 3 Or iNumbers(3) = 3 Or _
    iNumbers(4) = 3 Or iNumbers(5) = 3 Or iNumbers(6) = 3) Then

    MsgBox "Three of a kind! 10 points!"
    iScore = iScore + 10
    Forms("ChapterProgram").lblScore.Caption = _
        "Your score is " & iScore
    Exit Sub

End If

```

```
End Sub
```

---

```
Public Sub RollTheDice()
```

```

    Dim iCounter As Integer

    ' Reset current hand
    For iCounter = 0 To 3
        iCurrentHand(iCounter) = Int((6 * Rnd) + 1)
    Next iCounter

    ' Assign a die to the first slot
    Select Case iCurrentHand(0)
        Case 1
            Forms("ChapterProgram").imgSlot1.Picture =
Forms("ChapterProgram").imgDice1.Picture

```

```

Case 2
    Forms("ChapterProgram").imgSlot1.Picture =
Forms("ChapterProgram").imgDice2.Picture
Case 3
    Forms("ChapterProgram").imgSlot1.Picture =
Forms("ChapterProgram").imgDice3.Picture
Case 4
    Forms("ChapterProgram").imgSlot1.Picture =
Forms("ChapterProgram").imgDice4.Picture
Case 5
    Forms("ChapterProgram").imgSlot1.Picture =
Forms("ChapterProgram").imgDice5.Picture
Case 6
    Forms("ChapterProgram").imgSlot1.Picture =
Forms("ChapterProgram").imgDice6.Picture
End Select

```

```

' Assign a die to the second slot
Select Case iCurrentHand(1)
Case 1
    Forms("ChapterProgram").imgSlot2.Picture =
Forms("ChapterProgram").imgDice1.Picture
Case 2
    Forms("ChapterProgram").imgSlot2.Picture =
Forms("ChapterProgram").imgDice2.Picture
Case 3
    Forms("ChapterProgram").imgSlot2.Picture =
Forms("ChapterProgram").imgDice3.Picture
Case 4
    Forms("ChapterProgram").imgSlot2.Picture =
Forms("ChapterProgram").imgDice4.Picture
Case 5
    Forms("ChapterProgram").imgSlot2.Picture =
Forms("ChapterProgram").imgDice5.Picture
Case 6
    Forms("ChapterProgram").imgSlot2.Picture =
Forms("ChapterProgram").imgDice6.Picture
End Select

```

```

' Assign a die to the third slot

```

```
Select Case iCurrentHand(2)
    Case 1
        Forms("ChapterProgram").imgSlot3.Picture =
Forms("ChapterProgram").imgDice1.Picture
    Case 2
        Forms("ChapterProgram").imgSlot3.Picture =
Forms("ChapterProgram").imgDice2.Picture
    Case 3
        Forms("ChapterProgram").imgSlot3.Picture =
Forms("ChapterProgram").imgDice3.Picture
    Case 4
        Forms("ChapterProgram").imgSlot3.Picture =
Forms("ChapterProgram").imgDice4.Picture
    Case 5
        Forms("ChapterProgram").imgSlot3.Picture =
Forms("ChapterProgram").imgDice5.Picture
    Case 6
        Forms("ChapterProgram").imgSlot3.Picture =
Forms("ChapterProgram").imgDice6.Picture
End Select

' Assign a die to the fourth slot
Select Case iCurrentHand(3)
    Case 1
        Forms("ChapterProgram").imgSlot4.Picture =
Forms("ChapterProgram").imgDice1.Picture
    Case 2
        Forms("ChapterProgram").imgSlot4.Picture =
Forms("ChapterProgram").imgDice2.Picture
    Case 3
        Forms("ChapterProgram").imgSlot4.Picture =
Forms("ChapterProgram").imgDice3.Picture
    Case 4
        Forms("ChapterProgram").imgSlot4.Picture =
Forms("ChapterProgram").imgDice4.Picture
    Case 5
        Forms("ChapterProgram").imgSlot4.Picture =
Forms("ChapterProgram").imgDice5.Picture
```

## Case 6

```
Forms("ChapterProgram").imgSlot4.Picture =  
Forms("ChapterProgram").imgDice6.Picture  
End Select
```

```
End Sub
```

## SUMMARY

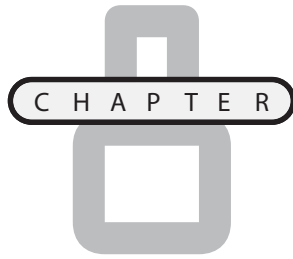
- Code reuse is implemented as user-defined subprocedures and function procedures.
- Function procedures return a value; subprocedures do not.
- Both subprocedures and function procedures can take one or more arguments.
- Arguments are the data passed into procedures. Parameters are the variables inside the procedure, which represent the argument data.
- Arguments can be passed by value and by reference.
- Arguments passed by value contain a copy of the original data. This prevents the procedure from modifying the original data.
- Arguments passed by reference contain a reference to the variable's memory address. The procedure can modify the original data.
- Arguments are passed by reference automatically.
- Standard modules are used to group commonly referenced user-defined procedures together.
- Arrays are used to store groupings of like data types as one variable.
- An array is a grouping of contiguous memory segments.
- Variables in an array are called elements.
- Each variable in an array shares the same name.
- Elements in an array are accessed via an index.
- VBA arrays are zero-based by default.
- Arrays are created just like other variables using the keywords **Dim**, **Static**, **Public**, and **Private**.
- Two-dimensional arrays are often thought of in terms of a table or matrix.
- Two looping structures (one of which is nested) are required to iterate through each element in a two-dimensional array.
- Dynamic arrays can be created and managed using the **ReDim** and **Preserve** keywords.
- Arrays can be passed as arguments to procedures.



- User-defined types are commonly referred to as structures.
- User-defined types are groupings of like information, which can be of different data types.
- User-defined types are created using the `Type` and `End Type` statements.
- User-defined types must be declared in a standard module in the general declarations area (outside of any procedure).
- Variables of user-defined type are analogous to rows in a database.

### PROGRAMMING CHALLENGES

1. Build a form with two text boxes that receive numbers as input and one command button that displays a message box containing the larger of the two numbers. To accomplish this, write code in the `Click` event of the command to call a user-defined function named `FindLargestNumber`. Pass it two arguments (text box values) and display the result in a message box. Write code in the `FindLargestNumber` function to determine the larger number and return the result to the calling procedure (command button `Click` event in this case).
2. Create a one-dimensional string-based array with five elements. Assign five different names to the array. Use a `For` loop to iterate through each of the array elements, displaying the names in a message box.
3. Declare a user-defined type called `HomeData` with elements `StreetAddress`, `City`, `State`, `SquareFootage`, `LotSize`, and `SalePrice` in a standard module. Create a form with six text boxes to add values to each variable type element. In the general declarations area of the form, create a single variable of `HomeType` to store the user-entered value. Add two command buttons to the form, one called `cmdAddHome` and the other `cmdDisplayHomeData`. In `cmdAddHome Click` event, store the data entered by the user into your user-defined type variable. In the `cmdDisplayHomeData Click` event, display each element's value in a message box.
4. Update the chapter program `D1ce` to check for two pair.



# DEBUGGING, INPUT VALIDATION, FILE PROCESSING, AND ERROR HANDLING

**T**his chapter teaches you techniques for preventing runtime errors through input validation and error handling. You also see how to debug your VBA code using common VBE debugging windows. In addition, I show you how VBA manages file input and output (*file I/O*).

## DEBUGGING

Sooner or later, all programmers seek the Holy Grail of debugging. The Holy Grail of debugging is different for each programming language. VBA programmers are very lucky the VBE provides a multitude of debugging facilities not found in many other programming environments.

As a programming instructor, I've often encouraged my Visual Basic students to use the VBE debugging facilities not only to debug programs, but also to see how the program flows, how variables are populated, and how and when statements are executed. In other, less friendly languages, programmers must take for granted the order in which their statements are executed. In VBA, you can actually step through your application one statement at a time. You can even go back in time to re-execute statements, something I show you a little later on.

In this section, I show you how to leverage each of the following VBE debugging facilities:

- Breakpoints
- Immediate window
- Locals window
- Watch window

## WHAT ARE SOFTWARE BUGS?

Debugging can be one of the most challenging processes in software development, and unfortunately it's sometimes very costly. In a nutshell, *debugging* is the process by which programmers identify, find, and correct software errors.

There are three common types of bugs in software. *Syntax errors* are the most common form of software bugs. They are caused by misspellings in the program code and are most commonly recognized by the language's compiler. Syntax errors are generally easy to fix.

The next type of bug is called a runtime error. *Runtime errors* occur once the program is running and an illegal operation occurs. These errors generally occur because the programmer has not thought ahead of time to capture them (for example, File Not Found, Disk Not Ready, or Division by Zero). Runtime errors are most often easy to find and sometimes easy to fix.

The last common type of bug, and the most difficult to identify and fix, is known as the logic error. *Logic errors* are not easily identified, because they don't necessarily generate an error message. Logic errors are the result of erroneous logic implemented in the program code. Examples of logic errors include invalid mathematical calculations, the wrong variable used in an operation, or calling (executing) the wrong procedure.

## Stepping Through Code

By now you should be fairly comfortable with the design-time and runtime environments. Moreover, you may have discovered the break mode environment. As a refresher, the next bulleted list reviews each type of Access VBA environment.

- **Design time** is the mode by which you add controls to containers (such as forms) and write code to respond to events.
- The **runtime** environment allows you to see your program running the same way a user would. During runtime you can see all your Visual Basic code, but you cannot modify it.

- **Break mode** allows you to pause execution of your Visual Basic program (during runtime) to view, edit, and debug your program code.

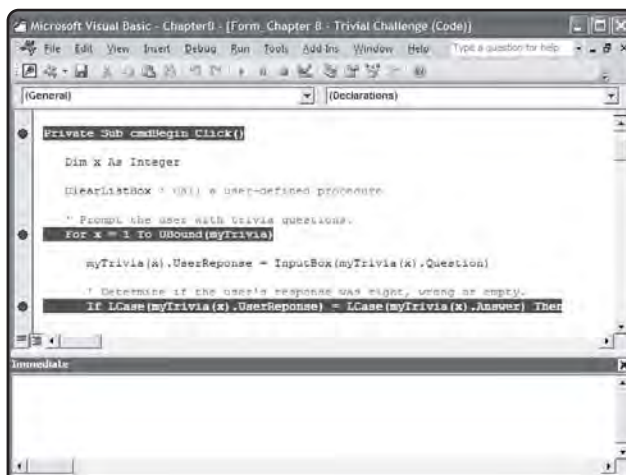
The VBE allows you to step through your program code one line at a time. Known as *stepping* or *stepping into*, this process allows you to graphically see what line of code is currently executing as well as values of current variables in scope. Using function keys or menu items, you can navigate through program code with ease. For example, once in break mode, you can press the F8 key to skip to the next line.

During break mode, it is also possible to step over a procedure without having to graphically execute the procedure's statements one at a time. Known as *procedure stepping* or *stepping over*, this process can be accomplished during break mode by pressing Shift + F8 simultaneously.

Sometimes you might want to skip ahead in program code to a predetermined procedure or statement. The VBE provides this functionality through the use of breakpoints.

## Breakpoints

*Breakpoints* can be inserted into your Visual Basic procedures during design time or break mode, as seen in Figure 8.1.



**FIGURE 8.1**

Inserting  
breakpoints on  
program  
statements.

To create a breakpoint, simply click in the left margin of the Code window where you want program execution to pause. When your program's execution reaches the statement where a breakpoint has been placed, program execution pauses. To continue execution to the next breakpoint, simply press F5. To continue program execution one statement at a time, with or without a breakpoint, press the F8 key.

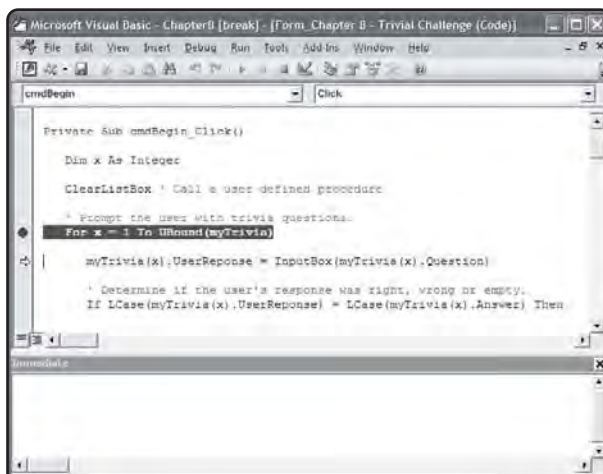


Breakpoints cannot be placed on empty lines in the Code window or on variable declarations.

There are occasions when you want to go back in time and re-execute a particular program statement without having to halt the entire program and re-run it. Believe it or not, the VBE provides a facility for traveling back in time while in break mode. To do so, simply click the yellow arrow in the left margin of the Code window (shown in Figure 8.2) and drag it to a previous program statement.

**FIGURE 8.2**

Going back in time to re-execute program statements while in break mode.

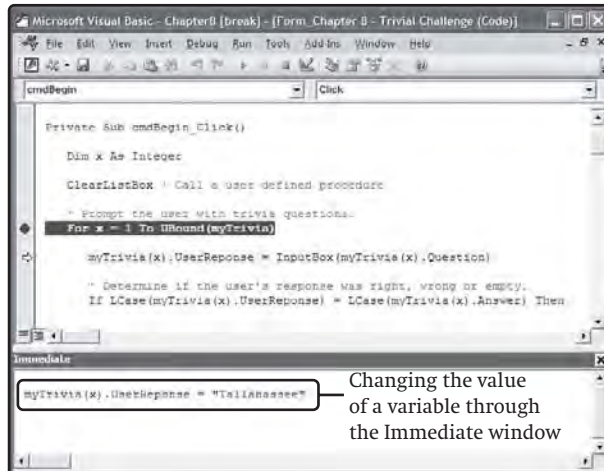


The arrow to the far left in Figure 8.2 is the current line of execution. Using your mouse, you can move the arrow to other valid lines of execution.

## Immediate Window

During testing or debugging, it is not always desirable to change the values of variables and properties by modifying program code. A safer way of testing program code is through the use of the Immediate window. The Immediate window can be used during design time or break mode. Most popular in break mode, the Immediate window can be accessed by pressing Ctrl + G or through the View menu.

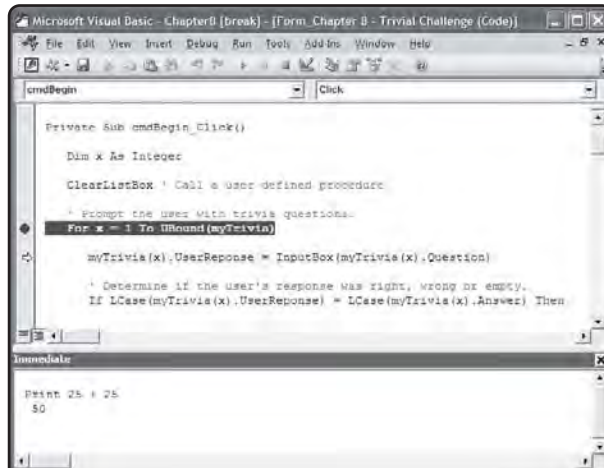
The Immediate window allows you to verify and change the values of properties or variables, as shown in Figure 8.3.

**FIGURE 8.3**

Changing a variable's value through the Immediate window.

Interestingly, you can type statements that do not directly correspond with your current program execution. For example, in Figure 8.4 I entered the following expression into the Immediate window.

Print 25 + 25

**FIGURE 8.4**

Using the Print keyword to display results in the Immediate window.

After I press the Enter key, the Immediate window produces the result of my expression (in this case, 50). The keyword `Print` tells the Immediate window to print the expression's result to the Immediate window's screen.





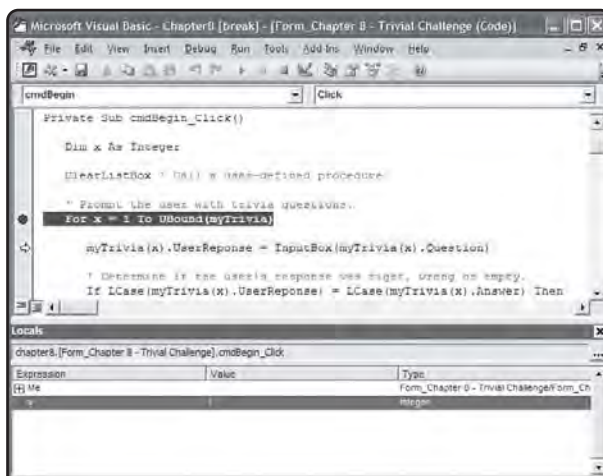
You can re-execute a statement in the Immediate window by moving the cursor to the statement's line and pressing Enter.

## Locals Window

The Locals window, a friendly companion to any VBA programmer, provides valuable information about variables and control properties in current scope. Accessed from the View menu group, the Locals window (seen in Figure 8.5) not only supplies information on variables and properties, but also allows for the changing of control property values.

**FIGURE 8.5**

The Locals window provides information on variables and control properties in current scope.



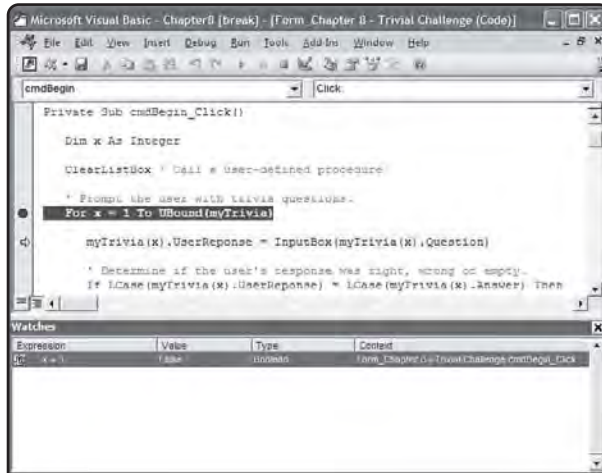
To change a property or variable's value using the Locals window, simply click the item in the Value column and type a new variable or property value.

## Watch Window

In addition to breakpoints, the Watch window can aid you in troubleshooting or debugging program code. Accessed from the View menu item, the Watch window can track values of expressions and break when expressions are True or have been changed. In a nutshell, the Watch window keeps track of Watch expressions, as seen in Figure 8.6.

A basic Watch expression allows you to graphically track the value of an expression throughout the life of a program. Moreover, you can create a Watch expression that pauses program execution when an expression has been changed or is True.

For example, let's say you know a bug occurs in your program because the value of a variable is being set incorrectly. You know the value of the variable is changing, but you do not know

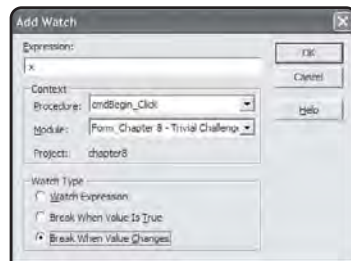
**FIGURE 8.6**

Tracking Watch expressions through the Watch window.

where in the code it is being changed. Using a Watch expression, you can create an expression that pauses program execution whenever the value of the variable in question changes.

Though Watch expressions can be created from within the Watch window, it is much easier to create them by right-clicking a variable or property name in the Code window and choosing Add Watch.

Figure 8.7 shows the dialog box that appears when you add a watch.

**FIGURE 8.7**

Adding a Watch expression.

The Add Watch dialog box provides many options for creating Watch expressions. Essentially, creating a Watch expression with the Add Watch dialog box is broken into three parts: Expression, Context, and Watch Type.

An *expression* is a variable, property, function call, calculation, or combination of valid expressions. By default, the expression's value is the name of the variable or property you're trying to watch. The *context* is the scope of the variable or property being watched. There are three values displayed:



- **Procedure.** Defines the procedure where the expression is evaluated.
- **Module.** Defines the module where the variable or property resides.
- **Project.** Displays the name of the current project.

The Watch Type determines how Visual Basic responds to the expression:

- **Watch Expression.** Displays the Watch expression and expression value in the Watch window.
- **Break When Value Is True.** Visual Basic breaks program execution when the value of the Watch expression is `True`.
- **Break When Value Changes.** Visual Basic breaks program execution when the value of the expression changes.

## INPUT VALIDATION

Input validation is a great place to begin learning about error handling and bug fixing. This is because a good portion of program errors come from unexpected user input or responses.

For example, what do you think would happen if a user entered a letter or character as an operand into a math quiz game? A better question is, “How do I prevent a user from entering a letter into a text box intended for numbers?” What about a game that prompts a user for a level; would testing that the input is a number be enough? Probably not, as most games have only a few levels, so you would also need to test for a range of numbers. In a nutshell, the art of input validation depends on a talented programmer with enough foresight to prevent errors before they happen.

In Microsoft Access, developers can create input validation for forms, tables, and queries with an input mask. In this section, I show you how to build input validation using VBA.

## IsNumeric

Sometimes preventing input errors can be as easy as determining whether a user has entered a number or a string. There are times when you will want the user to enter his or her name, or maybe you are looking for a number such as age. Either way, Visual Basic provides the `IsNumeric` function for testing such scenarios.

The `IsNumeric` function takes a variable or expression as a parameter and returns a Boolean value of `True` if the variable or expression is a number; `False` if it is not.

```
Private Sub cmdCheckForNumber_Click()
```

```
    If IsNumeric(txtNumbersOnly.Value) = False Then
```

```
MsgBox "Enter numbers only please."
```

```
Else
```

```
MsgBox "Thank you for entering numbers."
```

```
End If
```

```
End Sub
```

In the preceding example, you can see that by testing the `Value` of the text box with the `IsNumeric` function, I want the user to enter a number. If the `IsNumeric` function returns the Boolean value of `False`, I know that the user has entered something other than a number.

Conversely, you could use the `IsNumeric` function to check for a string value. Simply change the conditional expression in the `If` statement.

```
Private Sub cmdCheckForNumber_Click()
```

```
    If IsNumeric(txtStringDataOnly.Value) = True Then
```

```
        MsgBox "Enter string data only please."
```

```
    Else
```

```
        MsgBox "Thank you for entering non-numeric data."
```

```
    End If
```

```
End Sub
```



**Remember that Access 2007 VBA returns a `Null` value for an empty text box.**

When testing for numeric or nonnumeric data, it is also common to test for an empty text box using the `IsNull` function, as the next procedure demonstrates.

```
Private Sub cmdCheckForNumber_Click()
```

```
    If IsNull(txtStringDataOnly.Value) Then
```

```
MsgBox "Please enter a string value into the text box."  
Exit Sub
```

```
End If
```

```
If IsNumeric(txtStringDataOnly.Value) = True Then
```

```
    MsgBox "Enter string data only please."
```

```
Else
```

```
    MsgBox "Thank you for entering non-numeric data."
```

```
End If
```

```
End Sub
```

The `IsNull` function takes an expression as a parameter (in my example, the `Value` property of a text box), and returns a Boolean value (`True` or `False`), depending on whether or not the expression is `Null`.

## Checking a Range of Values

You may find at times that testing a value for a particular data type (such as number or string) is not enough to prevent input errors. Sometimes it is necessary to check for a range of values. For example, you may wish to prompt a user to enter a number from 1 to 100. Maybe you want a person to pick a letter from *a* to *z*.

Testing for a range of values involves a little more thought from the programmer. Specifically, your first thought should be to know the range(s) needing to be tested and whether the ranges are numeric or character based. Testing ranges of values with numbers or strings uses the same programming constructs consisting of compound conditions.

Let's take the 1 to 100 example I mentioned earlier. As seen here, I continue to use the `IsNumeric` function as part of the overall testing for a range of numbers (1 to 100):

```
Private Sub cmdCheckRange_Click()
```

```
    If IsNumeric(txtInput.Value) = True Then
```

```
        If Val(txtInput.Value) >= 1 And _
```

```
Val(txtInput.Value) <= 100 Then

    MsgBox "You entered a number between 1 and 100."

Else

    MsgBox "Your number is out of range."

End If

Else

    MsgBox "Please enter a number from 1 to 100."

End If

End Sub
```

Testing for a range of letters (characters) is not much different, if you remember that all characters (letters or numbers) can be represented with character codes, also known as ANSI (American National Standards Institute) values. For example, say I want a user to enter a letter in the range of **a** through **m** (including both uppercase and lowercase letters within the range). I can still use the `IsNumeric` function to help me out, but I need to perform some additional tests.

```
Private Sub cmdCheckRange_Click()

    If IsNumeric(txtInput.Value) = False Then

        If Asc(UCase(txtInput.Value)) >= 65 And _
            Asc(UCase(txtInput.Value)) <= 77 Then

            MsgBox "You entered a letter between a and m."

        Else

            MsgBox "Your letter is out of range."

        End If

    End If

End Sub
```

```
Else
```

```
    MsgBox "Please enter a letter between a and m."
```

```
End If
```

```
End Sub
```

In the preceding code, I'm looking for the `IsNumeric` function to return a `False` value, which means the input was not a number. Next I use the `Asc` function, which converts a character to its corresponding ANSI value.

Using compound conditions, I specifically look for an ANSI range between 65 and 77, the numbers that represent the capital letters A through M. You may also notice that I used the function `UCase` in association with the `Asc` function. The `UCase` function converts lowercase letters to uppercase letters. If I didn't convert the characters to uppercase, I would have needed to check for the lowercase letters as well (ANSI numbers 97 to 109).

A list of common ANSI values (character codes) can be found in Appendix A, "Common Character Codes," at the end of this book.

## ERROR HANDLING

Whenever your program interacts with the outside world, you should provide some form of error handling to counteract unexpected inputs or outputs. One way of providing error handling is to write your own error-handling routines.

*Error-handling routines* are the traffic control for your program. Such routines can handle any kind of programming or human-generated errors you can think of. They should not only identify the error, but try to fix it—or at least give the program or interacting human a chance to do so.

To begin error handling in a procedure, use the `On Error GoTo` statement to signify that you are going to use an error-handling routine:

```
On Error GoTo ErrorHandler
```

This statement can go anywhere in your procedure, but should be placed toward the top, generally right after any procedure-level variable declarations.

`ErrorHandler` is the name I've chosen for my error-handling routine. Error-handling routines can be given any name: `ErrorBin`, `ErrorBucket`, or whatever you like.

## WHERE DID GoTo Go?

The keyword `GoTo` is a carryover from an old programming practice made popular in various languages such as `BASIC` and `COBOL`. A `GoTo` was regularly used for designing and building modularized programs. To break programs into manageable pieces, programmers would create modules and link them together using the keyword `GoTo`.

After years of programming with `GoTo`, programmers began to realize that this created messy “spaghetti-like” code, which at times became nearly impossible to debug. Fortunately, event-driven and object-oriented programming techniques have virtually eliminated the use of `GoTo`.

Once an error handler has been declared, errors generated in the procedure are directed to the error-handling routine, as demonstrated in this example.

```
Public Function Verify_Input() As Boolean
```

```
On Error GoTo ErrorHandler
```

```
    'get Input from user
```

```
    Exit Function
```

```
ErrorHandler:
```

```
    MsgBox ("An error has occurred.")
```

```
    Resume
```

```
End Function
```

It is necessary to execute the `Exit Function` or `Exit Sub` statements before program execution enters the error-handling routine. Without these statements, a procedure that executes without errors executes the error handler as well. That’s an important note, so let me repeat it again in italics: *Without an `Exit` statement, a procedure that executes without errors executes the error-handling routine as well.*

Error handling begins by typing the name of the error handler followed by a colon. Within the error handler, you write code to respond to the error. In the previous example, I simply use a message box to report that an error has occurred.

The `Resume` keyword takes program execution back to the statement where the error occurred. Note that there are three possible ways for returning program control to the procedure:

- `Resume`. By itself, the keyword `Resume` returns program control to where the error occurred.
- `Resume Next`. The `Resume Next` statement returns program control to the statement after the statement where the error occurred.
- `Resume Label`. The `Resume Label` statement returns program control to a predetermined line number, as seen in the following code.

```
Public Function Verify_Input() As Boolean
```

```
    On Error GoTo ErrorHandler
```

```
    'get Input from user
```

```
    BeginHere:
```

```
    Exit Function
```

```
ErrorHandler:
```

```
    MsgBox ("An error has occurred.")
```

```
    Resume BeginHere:
```

```
End Function
```

Generally speaking, message boxes are good ways to let a user know an error has occurred. However, knowing that an error has occurred is not enough; the user also needs to know what caused the error and possible solutions to resolve the error.

In the next section, you learn how to identify specific and custom errors using the `Err` object.

## The Err Object

When a user encounters an error in your program, he should be provided with a clear, precise description of the problem and resolution. The Err object provides VBA programmers with an accessible means of finding or triggering Microsoft Windows-specific errors.

Essentially the Err (short for error) object maintains information about errors that occur in the current procedure. This information is stored in the form of properties. The most common of the Err properties follow:

- Description contains a description of the current error.
- Number contains the error number of the current error (0 to 65,535).
- Source contains the name of the object that generated the error.

Table 8.1 contains just a few of VBA’s more common trappable error numbers and descriptions. For a complete list of error numbers and descriptions, consult Appendix C, “Trappable Errors.”

TABLE 8.1 COMMON ERROR NUMBERS AND DESCRIPTIONS	
Error Number	Error Description
11	Division by 0
53	File Not Found
61	Disk Full
71	Disk Not Ready
76	Path Not Found

In the next program example, I use an error handler to check for division by 0.

```
Private Sub cmdDivide_Click()

    On Error GoTo ErrorBin

    MsgBox "Result is " & Val(txtOperand1.Value) _
        / Val(txtOperand2.Value)

    Exit Sub

ErrorBin:
```



```
MsgBox "Error Number " & Err.Number & ", " & Err.Description
```

```
Resume Next
```

```
End Sub
```

There may be times when an error occurs in your program that is similar to that of a given `Err` description, but does not trigger the specific `Err` number. The ability to trigger errors can be achieved through the `Err` object's `Raise` method.

The `Raise` method allows you to trigger a specific error condition, thus displaying a dialog box to the user. The `Raise` method takes a number as a parameter. For example, the following statement triggers a Disk Not Ready dialog box:

```
Err.Raise 71
```

Besides providing descriptive error messages, error handling prevents many unwanted problems for your users. In other words, error handling may prevent your program from crashing. Users expect an error for division by 0, but they don't expect division by 0 to crash their applications.

## The Debug Object

The `Debug` object is common with VBA programmers for troubleshooting problems by sending output to the Immediate window. The `Debug` object has two methods: `Print` and `Assert`. The `Print` method prints the value of its parameter and sends it to the Immediate window.

The `Assert` method conditionally breaks program execution when the method is reached. More specifically, the `Assert` method takes an expression as a parameter, which evaluates to a Boolean value. If the expression evaluates to `False`, the program's execution is paused. Otherwise, program execution continues. The next procedure demonstrates the use of the `Assert` method.

```
Private Sub cmdDivide_Click()
```

```
    Dim passedTest As Boolean
```

```
    On Error GoTo ErrorBin
```

```
    If Val(txtOperand2.Value) = 0 Then
```

```
        passedTest = False
```

```
    Else
```

```
        passedTest = True
    End If

    ' Conditionally pause program execution.
    Debug.Assert passedTest

    MsgBox "Result is " & Val(txtOperand1.Value) _
        / Val(txtOperand2.Value)

Exit Sub

ErrorBin:

    MsgBox "Error Number " & Err.Number & ", " & Err.Description

    Resume Next

End Sub
```

## FILE PROCESSING

Do you know that you, too, can build your own database to store a collection of data? You can—and you can do it with file I/O (input/output) and a little help from this chapter.

Within VBA there are many techniques for building and managing file I/O routines. File I/O is the approach taken by programmers to manage data stored in files. Data files that you create can be viewed and edited through Microsoft text editors such as Notepad.

Most data files that you work with are built upon a common foundation, much like a database. The data files you learn about in this chapter share the following relationships and building blocks:

- **Data File.** A collection of data that stores records and fields
- **Record.** A row of related data that contains one or more fields, separated by a space, tab, or comma
- **Field.** An attribute in a record, which is the smallest component in a data file

An example data file is shown in Figure 8.8. The `trivia.dat` data file is used in the chapter-based program. It has five records, with each record containing three fields separated by commas. This is called a *comma-delimited file*.

**FIGURE 8.8**

A sample data file.

In the sections to come, I show you how to build and manage your own data files using sequential file access.

## About Sequential File Access

Data files created with sequential file access have records stored in a file, one after another, in sequential order. When you access data files with sequential file access, records must be read in the same order in which they were written to the file. In other words, if you want to access the 20th record in a data file, you must first read records 1 through 19.

Sequential file access is useful and appropriate for small data files. If you find that your sequential file access program is starting to run slowly, you might want to change file access to an RDBMS such as Microsoft Access.

## Opening a Sequential Data File

The first step in creating or accessing a data file is to open it. Microsoft provides an easy-to-use facility for opening a data file through the `Open` function.

```
Open "Filename" For {Input | Output | Append} As #FileNumber [Len = Record Length]
```

The `Open` function takes three parameters. `Filename` describes the name of the file you wish to open or create. `Input|Output|Append` is a list from which you pick one to use. `#FileNumber` is a number from 1 to 511 that is used for referencing the file. `Len` is an optional parameter that can control the number of characters buffered. The sequential access modes are shown in Table 8.2.

I use the `Open` method to create a new file for output called `quiz.dat`.

```
Open "quiz.dat" For Output As #1
```



The `Filename` attribute can contain paths in addition to filenames. For example, if you want to create employee records in a file named `employee.dat` on removable storage, you could use the following syntax.

```
Open "a:\employee.dat" For Output As #1
```

**TABLE 8.2 SEQUENTIAL ACCESS MODES**

Mode	Description
Input	Reads records from a data file
Output	Writes records to a data file
Append	Writes or appends records to the end of a data file

The result of the `Open` function varies depending on the initial action chosen. If the `Input` parameter is chosen, the `Open` function searches for the file and creates a buffer in memory. If the file is not found, VBA generates an error.



*A **buffer** is an area where data is temporarily stored.*

If the file specified is not found, a new file is created using the `Filename` parameter as the filename. Note that the `Output` mode always overwrites an existing file. After a data file has been successfully opened, you can then read from it, write to it, and close it.

### Reading Sequential Data from a File

If you want to read records from a data file, you must use the `Input` parameter with the `Open` function.

```
Open "quiz.dat" For Input As #1
```

Once the file is opened for input, use the `Input` function to retrieve fields from the file.

```
Input #Filenum, Fields
```

The `Input` function takes two parameters: `#Filenum` and a list of fields. For example, if you want to read the first record in a data file called `quiz.dat` (assuming `quiz.dat` contains three fields for each record), you could use the following program statements.

```
Dim liQuestionNumber as Integer
```

```
Dim lsQuestion as String
```

```
Dim lsAnswer as String
```

```
Open "quiz.dat" For Input As #1
```

```
Input #1, liQuestionNumber, lsQuestion, lsAnswer
```

Notice that I pass three variables as the field list to the `Input` function. These variables hold the contents of the first record found.

By now, you may be thinking, “So far, so good, but how do I read all records in a data file?” The answer involves something new and something old. First, you must use a loop to search through the data file. Second, your loop’s condition should use the `EOF` function.

The `EOF` (end of file) function tests for the end of the data file. It takes a file number as a parameter, returning a `True` Boolean value if the end of the file is found or `False` if the end of file has not been reached.

To test for the end of file, the `EOF` function looks for an `EOF` marker placed at the end of a file by the `Close` function. I discuss closing data files later in the chapter.

```
Dim liQuestionNumber as Integer
Dim lsQuestion as String
Dim lsAnswer as String

Open "quiz.dat" For Input As #1

Do Until EOF(1)
    Input #1, liQuestionNumber, lsQuestion, lsAnswer
    List1.AddItem "Question number: " & _
        liQuestionNumber & lsQuestion
Loop
```

The preceding loop iterates until the `EOF` function returns a `True` value. Inside the loop, each record is read one at a time. After a record is read, the `print` method of a picture box control is used to output two of the fields (`liQuestionNumber` and `lsQuestion`) for display.

## Writing Sequential Data to a File

In order to write data to a sequential file, you want to use either the `Output` mode, which creates a new file for writing, or the `Append` mode, which writes records to the end of a data file. Note that these are two separate lines of code.

```
Open "quiz1.dat" For Output As #1
Open "quiz.dat" For Append As #1
```

After opening a file for writing, you can use the `Write` function to write records.

```
Write #Filenummer, Fields
```

The `Write` function takes two parameters: `#FileNumber` and a list of fields. `#FileNumber` denotes the file number used in the `Open` function, and the `Fields` parameter is a list of strings, numbers, variables, and properties that you want to use as fields.

For example, if I want to create a data file and write quiz records to it, I could use the following syntax.

```
Open "quiz.dat" For Output As #1
Write #1, 1, "Is Visual Basic an Event Driven language?", "Yes"
```

I could also use variable names for my fields list.

```
Write #1, liQuestionNumber, lsQuestion, lsAnswer
```

Either way, VBA outputs numbers as numbers and strings as strings surrounded with quotation marks.

## Closing Data Files

As you may have guessed, closing a data file is an important part of file processing. Specifically, closing a data file performs the following operations:

- Writes the EOF marker
- When using the Output or Append mode, writes records to the physical file in the sequential order in which they were created
- Releases the file number and buffer for memory conservation

To close a data file, simply use the `Close` function after all file processing has completed.

```
Close #FileNumber
```

The `Close` function takes the file number as its only parameter. For example, to close the file `quiz.dat` after writing one record, I could use the `Close` function:

```
Open "quiz.dat" For Output As #1

Write #1, 1, "Is Visual Basic an Event Driven language?", "Yes"

Close 1
```

If the `Close` function is used without any parameters, it closes all open sequential data files.

## Error Trapping for File Access

Error trapping is almost always a must when dealing with file I/O. Why? Have you ever tried to access a CD-ROM from Windows Explorer, only to get an error message because there is no CD-ROM in the CD-ROM drive? What if the CD-ROM is in the drive, but the file is not found, or better yet—the file is there but it's corrupted?

There are all types of potential errors when dealing with data files and file I/O. Your best bet is to plan ahead and create error-trapping or error-handling routines. In fact, it is safe to promote error trapping in any procedure that opens, writes, reads, appends, or closes files.

An adequate facility for capturing file I/O errors is to use VBA's `Err` object. The `Err` object contains preexisting codes for various known errors such as "File Not Found," "Disk Not Ready," and many more that can be used to your advantage.

Here's an error-handling routine for a quiz game that uses the `Err` object to check for specific errors when the game attempts to open a file in the form `Load` event:

```
Private Sub Form_Load()
```

```
    On Error GoTo ErrorHandler:
```

Like any other error-handling routine, I start my procedure by declaring an error-handling label with an `On Error GoTo` statement. You can insert unique labels throughout your code as I've done here with the `BeginHere: label`.

```
    BeginHere:
```

Labels can serve useful purposes as long as you keep their existence minimal and easy to follow. As you see later in the code, I choose the `BeginHere: label` as a good starting point in this procedure.

```
    Open "quiz.dat" For Input As #1
    Exit Sub
```

After opening the sequential data file, the procedure is exited, providing no errors have occurred.

```
ErrorHandler:
    Dim liResponse As Integer
```

If an error does occur in opening the file, my guess is that it is one of the following error conditions (error codes). You can see in the following code that I'm using the `Select Case` structure to check for specific `Err` object codes. If an error code is found, the user is prompted

with an opportunity to fix the problem. If the user decides to retry the operation, the program resumes control to the `BeginHere:` label.

```
Select Case Err.Number
```

```
Case 53
```

```
    'File not found
    liResponse = MsgBox("File not found!", _
        vbRetryCancel, "Error!")

    If liResponse = 4 Then 'retry
        Resume BeginHere:
    Else
        cmdQuit_Click
    End If
```

```
Case 71
```

```
    'Disk not ready
    liResponse = MsgBox("Disk not ready!", _
        vbRetryCancel, "Error!")

    If liResponse = 4 Then 'retry
        Resume BeginHere:
    Else
        cmdQuit_Click
    End If
```

```
Case 76
```

```
    liResponse = MsgBox("Path not found!", _
        vbRetryCancel, "Error!")

    If liResponse = 4 Then 'retry
        Resume BeginHere:
```



```
Else
    cmdQuit_Click
End If
```

```
Case Else
```

```
MsgBox "Error in program!", , "Error"
cmdQuit_Click
```

```
End Select
```

```
End Sub
```

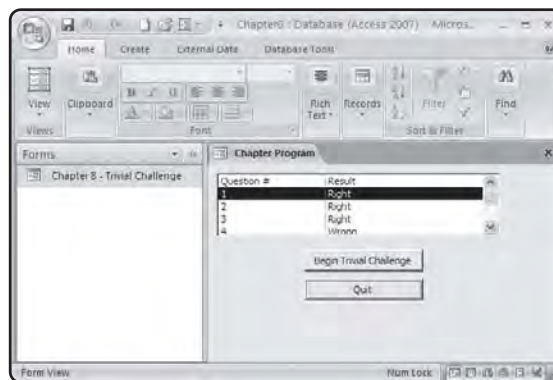
## CHAPTER PROGRAM: TRIVIAL CHALLENGE

The Trivial Challenge program, shown in Figure 8.9, is a fun game that uses chapter-based techniques and concepts such as data files, sequential file access, and error handling.

Program code for the game is broken into two separate code modules. The standard module contains a public user-defined type, which stores and manages quiz components such as question numbers, question, answer, and user's response. Most of the program code is in the form class module where the game's logic is managed.

**FIGURE 8.9**

Using chapter-based concepts to build the Trivial Challenge program.



Controls and properties that build the Trivial Challenge program are described in Table 8.3.

**TABLE 8.3    CONTROLS AND PROPERTIES OF THE TRIVIAL CHALLENGE PROGRAM**

Control	Property	Property Value
Form	Name	Chapter 8 - Trivial Challenge
	Caption	Chapter Program
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No
List Box	Name	lstResults
	Row Source Type	Value List
	Column Count	2
	Column Heads	Yes
Command Button	Name	cmdBegin
	Caption	Begin Trivial Challenge
Command Button	Name	cmdQuit
	Caption	Quit

Module-level code defines a public user-defined type.

Option Compare Database  
Option Explicit

Public Type Trivia

    QuestionNumber As Integer  
    Question As String  
    Answer As String  
    UserResponse As String

End Type

Shown next is the form class module code for building the remainder of the Trivial Challenge program.

Option Compare Database  
Option Explicit

' Declare dynamic array of Trivia type.

```
Dim myTrivia() As Trivia
```

---

```
Private Sub cmdBegin_Click()
```

```
    Dim x As Integer
```

```
    ClearListBox ' Call a user-defined procedure
```

```
    ' Prompt the user with trivia questions.
```

```
    For x = 1 To UBound(myTrivia)
```

```
        myTrivia(x).UserResponse = InputBox(myTrivia(x).Question)
```

```
        ' Determine if the user's response was right, wrong or empty.
```

```
        If LCase(myTrivia(x).UserResponse) = LCase(myTrivia(x).Answer) Then
```

```
            Me.lstResults.AddItem myTrivia(x).QuestionNumber _  
                & ";" & "Right"
```

```
        Else
```

```
            If myTrivia(x).UserResponse = "" Then
```

```
                ' User did not respond (pressed Cancel on input box).
```

```
                Me.lstResults.AddItem myTrivia(x).QuestionNumber _  
                    & ";" & "No Response"
```

```
            Else
```

```
                Me.lstResults.AddItem myTrivia(x).QuestionNumber _  
                    & ";" & "Wrong"
```

```
            End If
```

```
        End If
```

```
    Next x
```

```
End Sub
```

---

```
Private Sub cmdQuit_Click()
```

```
DoCmd.Quit
End Sub
```

---

```
Private Sub Form_Load()

    ' Create initial element in dynamic array.
    ReDim myTrivia(1)

    ' Add header to each column in the list box if one
    ' hasn't already been added.
    If lstResults.ListCount = 0 Then
        Me.lstResults.AddItem "Question #;Result"
    End If

    ' Load trivia questions into memory.
    LoadTrivia

End Sub
```

---

```
Public Sub ClearListBox()

    Dim x As Integer

    ' Clear list box
    For x = 1 To (lstResults.ListCount - 1)
        Me.lstResults.RemoveItem lstResults.ListCount - 1
    Next x

End Sub
```

---

```
Public Sub LoadTrivia()

    On Error GoTo ErrorHandler

    ' Open file for sequential input using
    ' the application's current path.
```

```
Open Application.CurrentProject.Path & "\" & "trivia.dat" For Input As #1

' Read all records until end of file is reached.
' Store each question and answer in a user defined type.
Do While EOF(1) = False

    ' Read trivia data into variables.
    Input #1, myTrivia(UBound(myTrivia)).QuestionNumber, _
        myTrivia(UBound(myTrivia)).Question, _
        myTrivia(UBound(myTrivia)).Answer

    ' Print debug data to the immediate window.
    'Debug.Print myTrivia(UBound(myTrivia)).QuestionNumber, _
        'myTrivia(UBound(myTrivia)).Question, myTrivia(UBound(myTrivia)).Answer

    If EOF(1) = False Then

        ' Increment dynamic array for each next trivia question.
        ReDim Preserve myTrivia(UBound(myTrivia) + 1)

    End If

Loop

' Close the sequential file.
Close #1

Exit Sub

ErrorHandler:

MsgBox "Error number " & Err.Number & Chr(13) & _
    Err.Description

End Sub
```

---

```
Private Sub lstResults_Click()  
  
    ' Display the selected question back to the user.  
    If lstResults.ListIndex = -1 Then  
        Exit Sub  
    End If  
  
    MsgBox myTrivia(lstResults.ListIndex + 1).Question  
  
End Sub
```

## SUMMARY

- Debugging is the process by which programmers identify, find, and correct software errors.
- Software bugs are generally grouped into one of three categories: syntax errors, runtime errors, and logic errors.
- The Visual Basic Environment includes many debugging features such as breakpoints, the Immediate window, the Locals window, and the Watch window.
- Breakpoints are used to pause program execution.
- The Immediate window can be used to ascertain variable and property values.
- Variable and property values can be altered in the Immediate window.
- Variable and property values within scope can be viewed and managed in the Locals window.
- Watch expressions can be created and managed in the Watch window.
- Input validation generally involves checking for numeric or nonnumeric data entered by the user.
- Programmers can use input validation to check for a range of numbers or characters.
- The `IsNull` function takes an expression as a parameter and returns a Boolean value (True or False) depending on whether or not the expression is Null.
- VBA programmers often use the `Err` and `Debug` objects to aid in debugging and error handling.
- The `Err` object contains properties for discerning the current error number and error description.
- The `Debug` object contains methods commonly used in conjunction with the Immediate window for pausing program execution and displaying program output.

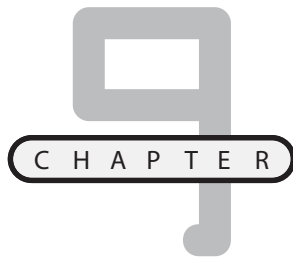
- VBA error-handling routines are initiated using the `On Error GoTo` statement.
- There are three possible ways to return program control to the procedure using the keywords `Resume`, `Resume Next`, and `Resume Label`.
- Data files contain records and fields.
- In VBA, file processing can be achieved with sequential file access using the `Open`, `Write`, `Input`, and `Close` methods.
- Error handling should always be incorporated into file-processing routines.

### PROGRAMMING CHALLENGES

1. **Build a form with one text box and one command button. The text box should receive the user's name. In the `Click` event of the command button, write code to validate that the user has entered nonnumeric data and display the outcome in a message box. Test your program by entering numeric data into the text box.**
2. **Build a form with one text box and one command button. The text box should receive a number between 1 and 10. In the `Click` event of the command button, write code to validate that the data entered is a number and that it is in the range of 1 to 10. Use a message box to display the outcome.**
3. **Create a data file called `friends.dat`. Insert a few records into the `friends.dat` file. The record layout should have three fields (phone number, first name, and last name), which should look similar to this:**

"111-222-3333", "Michael", "Massey"

4. **Create a form that allows a user to view all records in the `friends.dat` file. Populate a list box on the form with the phone numbers and names of friends. Remember to use error handling when opening the `friends.dat` file.**
5. **Create a form that allows a user to enter more friends into the `friends.dat` file. Retrieve information from the user with text boxes on the form. Remember to use the `Append` option when opening the `friends.dat` file and use error handling accordingly.**



# MICROSOFT ACCESS SQL

**I**n this chapter, I show you how to use Microsoft Access SQL for querying and managing databases without the help of Access wizards. I specifically show you two subsets of the Access SQL language, called DML and DDL.

If you're new to database languages such as SQL, consider this chapter a prerequisite for Chapter 10, "Database Programming with ADO." Even if you've worked with SQL before, you may find this chapter a refresher for Microsoft Access SQL syntax and common functionality.

## INTRODUCTION TO ACCESS SQL

Most databases, including Microsoft Access, incorporate one or more data languages for querying information and managing databases. The most common of these data languages is SQL (Structured Query Language), which contains a number of commands for querying and manipulating databases. SQL commands are typically grouped into one of two categories known as *data manipulation language (DML)* commands and *data definition language (DDL)* commands.

Microsoft Access SQL follows a standard convention known as ANSI SQL, which is used by many database vendors, including Microsoft, Oracle, and IBM. Each manufacturer, however, incorporates its own proprietary language-based



functions and syntax. Access SQL is no exception with key differences in reserved words and data types.

To demonstrate Access SQL, I use Access Queries in SQL View with Microsoft's sample Northwind database that can be found in your Access 2007 Local Templates area, as seen in Figure 9.1.



**FIGURE 9.1**

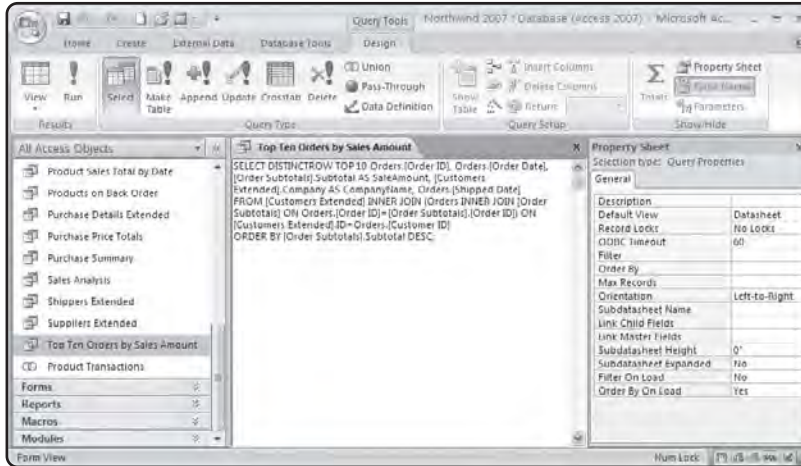
Creating the Microsoft Access 2007 Northwind database.



You can bypass the default Northwind login form by holding down the Shift key while simultaneously opening the Northwind database file.

Building queries in Microsoft Access is much like the experience of building tables and forms in Access. Essentially, Microsoft provides wizards and graphical interfaces for building everything, including queries. In this chapter, I show you how to go beyond Access wizards to build your own queries using SQL!

To access the SQL window in Access, open a query by double-clicking it, or right-click it and select either Design View or Open. After the query is opened, right-click the query's tab and select SQL View from the menu, which I've done for the Top Ten Orders by Sales Amount query in Figure 9.2.

**FIGURE 9.2**

Viewing the Top Ten Orders by Sales Amount query in SQL View.

SQL is not considered to be a full-fledged programming language like VBA, C, or Java. In this author's mind, a real programming language must, at minimum, contain facilities for creating variables, as well as structures for conditional logic branches and iteration through loops. Regardless, SQL is a powerful language for asking the database questions (also known as *querying*).

**Pronounced *sequel*, SQL was originally developed by IBM researchers in the 1970s. It has become the de facto database manipulation language for many database vendors. For database users, mastering SQL has become a sought-after skill set in the information technology world. Most persons who master the SQL language have no trouble finding well-paid positions.**

To provide readability in the sections to come, I use a preferred syntax nomenclature for SQL:

- All SQL commands and reserved language keywords are in uppercase. For example, SELECT, FROM, WHERE, and AND are all Access SQL commands and reserved keywords.
- Even though Microsoft Access is not a case-sensitive application, table and column names used in SQL statements use the same case as defined in the database. For example, a column defined as `EmployeeId` is spelled `EmployeeId` in the SQL query.
- Table and column names that contain spaces must be enclosed in brackets. For example, the column name `Customer Number` must be contained in SQL as `[Customer Number]`. Failure to do so causes errors or undesired results when executing your queries.

- A query can be written on a single line. For readability, I break SQL statements into logical blocks on multiple lines. For example, look at this SQL statement:

```
SELECT [Order Details].OrderID, SUM(CCur([UnitPrice]*[Quantity]*(1-[Discount])/
100)*100) AS Subtotal FROM [Order Details] GROUP BY [Order Details].OrderID;
```

It should look like this:

```
SELECT      [Order Details].OrderID,
SUM        (CCur([UnitPrice]*[Quantity]*(1-[Discount]))/100)*100)
AS          Subtotal
FROM        [Order Details]
GROUP BY    [Order Details].OrderID;
```

## DATA MANIPULATION LANGUAGE

SQL contains many natural-language commands for querying, computing, sorting, grouping, joining, inserting, updating, and deleting data in a relational database. These querying and manipulation commands fall under the Data Manipulation Language subset, also known as DML.

### Simple SELECT Statements

To retrieve information from a relational database, SQL provides the simple SELECT statement. A simple SELECT statement takes the following form.

```
SELECT  ColumnName, ColumnName
FROM    TableName;
```

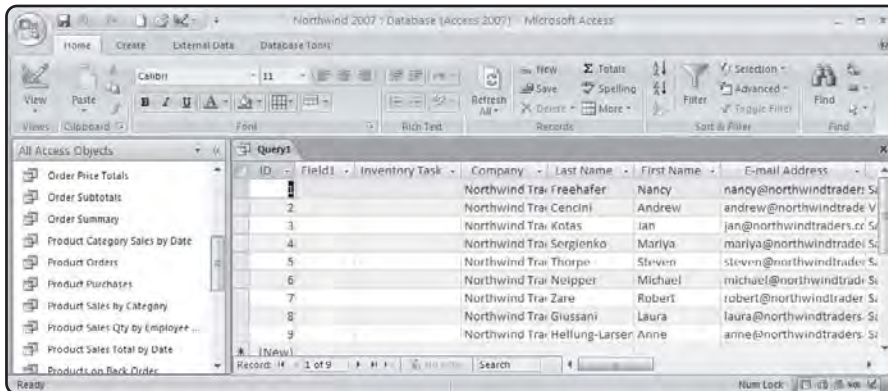
The SELECT clause identifies one or more column names in a database table(s). After identifying the columns in the SELECT clause, you must tell the database which table(s) the columns live in using the FROM clause. It is customary in SQL to append a semicolon (;) after the SQL statement to indicate the statement's ending point.

To retrieve all rows in a database table, the wildcard character (\*) can be used like this.

```
SELECT  *
FROM    Employees;
```

You can execute SQL queries in Access in one of a couple of ways. You can simply save your query and double-click from the Access Objects window. Or, leaving your SQL View window open, right-click the tab of your query and select Datasheet view from the menu.

Another way to execute your SQL queries is to click the red exclamation mark (!) in the Results area of the Design tab. Either way, the results from the preceding query (`SELECT * FROM Employees;`) running against the Northwind database are shown in Figure 9.3.



**FIGURE 9.3**

Viewing the results of a simple query.



A *result set* is a common phrase used to describe the result or records returned by a SQL query.

Sometimes it is not necessary to retrieve all columns in a query. To streamline your query, supply specific column names separated by commas in the `SELECT` clause.

```
SELECT  [Last Name], [First Name], [Job Title]
FROM    Employees;
```

In the preceding query, I ask the database to retrieve only the last names, first names, and titles of each employee record. Output is shown in Figure 9.4.



Microsoft Access allows users to create table and column names with spaces. Use brackets (`[ ]`) to surround table and column names with spaces in SQL queries. Failure to do so can cause errors when running your queries.

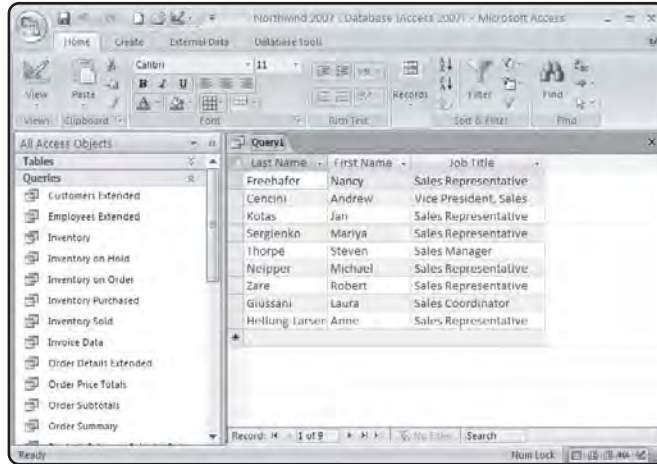
You can change the order in which the result set displays columns by changing the column order in your SQL queries.

```
SELECT  Title, FirstName, LastName
FROM    Employees;
```

Changing the order of column names in a SQL query does not alter the data returned in a result set, but rather its column display order.

**FIGURE 9.4**

Specifying individual column names in a SQL query.



## Conditions

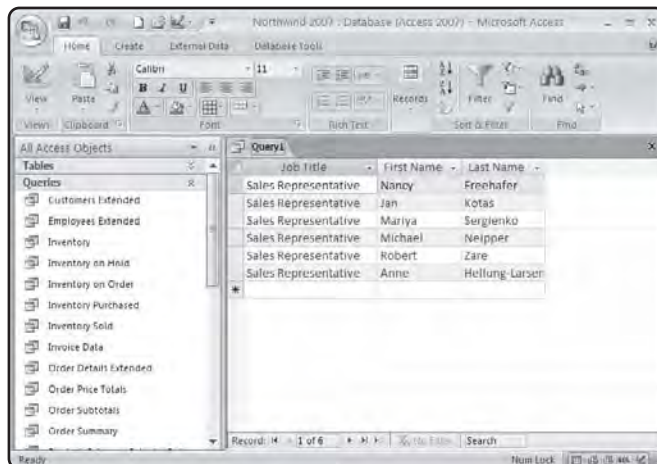
SQL queries allow basic conditional logic for refining the result set returned by the query. Conditions in SQL are built using the WHERE clause.

```
SELECT [Job Title], [First Name], [Last Name]
FROM Employees
WHERE [Job Title] = 'Sales Representative';
```

In the preceding query, I use a condition in the WHERE clause to only return rows from the query where the employee's title equals Sales Representative. Output from this query is seen in Figure 9.5.

**FIGURE 9.5**

Using conditions in the WHERE clause to refine the result set.



Note that textual data such as 'Sales Representative' in the WHERE clause's expression must always be enclosed by single quotes.

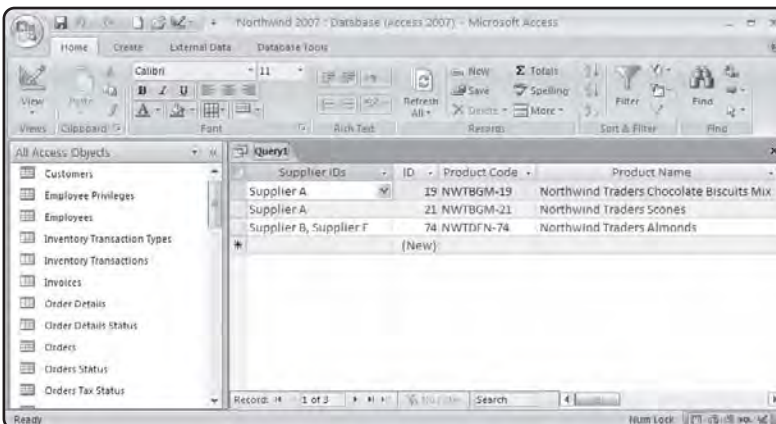
SQL conditions work much like the conditions you've already learned about in Access VBA in that the WHERE clause's condition evaluates to either True or False. You can use the operators seen in Table 9.1 in SQL expressions.

**TABLE 9.1**    **CONDITIONAL OPERATORS USED IN SQL EXPRESSIONS**

Operator	Description
=	Equals
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

To demonstrate conditional operators, the next query returns the rows in the Products table where the value for Reorder Level is less than or equal to 5. Output is seen in Figure 9.6.

```
SELECT *
FROM Products
WHERE [Reorder Level] <= 5;
```



**FIGURE 9.6**

Refining the products result set with conditional operators.



Note that single quotes are not used to surround numeric data when searching numeric data types.

SQL queries also can contain compound conditions using the keywords AND, OR, and NOT. The next two SQL queries demonstrate the use of compound conditions in the WHERE clause.

```
SELECT    *
FROM      Products
WHERE     [Reorder Level] <= 5 AND [List Price] < 10;

SELECT    *
FROM      Products
WHERE     [Reorder Level] <= 5 AND NOT([List Price] = 10);
```

Before moving on to the next section on SQL, I'd like to share with you a paradigm shift. Believe it or not, most SQL programmers are the translators for their companies' information needs. To better understand this, think of SQL programmers as the intermediaries between business people and the unwieldy database. The business person comes in to your office and says, "I'm concerned about products mistakenly listed as discontinued. Could you tell me what products we have in stock that have been discontinued?" As the SQL programmer, you smile and say, "Sure, give me a minute." After digesting what your colleague is asking, you translate the inquiry into a question the database understands—in other words, a SQL query such as the following.

```
SELECT    *
FROM      Products
WHERE     Discontinued = TRUE;
```

Within seconds, your query executes and you print out the results for your amazed and thankful colleague.

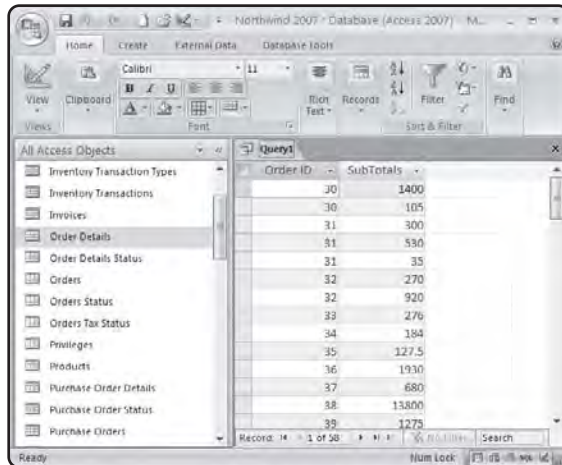
## Computed Fields

Computed fields do not exist in the database as columns. Instead, computed fields are generated using calculations on columns that do exist in the database. Simple calculations such as addition, subtraction, multiplication, and division can be used to create computed fields.

When creating computed fields in SQL, the AS clause assigns a name to the computed field. The next SQL statement uses a computed field to calculate subtotals based on two columns (Unit Price and Quantity) in the Order Details table. Output is seen in Figure 9.7.



```
SELECT [Order ID], ([Unit Price] * Quantity) AS SubTotals
FROM [Order Details];
```

**FIGURE 9.7**

Using SQL to build computed fields.

Note the presence of the `SubTotals` column name in Figure 9.7. The `SubTotals` field does not exist in the `Order Details` table. Rather, the `SubTotals` field is created in the SQL statement using the `AS` clause to assign a name to an expression. Although parentheses are not required, I use them in my computed field's expression to provide readability and order of operations, if necessary.

## Built-In Functions

Just as VBA incorporates many intrinsic functions such as `Val`, `Str`, `UCase`, and `Len`, SQL provides many built-in functions for determining information on your result sets. You learn about these SQL *aggregate functions* in this section:

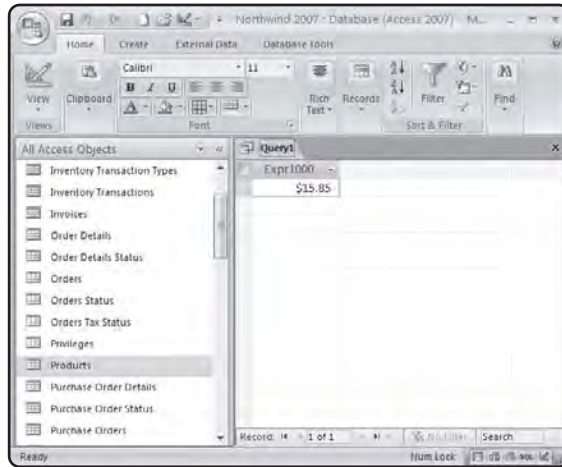
- AVG
- COUNT
- FIRST, LAST
- MIN, MAX
- SUM
- DISTINCT

The `AVG` function takes an expression, such as a column name for a parameter, and returns the mean value in a column or other expression.



```
SELECT      AVG([List Price])
FROM        Products;
```

The preceding SQL statement returns a single value, which is the mean value of the List Price column in the Products table. Output is seen in Figure 9.8.



**FIGURE 9.8**

Using the AVG function to calculate the mean value of a column.

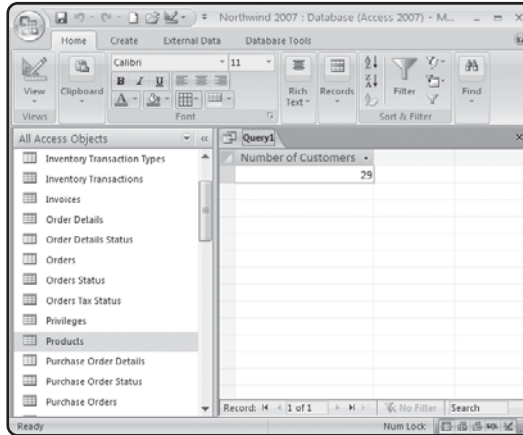
Notice in Figure 9.8 that the column heading gives no clue as to the meaning of the SQL statement's return value. To correct this, simply use the AS clause.

```
SELECT      AVG([List Price]) AS [Average Unit Price]
FROM        Products;
```

The COUNT function is a very useful function for determining how many records are returned by a query. For example, the following SQL query uses the COUNT function to determine how many customer records are in the Customers table.

```
SELECT      COUNT(*) AS [Number of Customers]
FROM        Customers;
```

Figure 9.9 reveals the output from the COUNT function in the preceding SQL statement. Note that it's possible to supply a column name in the COUNT function, but the wildcard character (\*) performs a faster calculation on the number of records found in a table.

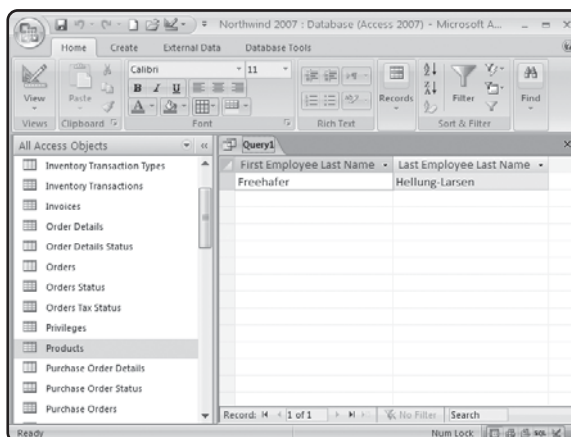
**FIGURE 9.9**

Displaying the result of a COUNT function.

The FIRST and LAST functions return the first and last records in a result set, respectively. Because records are not necessarily stored in alphanumeric order, the FIRST and LAST functions may produce seemingly unexpected results. The results, however, are accurate. These functions report the first and last expressions in a result set as stored in a database and returned by the SQL query.

```
SELECT  FIRST([Last Name]) AS [First Employee Last Name],
        LAST([Last Name]) AS [Last Employee Last Name]
FROM    Employees;
```

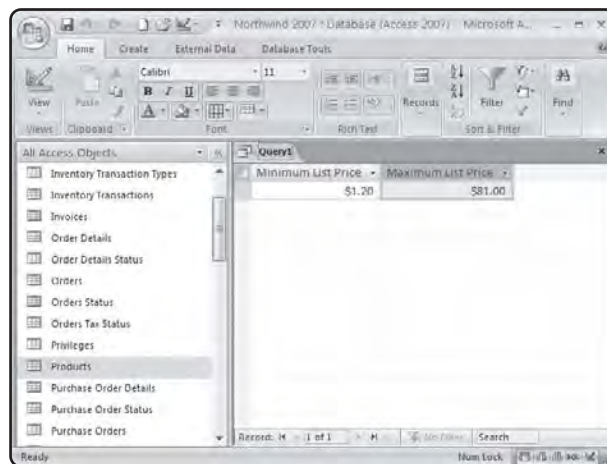
The preceding SQL statement uses the FIRST and LAST functions to retrieve the first and last results on the last name of employee records in the Employees table. Output is seen in Figure 9.10.

**FIGURE 9.10**

Using the FIRST and LAST functions to retrieve the first and last values of a result set.

To determine the minimum and maximum values of an expression in SQL, use the MIN and MAX functions, respectively. Like other SQL functions, the MIN and MAX functions take an expression and return a value. The next SQL statement uses these two functions to determine the minimum and maximum unit prices found in the Products table. Output is seen in Figure 9.11.

```
SELECT    MIN([List Price]) AS [Minimum List Price],
          MAX([List Price]) AS [Maximum List Price]
FROM      Products;
```



**FIGURE 9.11**

Retrieving the minimum and maximum values from an expression using the MIN and MAX functions.

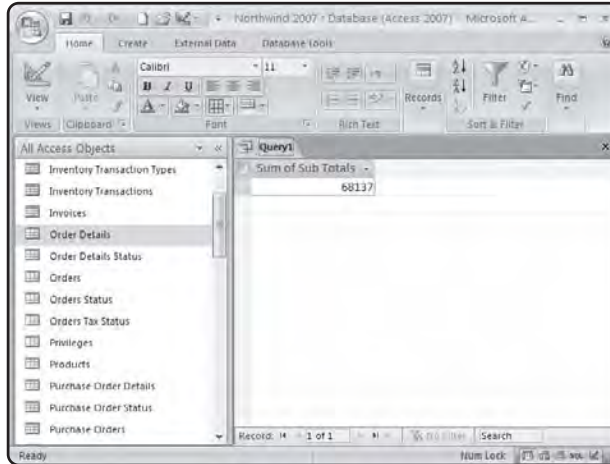
The SUM function takes an expression as argument and returns the sum of values. The SUM function is used in the next SQL statement, which takes a computed field as an argument to derive the sum of subtotals in the Order Details table.

```
SELECT    SUM([Unit Price] * Quantity) AS [Sum of Sub Totals]
FROM      [Order Details];
```

Output from the SQL statement using the SUM function is displayed in Figure 9.12.

The last built-in function for this section is DISTINCT, which returns a distinct set of values for an expression. To demonstrate, if I want to find a unique list of countries for the suppliers in the Northwind database, I need to sift through every record in the Suppliers table and count each distinct country name. Or, I could use the DISTINCT function to return a distinct value for each country in the Country/Region column.

```
SELECT    DISTINCT([Country/Region])
FROM      Suppliers;
```

**FIGURE 9.12**

Displaying the output of the SUM function.

## Sorting

You may recall from the discussions surrounding the FIRST and LAST functions that data stored in a database are not stored in any relevant order, including alphanumeric. Most often, data is stored in the order in which it was entered, but not always. If you need to retrieve data in a sorted manner, use the ORDER BY clause.

The ORDER BY clause is used at the end of a SQL statement to sort a result set (records returned by a SQL query) in alphanumeric order. Sort order for the ORDER BY clause can either be ascending (A to Z, 0 to 9) or descending (Z to A, 9 to 0). Use the keyword ASC for ascending or DESC for descending. Note that neither the ASC nor DESC keywords are required with the ORDER BY clause, and that the default sort order is ascending.

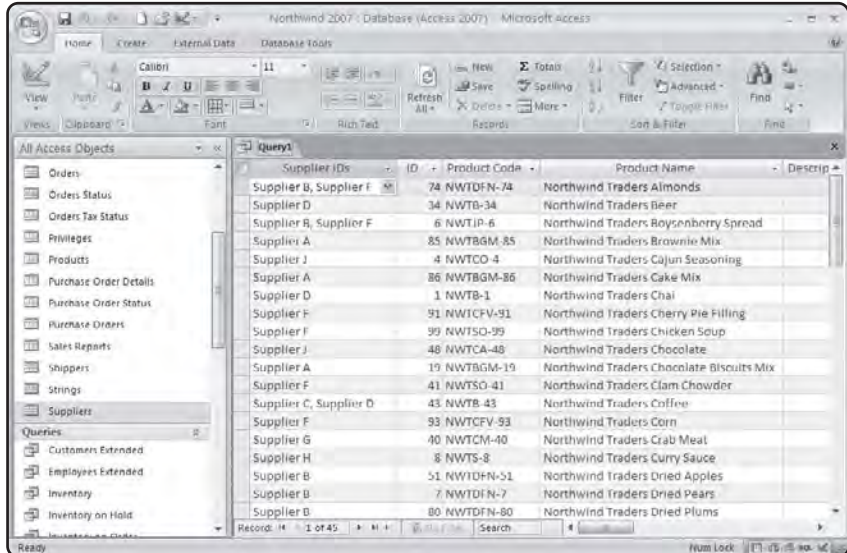
To properly use the ORDER BY clause, simply follow the clause with a *sort key*, which is a fancy way of saying a column name to sort on. The optional keywords ASC and DESC follow the sort key.

To exhibit SQL sorting techniques, study the next two SQL statements and their outputs, shown in Figures 9.13 and 9.14.

```
SELECT      *
FROM        Products
ORDER BY    [Product Name] ASC;
```

**FIGURE 9.13**

Using the ORDER BY clause and the ASC keyword to sort product records by product name in ascending order.

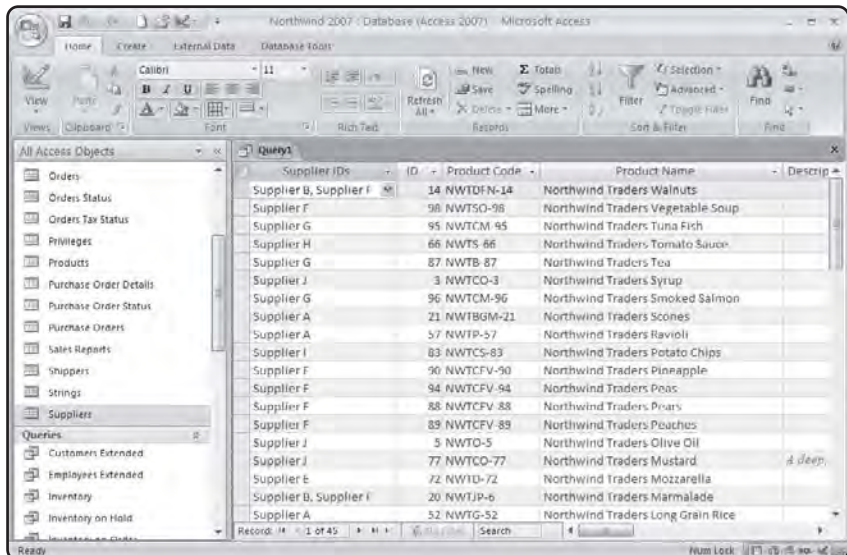


Supplier IDs	ID	Product Code	Product Name	Descrip
Supplier B, Supplier F	74	NWTFN-74	Northwind Traders Almonds	
Supplier D	34	NWTFB-34	Northwind Traders Beer	
Supplier B, Supplier F	6	NWTFP-6	Northwind Traders Boysenberry Spread	
Supplier A	85	NWTFBM-85	Northwind Traders Brownie Mix	
Supplier J	4	NWTFCO-4	Northwind Traders Cajun Seasoning	
Supplier A	86	NWTFBGM-86	Northwind Traders Cake Mix	
Supplier D	1	NWTFB-1	Northwind Traders Chai	
Supplier F	91	NWTFV-91	Northwind Traders Cherry Pie Filling	
Supplier F	99	NWTFSO-99	Northwind Traders Chicken Soup	
Supplier J	48	NWTFCA-48	Northwind Traders Chocolate	
Supplier A	19	NWTFBGM-19	Northwind Traders Chocolate Biscuits Mix	
Supplier F	41	NWTFSO-41	Northwind Traders Clam Chowder	
Supplier C, Supplier D	43	NWTFB-43	Northwind Traders Coffee	
Supplier F	93	NWTFV-93	Northwind Traders Corn	
Supplier G	40	NWTFM-40	Northwind Traders Crab Meat	
Supplier H	8	NWTF-8	Northwind Traders Curry Sauce	
Supplier B	51	NWTFDN-51	Northwind Traders Dried Apples	
Supplier B	7	NWTFDN-7	Northwind Traders Dried Pears	
Supplier B	80	NWTFDN-80	Northwind Traders Dried Plums	

```
SELECT      *
FROM        Products
ORDER BY    [Product Name] DESC;
```

**FIGURE 9.14**

Using the ORDER BY clause and the DESC keyword to sort product records by product name in descending order.



Supplier IDs	ID	Product Code	Product Name	Descrip
Supplier B, Supplier F	14	NWTFDN-14	Northwind Traders Walnuts	
Supplier F	98	NWTFSO-98	Northwind Traders Vegetable Soup	
Supplier G	95	NWTFM-95	Northwind Traders Tuna Fish	
Supplier H	66	NWTF-66	Northwind Traders Tomato Sauce	
Supplier G	87	NWTFB-87	Northwind Traders Tea	
Supplier J	3	NWTFCO-3	Northwind Traders Syrup	
Supplier G	96	NWTFM-96	Northwind Traders Smoked Salmon	
Supplier A	21	NWTFBGM-21	Northwind Traders Scones	
Supplier A	57	NWTFP-57	Northwind Traders Ravioli	
Supplier I	83	NWTFCS-83	Northwind Traders Potato Chips	
Supplier F	90	NWTFV-90	Northwind Traders Pineapple	
Supplier F	94	NWTFV-94	Northwind Traders Pitas	
Supplier F	88	NWTFV-88	Northwind Traders Pears	
Supplier F	88	NWTFV-88	Northwind Traders Peaches	
Supplier J	5	NWTFO-5	Northwind Traders Olive Oil	
Supplier J	77	NWTFCO-77	Northwind Traders Mustard	
Supplier E	72	NWTFD-72	Northwind Traders Mozzarella	
Supplier B, Supplier F	20	NWTFP-20	Northwind Traders Marmalade	
Supplier A	52	NWTFG-52	Northwind Traders Long Grain Rice	

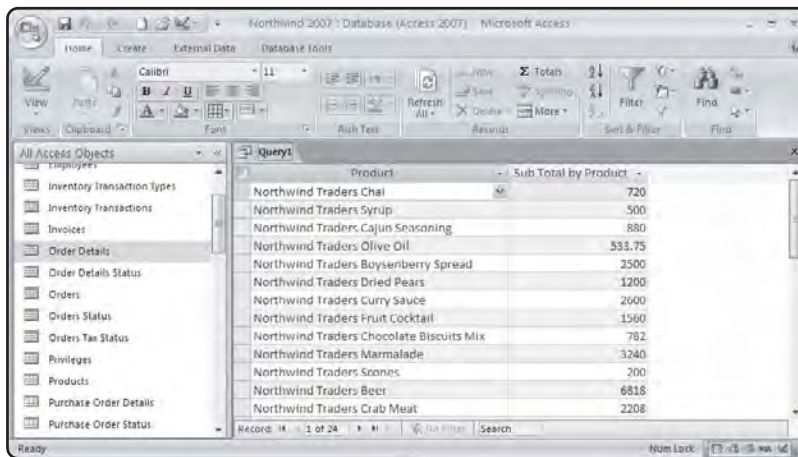
## Grouping

Grouping in SQL provides developers with an opportunity to group like data together. Without the use of grouping, built-in functions such as SUM and AVG calculate every value in a column. To put like data into logical groups of information, SQL provides the GROUP BY clause.

In the next SQL statement, I use the GROUP BY clause to group a computed field by product ID in the Products table.

```
SELECT      [Product ID], SUM([Unit Price] * Quantity) AS [Sub Total by Product]
FROM        [Order Details]
GROUP BY    [Product ID];
```

Notice the output from the preceding SQL statement in Figure 9.15. Even though I specified Product ID as the desired column, the output column and the data it contains show as a product name. This occurs because the Order Details table uses a SQL lookup to retrieve the Product Name by Product ID.



**FIGURE 9.15**

Using the GROUP BY clause to group like data together.

There are times when you need conditions on your groups. In these events, you cannot use the WHERE clause. Instead, SQL provides the HAVING clause for condition building when working with groups. To demonstrate, I modify the previous SQL statement to use a HAVING clause, which asks the database to retrieve only groups that have a subtotal by product greater than 15,000.00.

```
SELECT      [Product ID], SUM([Unit Price] * Quantity) AS [Sub Total by Product]
FROM        [Order Details]
```



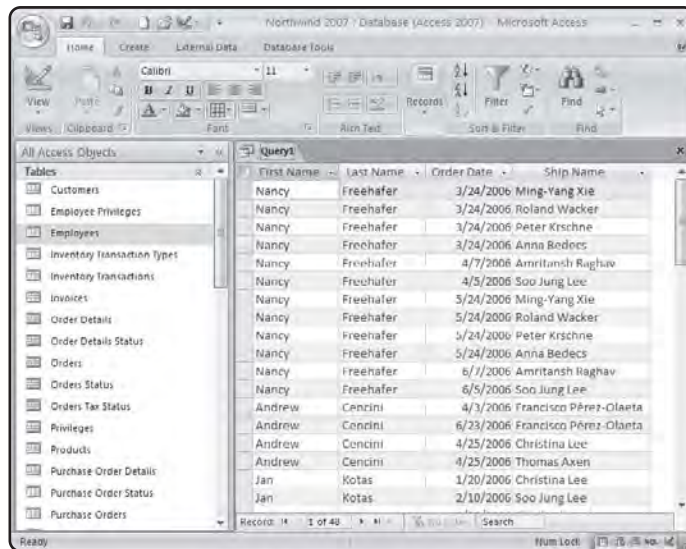
```
GROUP BY [Product ID]
HAVING SUM ([Unit Price] * Quantity) > 15000.00;
```

## Joins

*Joins* are used when you need to retrieve data from more than one table. Specifically, a SQL join uses keys to combine records from two tables where a primary key from one table is matched up with a foreign key from another table. The result is a combination of result sets from both tables where a match is found. If a match is not found, information from either table is discarded in the final result set.

SQL joins are created by selecting columns from more than one table in the `SELECT` clause, including both table names in the `FROM` clause, and matching like columns from both tables in the `WHERE` clause. The following example join's output is shown in Figure 9.16.

```
SELECT [First Name], [Last Name], [Order Date], [Ship Name]
FROM Employees, Orders
WHERE ID = [Employee ID];
```



First Name	Last Name	Order Date	Ship Name
Nancy	Freehafer	3/24/2006	Ming-yang Xie
Nancy	Freehafer	3/24/2006	Roland Wacker
Nancy	Freehafer	3/24/2006	Peter Kirschne
Nancy	Freehafer	3/24/2006	Anna Bedecs
Nancy	Freehafer	4/7/2006	Amritansh Raghav
Nancy	Freehafer	4/5/2006	Soo Jung Lee
Nancy	Freehafer	5/24/2006	Ming-yang Xie
Nancy	Freehafer	5/24/2006	Roland Wacker
Nancy	Freehafer	5/24/2006	Peter Kirschne
Nancy	Freehafer	5/24/2006	Anna Bedecs
Nancy	Freehafer	6/7/2006	Amritansh Raghav
Nancy	Freehafer	6/5/2006	Soo Jung Lee
Andrew	Cencini	4/3/2006	Francisco Pérez-Olaeta
Andrew	Cencini	6/23/2006	Francisco Pérez-Olaeta
Andrew	Cencini	4/25/2006	Christina Lee
Andrew	Cencini	4/25/2006	Thomas Axen
Jan	Kotas	1/20/2006	Christina Lee
Jan	Kotas	2/10/2006	Soo Jung Lee

**FIGURE 9.16**

Joining the Employees and Orders tables with the `WHERE` clause.

In Figure 9.16, I've retrieved columns from both the Employees and Orders tables where the Employee ID values from both tables match (ID from Employees table and Employee ID from the Orders table). If the join keys from both tables are spelled the same, I must explicitly tell SQL what table name I'm referring to by using dot notation, as seen here.

```
WHERE TableName.Key = TableName.Key;
```

Even in cases where the dot notation is not required, you can leverage the dot notation in a `SELECT` clause to explicitly denote which table a column belongs to, which ultimately provides readability in your queries. An example of using the dot notation in the `SELECT` clause for readability is seen next.

```
SELECT  Employees.[First Name], Employees.[Last Name], Orders.[Order Date],  
        Orders.[Ship Name]  
FROM    Employees, Orders  
WHERE   ID = [Employee ID];
```



If the expression in the `WHERE` clause is incorrect, a Cartesian join results. A *Cartesian result* is when the query returns every possible number of combinations from each table involved.

The preceding join is typically called a *natural join*, where a row from one table matches a row from another table using a common column and matching column value. There are times, however, when you want rows from one table that do not match rows in the other table. SQL solves this dilemma with an *outer join*.

There are two types of outer joins: left outer joins and right outer joins. A *left outer join* includes all records from the left (first) of the two tables even if there are no matching rows from right (second) table. The *right outer join* includes all rows from the right (second) table even if there are no matching rows from the left (first) table. Outer joins are created using the `LEFT JOIN` or `RIGHT JOIN` keywords in the `FROM` clause and replacing the `WHERE` clause with an `ON` clause.

To demonstrate, let's start off with a natural join query that will show me all orders with matching invoice data.

```
SELECT  Orders.[Order ID], Orders.[Customer ID], Orders.[Order Date],  
        Orders.[Shipped Date], Invoices.[Invoice ID], Invoices.[Invoice Date]  
FROM    Orders, Invoices  
WHERE   Orders.[Order ID] = Invoices.[Order ID]
```

There's only one problem with this join—it's possible I have orders that don't yet have a matching invoice. The natural join in this case does not show me orders without an invoice. I can, however, view both orders with invoices and orders without invoices using a left outer join. You can see this in the following code and in Figure 9.17.



```

SELECT Orders.[Order ID], Orders.[Customer ID], Orders.[Order Date],
       Orders.[Shipped Date], Invoices.[Invoice ID], Invoices.[Invoice Date]
FROM   Orders LEFT JOIN Invoices
ON     Orders.[Order ID] = Invoices.[Order ID]

```

Order ID	Customer	Order Date	Shipped Date	Invoice ID	Invoice Date
38	Company BB	3/10/2006	3/11/2006	9	3/24/2006
39	Company H	3/22/2006	3/24/2006	8	3/24/2006
40	Company J	3/24/2006	3/24/2006	7	3/24/2006
41	Company G	3/24/2006			
42	Company I	3/24/2006	4/7/2006	36	4/4/2006
43	Company K	3/24/2006			
44	Company A	3/24/2006			
45	Company BB	4/7/2006	4/7/2006	23	4/4/2006
46	Company I	4/5/2006	4/5/2006	22	4/4/2006
47	Company F	4/8/2006	4/8/2006	21	4/4/2006
48	Company H	4/5/2006	4/5/2006	20	4/4/2006
50	Company V	4/5/2006	4/5/2006	19	4/4/2006
51	Company Z	4/5/2006	4/5/2006	18	4/4/2006
55	Company Z	4/5/2006	4/5/2006	17	4/4/2006

**FIGURE 9.17**

Using a left outer join to find orders with and without a matching invoice.

Orders without matching invoices

Looking at Figure 9.17, you can see there are orders with no matching invoice, which a natural join would have missed.

To create my outer join, I inserted the `LEFT JOIN` keywords into my `FROM` clause and replaced the `WHERE` keyword with the `ON` keyword.

```

FROM   Orders LEFT JOIN Invoices
ON     Orders.[Order ID] = Invoices.[Order ID]

```

## INSERT INTO Statement

You can use `SQL` to insert rows into a table with the `INSERT INTO` statement. The `INSERT INTO` statement inserts new records into a table using the `VALUES` clause.

```

INSERT INTO Books
VALUES ('1234abc456edf', 'Beginning SQL', 'Vine',
      'Michael', 'Thomson Course Technology');

```

Though not required, matching column names can be used in the `INSERT INTO` statement to clarify the fields with which you're working.

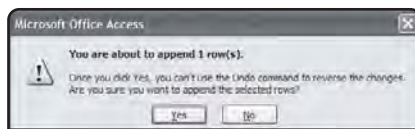
```
INSERT INTO Books (ISBN, Title, [Last Name], [First Name], Publisher)
VALUES ('1234abc456edf', 'Beginning SQL', 'Vine',
       'Michael', 'Thomson Course Technology ');
```

Using matching column names is necessary, not just helpful, when you only need to insert data for a limited number of columns in a table. A case in point is when working with the AutoNumber field type, which Access automatically creates for you when inserting a record.

The concept of working with an AutoNumber and an INSERT INTO statement is shown in the following SQL statement.

```
INSERT INTO Shippers (Company, [Business Phone])
VALUES ('Slow Boat Express', '123-456-9999');
```

When inserting (appending) a record into a table with the INSERT INTO statement, Access prompts you to confirm or undo the command, as seen in Figure 9.18.



**FIGURE 9.18**

Appending (inserting) a new record into a table with the INSERT INTO statement.

## UPDATE Statement

You can use the UPDATE statement to change field values in one or more tables. The UPDATE statement works in conjunction with the SET keyword:

```
UPDATE Products
SET [Reorder Level] = [Reorder Level] + 5;
```

You supply the table name to the UPDATE statement and use the SET keyword to update any number of fields that belong to the table in the UPDATE statement. In my example, I'm updating every record's Reorder Level field by adding the number 5. Notice that I said *every* record! Because I didn't use a WHERE clause, the UPDATE statement updates every row in the table.

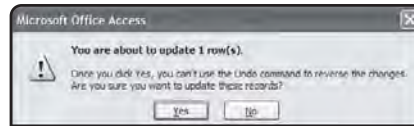
In the next UPDATE statement, I use a WHERE clause to put a condition on the number of rows that receive updates.

```
UPDATE Products
SET [Reorder Level] = [Reorder Level] + 5
WHERE ID = 34;
```

It is possible for SQL programmers to forget to place conditions on their UPDATE statements. Because there is no undo or rollback feature after an update has successfully occurred, pay attention to the dialog box, which you see in Figure 9.19, that appears before you commit (save) changes.

**FIGURE 9.19**

Updating one record in the Products table with an UPDATE statement.



## DELETE Statement

The DELETE statement removes one or more rows from a table. It's possible to delete all rows from a table using the DELETE statement and a wildcard.

```
DELETE    *
FROM      Products;
```

More often than not, conditions are placed on DELETE statements using the WHERE clause.

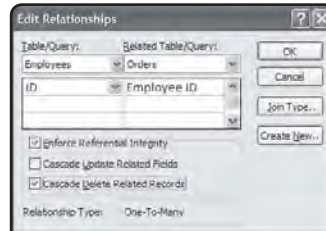
```
DELETE    *
FROM      Products
WHERE     ID = 34;
```

Once again, pay close attention to Access's informational dialog boxes when performing any inserts, updates, or deletes on tables.

The DELETE statement can perform cascade deletes on tables with one-to-many relationships if the Cascade Delete Related Records option is chosen in the Edit Relationships window, as seen in Figure 9.20.

**FIGURE 9.20**

Selecting cascade deletes on one-to-many table relationships.



With the Cascade Delete Related Records option chosen for the `Employees` and `Orders` tables, any employee records deleted also initiate a corresponding deletion in the `Orders` table where a matching `Employee ID` was found.

## DATA DEFINITION LANGUAGE

Data definition language, also known as DDL, contains commands that define a database's attributes. Most commonly, DDL creates, alters, and drops tables, indexes, constraints, views, users, and permissions.

In this section, you investigate a few of the more common uses for DDL in beginning database programming:

- Creating tables
- Altering tables
- DROP statements

### Creating Tables

Creating tables in DDL involves using the `CREATE TABLE` statement. With the `CREATE TABLE` statement, you can define and create a table, its columns, column data types, and any constraints that might be needed on one or more columns. In its simplest form, the `CREATE TABLE` syntax and format is shown here.

```
CREATE TABLE    TableName
                (FieldName FieldType,
                 FieldName FieldType,
                 FieldName FieldType);
```

The `TableName` attribute defines the table to be created. Each `FieldName` attribute defines the column to be created. Each `FieldName` has a corresponding `FieldType` attribute, which defines the column's data type.

The next `CREATE TABLE` statement creates a new table called `Books` that contains seven columns.

```
CREATE TABLE    Books
                (ISBN Text,
                 Title Text,
                 AuthorLastName Text,
                 AuthorFirstName Text,
                 Publisher Text,
```

```
Price Currency,  
PublishDate Date);
```

The `CREATE TABLE` statement allows you to specify if one or more columns should not allow `NULL` values. By default, columns created in the `CREATE TABLE` statement allow `NULL` entries. To specify a not `NULL` column, use the `Not Null` keywords.

```
CREATE TABLE Books  
    (ISBN Text Not Null,  
     Title Text,  
     AuthorLastName Text,  
     AuthorFirstName Text,  
     Publisher Text,  
     Price Currency,  
     PublishDate Date);
```

Using the `Not Null` keywords sets the column's `Required` attribute to `Yes`.

## Altering Tables

You can use the `ALTER TABLE` statement to alter tables that have already been created. Three common uses of the `ALTER TABLE` statement are to add column(s) to an existing table, to change the field type attributes of one or more columns, or to remove a column from a table.

The next `ALTER TABLE` statement adds a `Salary` column to the `Employees` table with the help of the `ADD COLUMN` keywords.

```
ALTER TABLE Employees  
ADD COLUMN Salary Currency;
```

Adding the `Salary` column with the `ALTER TABLE` statement appends the new column to the end of the `Employees` table. To change a column's data type, use the `ALTER COLUMN` keywords in conjunction with the `ALTER TABLE` statement.

```
ALTER TABLE Books  
ALTER COLUMN Title Memo;
```

In the preceding `ALTER TABLE` statement, I changed the data type of the `Title` column from a `Text` data type to a `Memo` data type. To remove a column from a table in Access, use the `DROP COLUMN` keywords in conjunction with the `ALTER TABLE` statement:

```
ALTER TABLE Employees  
DROP COLUMN Salary;
```



Access does not always warn you of your impending database alterations. In the case of dropping (removing) a column, Access simply performs the operation without mention.

## DROP Statement

The **DROP** statement can be used to remove many entities from a database such as tables, indexes, procedures, and views. In this section, you see how the **DROP** statement is used to drop a table from a database.

Removing a table from a database with the **DROP** statement is really quite easy. Simply supply the entity type to be dropped, along with the entity name.

```
DROP TABLE Books;
```

In the preceding example, the dropped entity typed is a **TABLE** and the entity name is **Books**. Once again, beware: Access does not always warn you when it modifies the database. In the **DROP TABLE** example, Access simply executes the command without any confirmation.

## SUMMARY

- Most relational databases, including Access 2007, contain a version of SQL for retrieving data and manipulating database entities.
- Data definition language (DDL) is a set of SQL commands used to define attributes, such as tables and columns, of a relational database.
- Data manipulation language (DML) is a set of commands for querying, computing, sorting, grouping, joining, inserting, updating, and deleting data in a relational database.
- SQL statements are freeform, meaning one SQL statement can be written on one or more lines. For readability, SQL programmers break SQL statements into one or more logical groups on multiple lines.
- Information is retrieved from a relational database using **SELECT** statements.
- Simple and compound conditions can be used in SQL statements using the **WHERE** clause.
- Computed field values are derived using calculations in SQL statements.
- Computed fields are given display names using the **AS** clause.
- SQL contains many aggregate, or built-in, functions such as **COUNT**, **DISTINCT**, **MIN**, **MAX**, and **SUM**.
- Database records returned by a SQL statement are not sorted by default. To sort SQL query results, use the **ORDER BY** clause.
- SQL query results can be grouped using the **GROUP BY** clause.

- Natural joins are created by matching key fields in two or more tables in the `WHERE` clause.
- Incorrect joins can produce an unwanted Cartesian product.
- A left outer join includes all records from the left (first) of the two tables even if there are no matching rows from the right (second) table.
- A right outer join includes all rows from the right (second) table even if there are no matching rows from the left (first) table.
- Records can be manually inserted into a table using the `INSERT INTO` statement.
- The `UPDATE` statement can be used to update fields in a database table.
- Records in a table can be removed using the `DELETE` statement.
- Tables can be manually created using the `CREATE TABLE` statement.
- In its simplest form, the `CREATE TABLE` statement defines the table's name, its columns, and its column types.
- The `ALTER TABLE` statement can be used to add columns to an existing table, update a column's data type, and remove a column from a table.
- The `DROP` statement is used for removing tables, indexes, views, and procedures from a database.

## PROGRAMMING CHALLENGES

Leverage the Microsoft Access 2007 Northwind database and a SQL View window for all the following challenges.

1. Write and test a SQL query that retrieves all columns from the Employee Privileges table.
2. Write and test a SQL query that retrieves only the first and last names and a business phone number from the Customers table.
3. Write and test a SQL query that uses a computed field to calculate the total cost of each record in the Order Details table.
4. Write and test the SQL query that returns the total number of records in the Employees table.
5. Use the Orders table to write and test the SQL query that returns the sum of shipping fees grouped by customer.
6. Using the INSERT INTO statement, write a SQL query that inserts a new record into the Employees table.
7. Update the unit price by \$3.25 in the Products table for all products by Supplier A.
8. Delete all records in the Products table for the soups category only.
9. Using DDL commands, create a new table called HomesForSale. The HomesForSale table should contain the following fields: StreetAddress, City, State, ZipCode, SalePrice, and ListDate. Ensure that the StreetAddress column does not allow Null values.
10. Using DDL commands, add three new columns to the HomesForSale table: AgentLastName, AgentFirstName, and AgentPhoneNumber.
11. Using DDL commands, remove the HomesForSale table from the database.



*This page intentionally left blank*



# CHAPTER 10

## DATABASE

# PROGRAMMING WITH

# ADO

With a basic knowledge of VBA programming, you can leverage the power of Microsoft's ActiveX Data Objects (commonly referred to as ADO) to access and manage data sources such as Oracle, Microsoft SQL Server, Microsoft Access, and many others. In this chapter, I will show you how to connect to a remote Microsoft Access database; retrieve, update, add, and delete records using VBA; and I will explain the ADO application programming interface.

### ADO OVERVIEW

For many years, Microsoft has implemented and supported quite a few database programming models such as *RDO* (*Remote Data Objects*), *DAO* (*Data Access Objects*), and *ADO* (*ActiveX Data Objects*).

ADO is an object-based programming model that allows programmers in many Microsoft programming languages, such as Visual C++, Visual Basic, ASP (Active Server Pages), C#, and of course VBA, to access and manage data sources. ADO has become Microsoft's most important and reliable method for data source connectivity, retrieval, and management.

Data sources can be as simple as text files, or they can be more sophisticated relational data sources such as Microsoft Access, Microsoft SQL Server, or even

non-Microsoft databases like Oracle's RDBMS. Specifically, ADO enables you to connect to data sources that support *open database connectivity* (ODBC). ADO also allows you to leverage the power of *structured query language* (SQL) for those data sources that support it.

Each ADO programming endeavor involves working with the ADO API, also called the *ADO object model*. An *API* (*application programming interface*) is a set of interfaces, or classes, that allow you to access the low-level functionality of programming models such as ADO. The ADO API consists of many objects, collections, events, methods, and properties.

Although most Microsoft programming languages support the ADO object model, there are some slight differences in how ADO is implemented and used within each language. In this chapter, you learn how ADO is implemented and used in Access VBA.

Before getting started, you might want to familiarize yourself with some key ADO terminology and objects, as outlined in Table 10.1.

**TABLE 10.1 KEY ADO TERMINOLOGY**

Item	Description
<b>Connection</b>	A connection is how you gain access to a data source. In ADO, connections are achieved through the <code>Connection</code> object.
<b>Command</b>	In ADO, commands are defined as a set of instructions, such as SQL statements or a stored procedure, that typically inserts, deletes, or updates data. ADO commands are embodied in the <code>Command</code> object.
<b>Field</b>	ADO recordsets contain one or more fields. ADO fields are implemented with the <code>Field</code> object, which contains properties for field names, data types, and values.
<b>Parameter</b>	Parameters allow you to use variables to pass information to commands such as SQL statements. ADO uses the <code>Parameter</code> object to build parameterized queries and stored procedures.
<b>Recordset</b>	Rows returned by a command, such as a SQL statement, are stored in recordsets. ADO's <code>Recordset</code> object allows you to iterate through the returned rows and insert, update, and delete rows in the recordset.

## CONNECTING TO A DATABASE

Before you and ADO can work with data in a data source, you must first establish a connection using the `Connection` object. To declare variables of ADO object type, use the `ADODB` library name followed by a period and a specific ADO object type such as `Connection`. An example of declaring an ADO object variable of `Connection` type is seen here.

```
Dim myConnection As ADODB.Connection
```

If you're using ADO to connect to your current Microsoft Access application, you can use the `CurrentProject` object's `AccessConnection` property to set an ADO connection object to your `Connection` object variable. An example of connecting to a local database is shown next.

```
Private Sub cmdConnectToLocalDB_Click()

    On Error GoTo ConnectionError

    'Declare connection object variable
    Dim localConnection As ADODB.Connection

    'Set current Access connection to Connection object variable
    Set localConnection = CurrentProject.AccessConnection

    MsgBox "Local connection successfully established."

    Exit Sub

ConnectionError:

    MsgBox "There was an error connecting to the database. " & Chr(13) _
        & Err.Number & ", " & Err.Description

End Sub
```

Using the `Set` statement, I'm able to assign the current Access ADO connection to my `Connection` object variable, which is called `localConnection`. Note that whenever you open a connection, it's important to utilize error handling.

Many ADO programming occasions involve connecting to a remote database. Connecting to a remote database through ADO involves working with one or more `Connection` object properties and its `Open` method, as demonstrated here.

```
Private Sub cmdConnectToRemoteDB_Click()

    On Error GoTo ConnectionError

    'Declare connection object variable
    Dim remoteConnection As New ADODB.Connection
```

```
'Assign OLEDB provider to the Provider property
'Use the Open method to establish a connection to the database
With remoteConnection
    .Provider = "Microsoft.ACE.OLEDB.12.0"
    .Open "C:\Home\Northwind 2007.accdb"
End With
```

```
MsgBox "Remote connection successfully established."
```

```
'Close the current database connection
remoteConnection.Close
```

```
Exit Sub
```

```
ConnectionError:
```

```
MsgBox "There was an error connecting to the database. " & Chr(13) _
    & Err.Number & ", " & Err.Description
```

```
End Sub
```



Remote database access using ADO can be one of two types: connecting to Access databases on your local machine or connecting to databases across the network.

Depending on the type of database you're connecting to, you use either ODBC or OLE DB as your connection provider. In the case of Microsoft Access databases, you assign an OLE DB provider name to the Connection object's Provider property, as seen next.

```
.Provider = "Microsoft.ACE.OLEDB.12.0"
```

For Access 2002–2003 databases, leverage the older OLEDB 4.0 Jet provider, as shown here.

```
.Provider = "Microsoft.Jet.OLEDB.4.0"
```

Once a provider has been set, use the Open method to establish a connection to your Access database. In the cmdConnectToRemoteDB\_Click() example, I pass a connection string to the Connection object's Open method. This connection string tells ADO what my database name is and where it is located.



When working with examples in this book, you need to change the string of the Connection object's Open method to reflect your database's name and location.

After working with the ADO object model for some time, you learn that there are many programming methods for accomplishing the same task. Some ADO programmers like to use the Connection object's Properties collection to assign name/value pairs of connection attributes. As an example, the next procedure uses this technique for connecting to a remote Access database.

```
Private Sub cmdConnectToRemoteDB_Click()

    On Error GoTo ConnectionError

    'Declare connection object variable
    Dim remoteConnection As New ADODB.Connection

    'Assign OLEDB providers
    'Assign database name / location to Data Source
    'Use the Open method to establish a connection to the database
    With remoteConnection
        .Provider = "Microsoft.Access.OLEDB.10.0"
        .Properties("Data Provider").Value = "Microsoft.ACE.OLEDB.12.0"
        .Properties("Data Source").Value = "C:\Home\Northwind 2007.accdb"
        .Open
    End With

    MsgBox "Connection successfully established."

    remoteConnection.Close

    Exit Sub

ConnectionError:

    MsgBox "There was an error connecting to the database. " & Chr(13) _
        & Err.Number & ", " & Err.Description

End Sub
```



A common problem in beginning ADO programming is troubleshooting connection errors. One frequent error is to overlook the path and filename passed to the `Data Source` property or `Open` method. Make sure these values match correctly with the location, name, and version of your database.

Regardless of your connection choice, you should always close your database connections using the `Connection` object's `Close` method. The ADO `Close` method frees application resources, but does not remove the object from memory. To remove objects from memory, set the object to `Nothing`.

In general, connections should be opened once when the application is first loaded (`Load` event, for example) and closed once when the application is closing (`Unload` event, for example).

## WORKING WITH RECORDSETS

The ADO programming model uses recordsets to work with rows in a database table. Using ADO recordsets, you can add, delete, and update information in database tables.

The `Recordset` object represents all rows in a table or all rows returned by a SQL query. The `Recordset` object, however, can refer to only a single row of data at time. Once a database connection has been established, `Recordset` objects can be opened in one of three ways:

- Using the `Open` method of the `Recordset` object.
- Using the `Execute` method of the `Command` object.
- Using the `Execute` method of the `Connection` object.

The most common way of opening recordsets is through the `Open` method of a `Recordset` object.

`Recordset` object variables are declared like any other variable—using the `ADODB` library:

```
Dim rsEmployees As New ADODB.Recordset
```

Once a `Recordset` object variable has been declared, you can use its `Open` method to open a recordset and navigate through the result set. The `Open` method takes five arguments:

```
rsEmployees.Open Source, ActiveConnection, CursorType, LockType, Options
```

Before moving further into recordsets, let's investigate the concept of database locks and cursors and how Microsoft ADO uses them in conjunction with result sets and the `Recordset` object.



A *result set* is the set of rows retrieved by a command or SQL query. In Microsoft ADO, recordsets are embodied in the `Recordset` object, which is used to manage result sets. In an abstract sense, however, the notion of a recordset is synonymous with a result set.

## Introduction to Database Locks

Whether or not your `Recordset` objects can update, add, or delete rows depends on your database lock type. Most RDBMSs implement various forms of table and row-level locking. Database locking prevents multiple users (or processes) from updating the same row at the same time. For example, suppose both my friend and I attempt to update the same row of information at the same time. Left to its own devices, this type of simultaneous updating could cause memory problems or data loss. To solve this, RDBMS developers designed sophisticated software-locking techniques using a variety of algorithms.

Even though the locking dilemma has been solved and implemented for us, an ADO developer, that's you, needs to identify a valid locking mechanism such as read-only, batch update, optimistic, or pessimistic. These types of locking mechanisms can be specified in the `LockType` property of the `Recordset` object. Table 10.2 describes available recordset lock types.

**TABLE 10.2 LOCKTYPE PROPERTY VALUES**

Lock Type	Description
<code>adLockBatchOptimistic</code>	Used for batch updates.
<code>adLockOptimistic</code>	Records are locked only when the <code>Recordset</code> object's <code>Update</code> method is called. Other users can access and update the same row of data while you have it open.
<code>adLockPessimistic</code>	Records are locked as soon as record editing begins. Other users can't access or modify the row of data until you have called the <code>Recordset</code> 's <code>Update</code> or <code>CancelUpdate</code> methods.
<code>adLockReadOnly</code>	Records are read-only (default lock type).

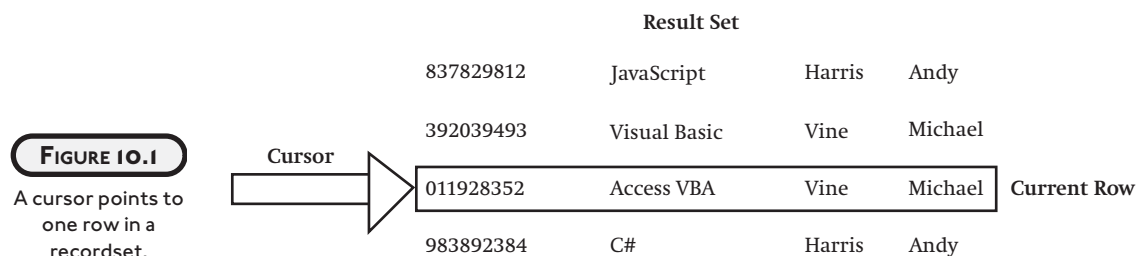
Generally speaking, most VBA programmers need to decide whether their database locks should be read-only or not. As a rule of thumb, use the read-only lock type (`adLockReadOnly`) when you simply need to scroll forward through a result set without modifying its contents. If you need to perform any updates on the result set, an optimistic locking solution (`adLockOptimistic`) is sufficient. Note that if a data provider cannot support the requested `LockType` setting, it will replace it with another type of locking.



## Introduction to Cursors

Since `Recordset` objects represent a single row of data, VBA programmers need a way to iterate through a list of rows. The ability to maneuver through a result set is implemented through database cursors.

In database terms, a *cursor* is a structure that names and manages a storage area in memory. Programmers use cursors to point to a row of data in a result set one row at a time. The concepts of a cursor and a result set are depicted in Figure 10.1.



Using a combination of other structures, such as loops and objects, programmers navigate through a recordset with the cursor pointing to the current row. You can think of programming with cursors as similar to file processing (discussed in Chapter 8, “Debugging, Input Validation, File Processing, and Error Handling”) where you open a data file and read one record at a time. When programming with cursors, you establish a cursor and move the cursor’s pointer to one row in a recordset at a time.

When working with ADO’s `Recordset` object, you can specify one of four cursor types in the `CursorType` property. Table 10.3 outlines this.

**TABLE 10.3** CURSORTYPE PROPERTY VALUES

Lock Type	Description
<code>adOpenForwardOnly</code>	Provides optimal performance through limited scrolling (forward only).
<code>adOpenKeyset</code>	Provides all types of movement in a <code>Recordset</code> object. Does not contain added or deleted rows.
<code>adOpenDynamic</code>	Provides all types of movement in a <code>Recordset</code> object. Includes added, deleted, and updated rows.
<code>adOpenStatic</code>	Provides all types of movement in a static <code>Recordset</code> object. Changes are not seen by users until the <code>Recordset</code> object is updated.

You should use forward-only (`adOpenForwardOnly`) cursors when updating rows is not required and reading rows in a result set from start to finish is acceptable. If you require dynamic updates in your result set, a dynamic cursor type (`adOpenDynamic`) is recommended.

In addition to cursor types, ADO allows programmers to specify a cursor location for the `Recordset` and `Connection` objects via the `CursorLocation` property. Depending on the location of your database and the size of your result set, cursor locations can have a considerable effect on your application's performance.

As outlined in Table 10.4, cursor locations can be either server side or client side.

**TABLE 10.4** CURSORLOCATION PROPERTY VALUES

Lock Type	Description
<code>adUseClient</code>	Records in the recordset are stored in local memory.
<code>adUseServer</code>	Builds a set of keys locally for Access databases and on the server for Microsoft SQL Server. The set of keys is used to retrieve and navigate through the result set.

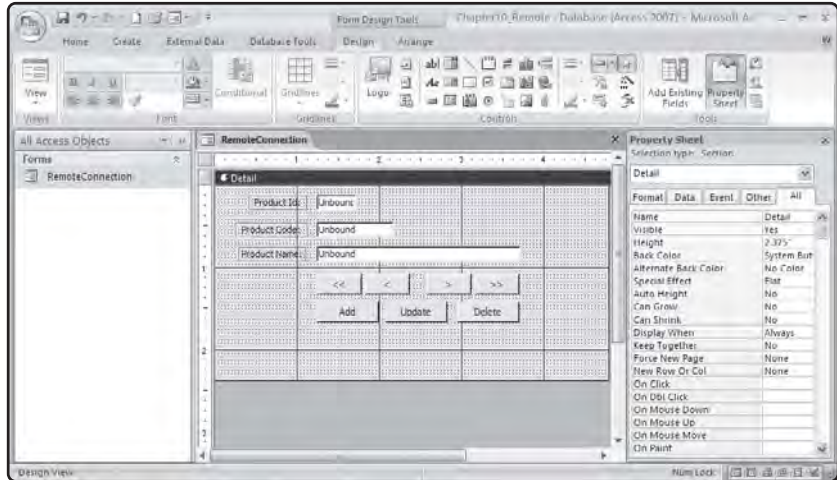
When considering cursor locations, Microsoft recommends server-side cursors when working with Microsoft Access databases and client-side cursors when working with Microsoft SQL Server databases.

## Retrieving and Browsing Data

After you have successfully established a connection to a database with ADO, retrieving and browsing data is quite easy. To retrieve data with ADO, you work with the `Recordset` object. In addition to cursors and locks, the `Recordset` object has many features for managing record-based data.

In this section, I use Microsoft's sample database `Northwind 2007.accdb` to demonstrate retrieving and browsing data with ADO. Figure 10.2 depicts a form I built, which remotely connects to the Access 2007 `Northwind` database and allows a user to browse through data found in the `Products` table.

To build the application seen in Figure 10.2, use the controls and properties in Table 10.5. I cover the VBA code for the `Add`, `Update`, and `Delete` command buttons later in the chapter.

**FIGURE 10.2**

Using ADO to connect and browse record-based data.

Next, I'll add the following VBA code to my form class module.

Option Compare Database

```
Dim remoteConnection As New ADODB.Connection
Dim rsProducts As New ADODB.Recordset
```

```
Private Sub Form_Load()
    Connect
    SetRecordset
End Sub
```

```
Private Sub Form_Unload(Cancel As Integer)
    Disconnect
End Sub
```

```
Public Sub Disconnect()

    On Error GoTo ConnectionError

    rsProducts.Close
    remoteConnection.Close
```

**TABLE 10.5 CONTROLS AND PROPERTIES OF THE REMOTE CONNECTION PROGRAM**

Control	Property	Property Value
Form	Name	Remote Connection
	Caption	Remote Connection
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No
Text Box	Name	txtProductId
Label	Name	lblProductId
	Caption	Product Id:
Text Box	Name	txtProductCode
Label	Name	lblProductCode
	Caption	Product Code:
Text Box	Name	txtProductName
Label	Name	lblProductName
	Caption	Product Name:
Command Button	Name	cmdMoveFirst
	Caption	<<
Command Button	Name	cmdMovePrevious
	Caption	<
Command Button	Name	cmdMoveNext
	Caption	>
Command Button	Name	cmdMoveLast
	Caption	>>
Command Button	Name	cmdAdd
	Caption	Add
Command Button	Name	cmdUpdate
	Caption	Update
Command Button	Name	cmdDelete
	Caption	Delete

```
Exit Sub
```

```
ConnectionError:
```

```
    MsgBox "There was an error closing the database." & _  
        Err.Number & ", " & Err.Description
```

```
End Sub
```

```
Private Sub Connect()
```

```
    On Error GoTo ConnectionError
```

```
    With remoteConnection
```

```
        .Provider = "Microsoft.ACE.OLEDB.12.0"
```

```
        .Open "C:\Home\Northwind 2007.accdb"
```

```
    End With
```

```
Exit Sub
```

```
ConnectionError:
```

```
    MsgBox "There was an error connecting to the database. " & _  
        Chr(13) & Err.Number & ", " & Err.Description
```

```
End Sub
```

---

```
Public Sub SetRecordset()
```

```
    Dim sql As String
```

```
    On Error GoTo DbError
```

```
    sql = "select * from Products"
```

```
    rsProducts.CursorType = adOpenKeyset
```

```
    rsProducts.LockType = adLockReadOnly
```

```
    rsProducts.Open sql, remoteConnection, _  
        , , adCmdText
```

```
    If rsProducts.EOF = False Then
```

```
        'Using three different techniques to access items in a recordset
```

```
        Me.txtProductId = rsProducts!ID
```

```
        Me.txtProductCode = rsProducts.Fields.Item("Product Code")
```

```
        Me.txtProductName = rsProducts.Fields.Item(3)
```

```
End If
```

```
Exit Sub
```

```
DbError:
```

```
MsgBox "There was an error retrieving information " & _  
    "from the database." _  
    & Err.Number & ", " & Err.Description
```

```
End Sub
```

I've created two procedures that handle connecting to the Northwind database and disconnecting from it. The `Connect` subprocedure is called during the form `Load` event, and the `Disconnect` subprocedure is called during the form `Unload` event, which is triggered when the form is closed or unloaded from memory.



When applying code from this book in your own applications, remember to change the path of the `Northwind 2007.accdb` database (or any other database, for that matter) in the `Open` method of the `Connection` object.

Notice in the form's `Load` event that I call another subprocedure: `SetRecordset`. This procedure sets up my `Recordset` object by establishing a SQL query, opening the recordset, and applying the first row of data to the text boxes.

Keep in mind that I declared my `Recordset` object variable (`rsProducts`) as a form-level variable. This allows me to access it throughout my form class module.

Before going any further, consider the following numbered list, which describes a common process for opening a recordset with ADO:

1. Define a SQL query using a `String` variable, which tells the `Recordset` object how it should be opened.
2. Assign cursor and lock type values to corresponding `Recordset` object properties. Note that you can set these properties on separate lines (as I did) or in the `Open` method of the `Recordset` object.
3. Use the `Open` method and pass in three of five optional parameters. The first parameter (`sql`) tells the `Recordset` object how to open the recordset. In addition to SQL statements, you can use a table name surrounded by double quotes ("`Products`", for example). The second parameter (`remoteConnection`) is the name of the active connection variable that

points to my copy of the Northwind 2997.accdb database. Note: This connection must have already been successfully opened. The last parameter is in the fifth parameter position. This is the options parameter, which tells the Recordset object how to use its first parameter. If the first parameter is a SQL string, use the constant `adCmdText`. If the first parameter is a table name ("Products", for instance), use the constant `adCmdTable`.

4. After the Recordset object has been successfully opened, I ensure that rows have been returned by using the EOF (end of file) property. If the EOF property is true, no records were returned and no further processing is done.
5. If the Recordset object contains one or more rows (`EOF = False`), I can access fields in a couple of ways. In my example, I use the Recordset object name (`rsProducts`) followed by an exclamation mark (!) and then the field name found in the database table. This is probably the most common way for ADO programmers to access recordset fields. Other ways of accessing fields involve using the Fields collection shown here.

```
Me.txtProductCode = rsProducts.Fields.Item("Product Code")  
Me.txtProductName = rsProducts.Fields.Item(3)
```

The first example passes a column name to the `Item` property of the Fields collection. The second example uses what's known as the *ordinal position* of the field returned. Ordinal positions start with 0.

Once data is successfully retrieved from a field, you can assign its value to variables or properties.

To browse through rows in a recordset, ADO's Recordset object provides the following four methods:

- `MoveFirst`. Moves the cursor to the first record in the result set.
- `MoveLast`. Moves the cursor to the last record in the result set.
- `MoveNext`. Moves the cursor to the next record in the result set.
- `MovePrevious`. Moves the cursor to the previous record in the result set.

When working with the `MoveNext` and `MovePrevious` methods, it's important to use cursors that allow forward and backward scrolling. Also, you need to check that the cursor's position is not already at the beginning of the recordset before moving to a previous entry or at the end of the recordset before moving to the next entry. Use the Recordset object's `AbsolutePosition` and `RecordCount` properties for these conditions.

## WHAT IS ORDINAL POSITION?

In ADO/database terms, the *ordinal position* refers to the relative position of a field or column in a collection such as `Fields`. Believe it or not, using ordinal positions for accessing fields is not uncommon in ADO. Consider an example of accessing the return value of the SQL function `Count`. Since SQL does not return a column name, you must work with ordinal position in the `Fields` collection.

Here is an example of using ordinal positions to retrieve the result of a SQL function.

```
Private Sub cmdCount_Click()

    Dim sql As String
    Dim rsCount As New ADODB.Recordset

    On Error GoTo DbError

    sql = "select count(*) from Products"

    rsCount.Open sql, remoteConnection, adOpenForwardOnly, _
        adLockReadOnly, adCmdText

    If rsProducts.EOF = False Then
        MsgBox "There are " & rsCount.Fields.Item(0) & _
            " rows in the Products table."
    End If

    Exit Sub

DbError:

    MsgBox "There was an error retrieving information from the database." _
        & Err.Number & ", " & Err.Description

End Sub
```



The `AbsolutePosition` property contains the ordinal position of the current record in the result set. The `AbsolutePosition` property contains a whole number beginning at 1. The `RecordCount` property contains the total number of rows contained in the recordset.

Using these properties and methods, you can build conditions for browsing through ADO records. To demonstrate, I continue my Remote Connection program by adding VBA/ADO code to the following four event procedures.

```
Private Sub cmdMoveFirst_Click()
```

```
    On Error GoTo DbError
```

```
    'Move to the first record in the result set.
```

```
    rsProducts.MoveFirst
```

```
    Me.txtProductId = rsProducts!ID
```

```
    Me.txtProductCode = rsProducts.Fields.Item("Product Code")
```

```
    Me.txtProductName = rsProducts.Fields.Item("Product Name")
```

```
Exit Sub
```

```
DbError:
```

```
    MsgBox "There was an error retrieving information " & _  
        "from the database." _  
        & Err.Number & ", " & Err.Description
```

```
End Sub
```

---

```
Private Sub cmdMoveLast_Click()
```

```
    On Error GoTo DbError
```

```
    'Move to the last record in the result set.
```

```
    rsProducts.MoveLast
```

```
    Me.txtProductId = rsProducts!ID
```

```
    Me.txtProductCode = rsProducts.Fields.Item("Product Code")
```

```
    Me.txtProductName = rsProducts.Fields.Item("Product Name")
```

Exit Sub

DbError:

```
MsgBox "There was an error retrieving information " & _  
    "from the database." &  
    & Err.Number & ", " & Err.Description
```

End Sub

---

Private Sub cmdMoveNext\_Click()

On Error GoTo DbError

'Move to the next record in the result set if the cursor is not  
'already at the last record.

```
If rsProducts.AbsolutePosition < _  
    rsProducts.RecordCount Then
```

```
    rsProducts.MoveNext
```

```
    Me.txtProductId = rsProducts!ID
```

```
    Me.txtProductCode = rsProducts.Fields.Item("Product Code")
```

```
    Me.txtProductName = rsProducts.Fields.Item("Product Name")
```

End If

Exit Sub

DbError:

```
MsgBox "There was an error retrieving information " & _  
    "from the database." &  
    & Err.Number & ", " & Err.Description
```

End Sub

---

```
Private Sub cmdMovePrevious_Click()

    On Error GoTo DbError

    'Move to the previous record in the result set, if the
    'current record is not the first record.
    If rsProducts.AbsolutePosition > 1 Then

        rsProducts.MovePrevious
        Me.txtProductId = rsProducts!ID
        Me.txtProductCode = rsProducts.Fields.Item("Product Code")
        Me.txtProductName = rsProducts.Fields.Item("Product Name")

    End If

    Exit Sub

DbError:

    MsgBox "There was an error retrieving information " & _
        "from the database." _
        & Err.Number & ", " & Err.Description

End Sub
```

## Updating Records

Updating records using ADO's `Recordset` object is relatively easy. Generally speaking, you perform the following tasks:

1. Declare a new `Recordset` object variable.
2. Define and create a SQL string that identifies the record you want to update.
3. Assign updatable cursor and lock types for updating a record.
4. Open the recordset, which should contain only one record, the record you wish to update.
5. Assign new data to the recordset fields.
6. Update the recordset using the `Recordset` object's `Update` method.

7. Close the recordset using the Recordset object's Close method.
8. Refresh other recordsets, if applicable, by closing and reopening the recordset or calling the Recordset object's Requery method.

The tricky part in updating records is ensuring that your SQL queries are well defined. For example, to update a record in the Products table of the Northwind database, I want to qualify my recordset using the table's primary key. In this case, that's the ID field of the Products table.

```
sql = "select * from Products where ID = " & _  
    Val(Me.txtProductId.Value)
```

In the preceding SQL string, I assign the value of the text box containing the product ID. Since I'm using a String variable, I can build a dynamic SQL statement using control properties—input from the user. By using a condition in my SQL string and supplying it with the primary key of a record, I'm making sure that only the record with the primary key is contained in the result set.



When concatenating string or text values to a dynamic SQL statement, you must use single quotes inside of double quotes to surround the expression.

```
sql = "select [Product Code] from Products where [Product Name] = '" & _  
    Me.txtProductName.Value & "'"
```

You may be asking yourself, “How do I know what record to update?” The answer to this question is based on the record selected in the graphical interface. As long as your GUI allows a user to select only one record, you are in good shape. To demonstrate, I now add VBA/ADO code to the Update command button's Click event in the Remote Connection program.

```
Private Sub cmdUpdate_Click()  
  
    Dim sql As String  
    Dim rsUpdate As New ADODB.Recordset  
  
    On Error GoTo DbError  
  
    'Build dynamic SQL statement based on record  
    'selected by the user.  
    sql = "select * from Products where ID = " & _  
        Val(Me.txtProductId.Value)
```

```
'Assign updatable cursor and lock type properties.
rsUpdate.CursorType = adOpenDynamic
rsUpdate.LockType = adLockOptimistic

'Open the Recordset object.
rsUpdate.Open sql, remoteConnection, , , adCmdText

'Don't try to update the record, if the recordset
'did not find a row.
If rsUpdate.EOF = False Then

    'Update the record based on input from the user.
    With rsUpdate
        .Fields.Item("Product Code") = Me.txtProductCode
        .Fields.Item("Product Name") = Me.txtProductName
        .Update
        .Close
    End With

End If

MsgBox "Record updated.", vbInformation

'Close the form-level Recordset object and
'refresh it to include the newly updated row.
rsProducts.Close
SetRecordset

Exit Sub

DbError:

MsgBox "There was an error updating the database." _
    & Err.Number & ", " & Err.Description

End Sub
```



The Recordset object's Update method is synonymous with saving.

## Adding Records

Adding records with ADO does not necessarily require the use of SQL queries. In most scenarios, you simply need a record added to a table based on user input. In the simplest form, records are added to tables using the following steps:

1. Declare a new Recordset object variable.
2. Assign updatable cursor and lock types for adding a record.
3. Open the Recordset object using its Open method with a table name as the first parameter and the associated adCmdTable constant name for the options parameter.
4. Call the Recordset object's AddNew method.
5. Assign new data to the recordset fields.
6. Save the new row of data using the Recordset object's Update method.
7. Close the recordset using the Recordset object's Close method.
8. Refresh other recordsets, if applicable, by closing and reopening the recordset or calling the Recordset object's Requery method.

Using the preceding steps, the following code implements the Click event procedure of the Add command button from Figure 10.2.

```
Private Sub cmdAdd_Click()

    Dim sql As String
    Dim rsAdd As New ADODB.Recordset

    On Error GoTo DbError

    'Assign updatable cursor and lock type properties.
    rsAdd.CursorType = adOpenDynamic
    rsAdd.LockType = adLockOptimistic

    'Open the Recordset object.
    rsAdd.Open "Products", remoteConnection, , , adCmdTable

    'Add the record based on input from the user
    '(except for the AutoNumber primary key field).
    With rsAdd
```

```

        .AddNew
        .Fields.Item("Product Code") = Me.txtProductCode
        .Fields.Item("Product Name") = Me.txtProductName
        .Update
        .Close
    End With

```

```

MsgBox "Record Added.", vbInformation

```

```

'Close the form-level Recordset object and refresh
'it to include the newly updated row.
rsProducts.Close
SetRecordset

```

```

Exit Sub

```

```

DbError:

```

```

MsgBox "There was an error adding the record." _
    & Err.Number & ", " & Err.Description

```

```

End Sub

```

## Deleting Records

Deleting records using ADO is somewhat similar to updating records in that you need to use SQL queries to identify the record for updating—in this case, deleting. The numbered steps identify a typical ADO algorithm for deleting a record:

1. Declare a new `Recordset` object variable.
2. Assign updatable cursor and lock types for deleting a record.
3. Construct a dynamic SQL string that uses a condition to retrieve the record selected by the user. The condition should use a field, which is a key (unique) value selected by the user.
4. Open the recordset, which should contain only one record (the record you wish to delete).
5. If the record was found, call the `Recordset` object's `Delete` method.
6. Save the record operation using the `Recordset` object's `Update` method.
7. Close the recordset using the `Recordset` object's `Close` method.
8. Refresh other recordsets, if applicable, by closing and reopening the recordset or calling the `Recordset` object's `Requery` method.

Using these steps, I can implement ADO program code in the Click event procedure of the Delete command button shown in Figure 10.2.

```
Private Sub cmdDelete_Click()

    Dim sql As String
    Dim rsDelete As New ADODB.Recordset

    On Error GoTo DbError

    'Build dynamic SQL statement based on
    'record selected by the user.
    sql = "select * from Products where ID = " & _
        Val(Me.txtProductId.Value)

    'Assign updatable cursor and lock type properties.
    rsDelete.CursorType = adOpenDynamic
    rsDelete.LockType = adLockOptimistic

    'Open the Recordset object.
    rsDelete.Open sql, remoteConnection, , , adCmdText

    'Don't try to delete the record, if the
    'recordset did not find a row.
    If rsDelete.EOF = False Then

        'Update the record based on input from the user.
        With rsDelete
            .Delete
            .Update
            .Close
        End With

    End If

    MsgBox "Record deleted.", vbInformation

    'Close the form-level Recordset object and refresh
```



```
'it to include the newly updated row.
```

```
rsProducts.Close
```

```
SetRecordset
```

```
Exit Sub
```

```
DbError:
```

```
MsgBox "There was an error deleting the record." _
```

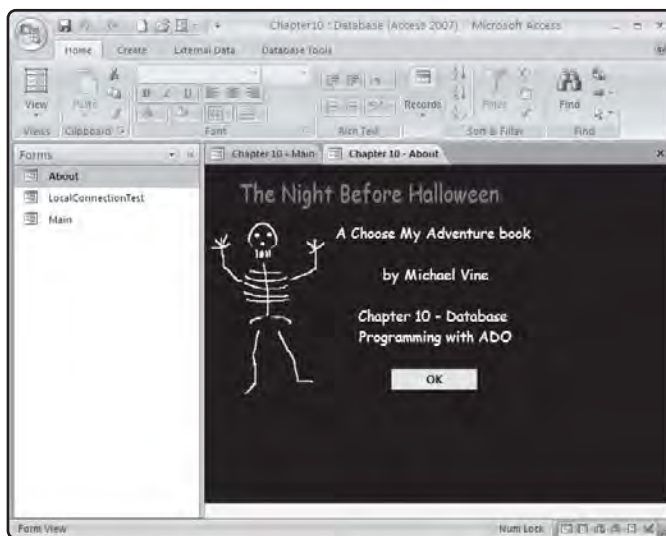
```
& Err.Number & ", " & Err.Description
```

```
End Sub
```

## CHAPTER PROGRAM: CHOOSE MY ADVENTURE

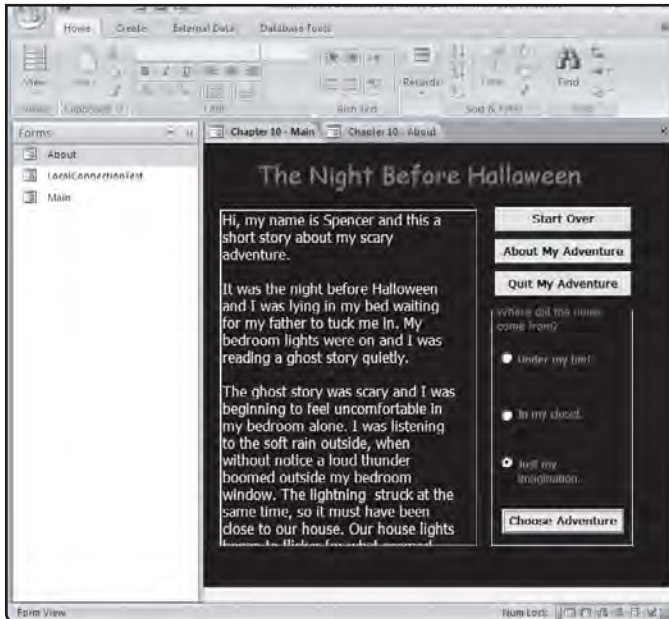
Without a doubt, Choose My Adventure is one of my favorite programs from the *For the Absolute Beginner* books. Choose My Adventure uses ADO programming techniques to access tables that contain a short story. The story presents questions on various pages and allows the reader to select an outcome. Depending on the selected outcome, the reader gets a different story and ending.

I used two forms to build Choose My Adventure—one to show About details (see Figure 10.3) and the other to house the program's main functionality (see Figure 10.4). Controls and properties of the Choose My Adventure About form are shown in Table 10.6.



**FIGURE 10.3**

The About form of the Choose My Adventure program.

**FIGURE 10.4**

Using chapter-based concepts to build the Choose My Adventure program.

**TABLE 10.6 CONTROLS AND PROPERTIES OF THE ABOUT FORM**

Control	Property	Property Value
Form	Name	About
	Caption	Chapter 10 - About
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No
Image	Name	imgBanner
	Picture	banner.jpg (located on companion website)
	Size Mode	Stretch
Image	Name	imgBanner
	Picture	skeleton.jpg (located on companion website)
	Size Mode	Stretch
Label	Name	lblAbout
	Caption	A Choose My Adventure book
Command Button	Name	cmdOK
	Caption	OK

There is minimal code required for the About form, which uses the DoCmd object's Close method to close the form.

```
Private Sub cmdOK_Click()
    DoCmd.Close acForm, "About"
End Sub
```

The core functionality of the Choose My Adventure program is contained in the Main form with controls and properties described in Table 10.7.

**TABLE 10.7 CONTROLS AND PROPERTIES OF THE MAIN FORM**

Control	Property	Property Value
Form	Name	Main
	Caption	Chapter 10 - Main
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No
Image	Name	imgBanner
	Picture	banner.jpg (located on companion website)
	Size Mode	Stretch
Text Box	Name	txtPage
	Enter Key Behavior	New Line in Field
Command Button	Name	cmdRead
	Caption	Start Over
Command Button	Name	cmdAbout
	Caption	About My Adventure
Command Button	Name	cmdQuit
	Caption	Quit My Adventure
Command Button	Name	cmdChoose
	Caption	Choose Adventure
Frame	Name	fraQuestionAndOutcomes
Frame Label	Name	lblQuestion
Option Button	Name	optOption1
Option Button Label	Name	lblOption1
Option Button	Name	optOption2
Option Button Label	Name	lblOption2
Option Button	Name	optOption3
Option Button Label	Name	lblOption3

All of the code required to implement the Choose My Adventure Main form follows.

```
Option Compare Database
Option Explicit
```

```
'Declare form-level Connection object variable.
Dim localConnection As ADODB.Connection
```

---

```
Private Sub cmdAbout_Click()
    'Show the About form.
    DoCmd.OpenForm "About"
End Sub
```

---

```
Private Sub cmdChoose_Click()
    ' Use the assigned option button value to get
    ' the next page in the book.
    GetPage Me.fraQuestionAndOutcomes.Value
End Sub
```

---

```
Private Sub cmdQuit_Click()
    'Quit the application.
    DoCmd.Quit
End Sub
```

---

```
Private Sub cmdRead_Click()

    MsgBox "Welcome to The Night Before Halloween, " & _
        "a Choose My Adventure book by Michael Vine.", _
        vbOKOnly, "Chapter 10 - Database Programming with ADO"

    Me.cmdRead.Caption = "Start Over"
    ResetForm
    Me.txtPage.Value = ""
    Me.lblQuestion.Caption = ""
    Me.cmdChoose.Visible = False
```

```
'Call the GetPage procedure to display the  
'first page in the book.  
GetPage 1
```

```
End Sub
```

---

```
Private Sub Form_Load()  
  
    'Perform some initial setup.  
    ResetForm  
    Me.txtPage.Value = ""  
    Me.lblQuestion.Caption = ""  
    Me.cmdChoose.Visible = False  
    Me.cmdRead.Caption = "Read My Adventure"  
  
    'Assign the current Access connection to my  
    'Connection object variable.  
    Set localConnection = CurrentProject.AccessConnection  
  
End Sub
```

---

```
Private Sub Form_Unload(Cancel As Integer)  
  
    On Error GoTo ErrorClosing  
  
    'Close the connection.  
    localConnection.Close  
  
    Exit Sub  
  
ErrorClosing:  
    'Do nothing!  
  
End Sub
```

---

```
Public Sub GetQuestion(pageID As Integer)
```

```
Dim rsQuestion As New ADODB.Recordset
Dim sql As String

On Error GoTo BookError

'Using the incoming pageID, get and display
'the associated question (if one exists).
'This procedure calls the GetOutcome procedure.
sql = "select * from Questions where PageID = " & pageID

rsQuestion.Open sql, localConnection, adOpenForwardOnly, _
    adLockReadOnly, adCmdText

If rsQuestion.EOF = False Then

    Me.lblQuestion.Caption = rsQuestion!Question
    GetOutcome rsQuestion!questionID

Else

    ResetForm

End If

rsQuestion.Close

Exit Sub

BookError:

    ErrorMessage

End Sub
```

---

```
Public Sub GetOutcome(questionID As Integer)
```

```
Dim rsOutcomes As New ADODB.Recordset
Dim x As Integer

Dim sql As String

On Error GoTo BookError

'Using the incoming questionID, get all possible outcomes
'for the associated question.
sql = "select * from Outcomes where QuestionID = " & _
    questionID

rsOutcomes.Open sql, localConnection, adOpenForwardOnly, _
    adLockReadOnly, adCmdText

ResetForm

If rsOutcomes.EOF = False Then

    Me.lblOption1.Visible = True
    Me.optOption1.Visible = True
    Me.lblOption1.Caption = rsOutcomes!Outcome
    Me.optOption1.OptionValue = rsOutcomes!GoToPage
    rsOutcomes.MoveNext

End If

If rsOutcomes.EOF = False Then
    Me.lblOption2.Visible = True
    Me.optOption2.Visible = True
    Me.lblOption2.Caption = rsOutcomes!Outcome
    Me.optOption2.OptionValue = rsOutcomes!GoToPage
    rsOutcomes.MoveNext

End If

If rsOutcomes.EOF = False Then
```

```
Me.lblOption3.Visible = True
Me.optOption3.Visible = True
Me.lblOption3.Caption = rsOutcomes!Outcome
Me.optOption3.OptionValue = rsOutcomes!GoToPage
```

```
End If
```

```
rsOutcomes.Close
```

```
Exit Sub
```

```
BookError:
```

```
ErrorMessage
```

```
End Sub
```

---

```
Public Function AnyMoreQuestions(pageID As Integer) As Boolean
```

```
Dim rsAnyMoreQuestions As New ADODB.Recordset
Dim returnValue As Boolean
Dim sql As String
```

```
On Error GoTo BookError
```

```
'This procedure is called by the GetPage procedure.
```

```
'It checks to see if there are any more questions
```

```
'for the current page passed in.
```

```
sql = "select * from Questions where PageID = " & pageID
```

```
rsAnyMoreQuestions.Open sql, localConnection, _
    adOpenForwardOnly, adLockReadOnly, adCmdText
```

```
If rsAnyMoreQuestions.EOF = False Then
```

```
    returnValue = True 'There are questions
```

```
Else
```



```
        returnValue = False 'There are no questions  
End If
```

```
rsAnyMoreQuestions.Close
```

```
AnyMoreQuestions = returnValue
```

```
Exit Function
```

```
BookError:
```

```
    ErrorMessage
```

```
End Function
```

---

```
Public Sub GetPage(pageID As Integer)
```

```
    Dim rsPage As New ADODB.Recordset
```

```
    Dim sql As String
```

```
    On Error GoTo BookError
```

```
    'Gets and displays the requested page. Calls GetQuestion  
    'and AnyMoreQuestions procedures.
```

```
    sql = "select * from Pages where PageID = " & pageID
```

```
    rsPage.Open sql, localConnection, adOpenForwardOnly, _  
        adLockReadOnly, adCmdText
```

```
    If rsPage.EOF = False Then  
        Me.txtPage.Visible = True  
        Me.txtPage.Value = rsPage!Content  
    End If
```

```
    GetQuestion rsPage!pageID
```

```
    Me.txtPage.SetFocus
```

```
If AnyMoreQuestions(rsPage!pageID) = False Then
```

```
    Me.cmdChoose.Visible = False
```

```
    Me.lblQuestion.Caption = ""
```

```
Else
```

```
    Me.cmdChoose.Visible = True
```

```
End If
```

```
rsPage.Close
```

```
Exit Sub
```

```
BookError:
```

```
    ErrorMessage
```

```
End Sub
```

---

```
Public Sub ErrorMessage()
```

```
    'A general error bin called by each error handler
```

```
    'in the form class.
```

```
    MsgBox "There was an error reading the book. " & Chr(13) _  
        & Err.Number & ", " & Err.Description
```

```
End Sub
```

---

```
Public Sub ResetForm()
```

```
    Me.lblOption1.Visible = False
```

```
    Me.lblOption2.Visible = False
```

```
    Me.lblOption3.Visible = False
```

```
    Me.optOption1.Visible = False
```

```
Me.optOption2.Visible = False  
Me.optOption3.Visible = False
```

```
End Sub
```

## SUMMARY

- ADO is Microsoft's popular programming vehicle for managing data in databases, such as Microsoft Access and SQL Server, and non-Microsoft relational databases such as Oracle.
- ADO's application programming interface (API) is made up of many objects, such as `Connection` and `Recordset`, and collections such as the `Fields` collection.
- It is good programming practice to use error handling whenever accessing a database through ADO.
- Connections to databases are established through ADO's `Connection` object.
- A result set is the set of rows retrieved by a command or SQL query.
- The `Recordset` object is used to work with rows in a database table.
- Database locking prevents multiple users (or processes) from updating the same row at the same time.
- A cursor is a structure that names and manages a storage area in memory. Programmers use cursors to point to one row of data at a time in a result set.
- The `Recordset` object methods `MoveFirst`, `MoveLast`, `MoveNext`, and `MovePrevious` are used to browse through records in a result set.
- Use the `Recordset` object properties `AbsolutePosition` and `RecordCount` to determine whether the cursor position is at the end or beginning of a recordset.
- The `Recordset` object method `Update` is synonymous with saving a record.
- The `Recordset` object method `AddNew` is used to add a row to a table.
- The `Recordset` object method `Delete` is used to delete a row from a table.

## PROGRAMMING CHALLENGES

1. Create a new Microsoft Access database application. Use the ADO Connection object to connect to a remote Northwind 2007.accdb database. Display a successful confirmation in a message box. Use error handling to catch any connection errors.
2. Add controls to a form that mimics fields contained in Microsoft's Northwind Employees table. Use ADO programming techniques to retrieve and fully browse the Employees table.
3. Update your application from Challenge 2 to allow a user to update records in the Employees table.
4. Update your application from Challenge 2 to allow a user to add records to the Employees table.
5. Update your application from Challenge 2 to allow a user to delete records from the Employees table.
6. Build your own Choose My Adventure program with a unique story, questions, and outcomes.

*This page intentionally left blank*



## CHAPTER

# OBJECT-ORIENTED PROGRAMMING WITH ACCESS VBA

**H**ave you ever wondered how VBA objects, methods, properties, and collections are created? Well, this chapter shows you how to leverage the power of *object-oriented programming* (also known as *OOP*) in Access VBA to create your very own custom objects, methods, properties, and collections!

## INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is not a language unto itself, but rather a programming practice. OOP is seemingly easy at the surface, but it can be quite challenging to master. In fact, many programmers coming from the procedural world of languages, such as C or COBOL, find they need to make a paradigm shift in how they think about programming. Even programmers who work with object-based languages, such as VBA, find the same paradigm shift inevitable. The paradigm shift I refer to is that of relating data, structures, and business requirements to objects.

OOP contains five core concepts, which are objects, classes, encapsulation, inheritance, and polymorphism:

- **Objects** represent a real-world thing such as person, place, or thing. Objects have behaviors and attributes.
- **Classes** are the blueprint for objects. They define how objects behave and how they expose attributes.
- **Encapsulation** hides implementation details from a user.
- **Inheritance** allows one class to inherit the features of another class.
- **Polymorphism** allows a class to implement the same operation in a number of different ways.

Unfortunately, VBA does not support inheritance or polymorphism in OOP's truest sense. Nevertheless, object-oriented programming in VBA allows the implementation of one of the most important benefits of OOP development, known as encapsulation. In OOP terms, *encapsulation* allows programmers to reduce code complexity by hiding data and complex structures in classes. You and other programmers simply *instantiate* (create an object from a class) these classes as objects and access the object's methods and properties. Encapsulating implementation details is a wonderful benefit of OOP. Not only are complex details hidden, but code reuse is promoted. In VBA, OOP development is achieved through custom objects that are defined in class modules. Once built, custom objects don't necessarily add new functionality to your code. In fact, the same code you write in class modules could be written in event procedures, subprocedures, and function procedures. The purpose of using class modules is to provide encapsulation, code-reuse, and self-documenting code. Programmers using your custom objects work with them just as they would with other built-in VBA objects, such as the ones found in the ADO library (Connection and Recordset objects).

Development with OOP generally requires more planning up front than in other programming paradigms. This design phase is crucial to OOP and your system's success.

At the very minimum, OOP design includes the following tasks:

- Identify and map objects to programming and business requirements.
- Identify the *actions* (methods) and *attributes* (properties) of each object. This action is commonly referred to as *identifying the responsibilities* of each object.
- Identify the relationships between objects.
- Determine the scope of objects and their methods and properties.

## CREATING CUSTOM OBJECTS

You begin your investigation into object-oriented programming by creating custom objects that encapsulate implementation details. To create custom objects, VBA programmers use OOP techniques and class modules. You specially learn how to build class modules that

contain member variables and property and method procedures. After learning how to build custom objects with class modules, you see how to instantiate custom objects and access custom object methods and properties.

## Working with Class Modules

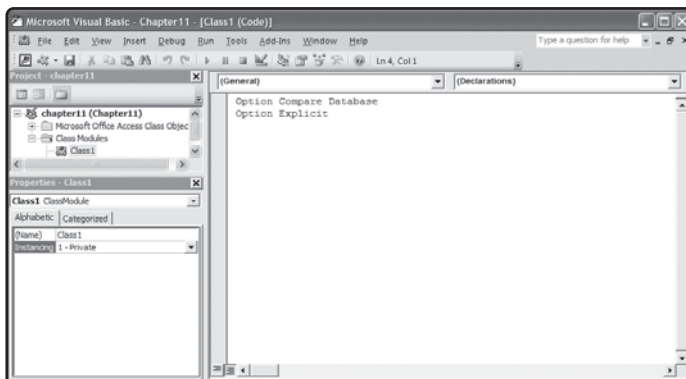
*Classes* are the blueprints for an object. They contain the implementation details, which are hidden from *users* (programmers who use your custom objects). In object-oriented programming with VBA, classes are implemented as class modules.

Class modules do not exist in memory. Rather, the instance of the class known as the object does. Multiple instances of a single class can be created. Each *instance* (object) created from a class shares the same access to the class's methods and properties. Even though multiple objects created from one class share the same characteristics, they are different in two ways. First, objects instantiated from the same class can have different property values. For example, an object called Bob instantiated from the Person class may have its `hairColor` property value set to brown, whereas an object called Sue, also instantiated from the Person class, could have its `hairColor` property value set to blonde. Second, objects instantiated from the same class have unique memory addresses.



In OOP terms, an *instance* refers to the object that was created from a class. The term *instantiate* means to create an object from a class.

To create a class module in Access VBA, simply open a Visual Basic window (VBE) and select the Class Module menu item from the Insert menu. Microsoft VBA automatically creates the class module for you, as depicted in Figure 11.1.



**FIGURE 11.1**

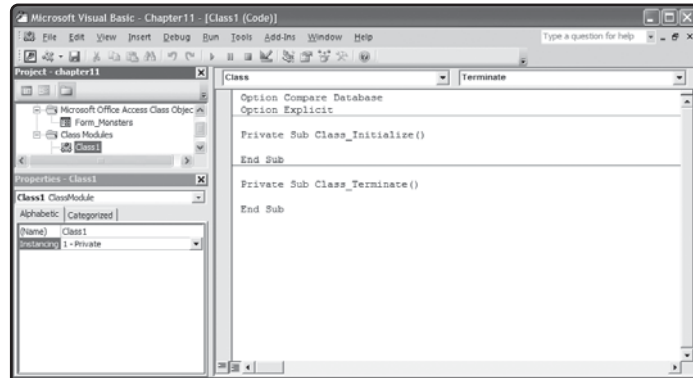
A newly created class module.



By default, VBA class modules contain two events called `Initialize` and `Terminate`. These events can be accessed through the Code window shown in Figure 11.2.

**FIGURE 11.2**

The `Initialize` and `Terminate` events are accessed through the VBE Code window.



The `Initialize` event for a class module is triggered each time the class is instantiated (created) using the `New` or `Set` keywords. The class module's `Initialize` event is similar to that of a constructor in OOP languages such as Java and C++. It is used to execute code when the object is first created. For example, you might want to initialize certain member variables each time an instance of your class is created.

The `Terminate` event is triggered each time the instance is removed from memory. You can place code in this event procedure to free up other objects from memory or finalize any necessary transactions.

Another common use of the `Initialize` and `Terminate` events is in debugging your applications. If you'd like to know each time your application creates and destroys one of your custom objects, simply use the `Initialize` and `Terminate` events, like I've done here.

```
Private Sub Class_Initialize()
```

```
    Debug.Print "Object created."
```

```
End Sub
```

---

```
Private Sub Class_Terminate()
```

```
    Debug.Print "Object destroyed."
```

```
End Sub
```



Microsoft recommends *not* using message boxes in the `Initialize` and `Terminate` events, which requires Windows messages to be processed.

## Property Procedures

VBA provides property procedures for managing the attributes of a class, which are exposed internally for the class to use or exposed externally as object properties. Simply put, properties are just variables. You could, of course, declare variables in your class modules for your procedures to use, but that would defeat the purpose of object-oriented programming.

To work with properties in VBA, you create variables of various scopes and use a combination of property procedures to manage them. VBA provides three types of property procedures:

- **Property Get.** Returns the value of a property.
- **Property Let.** Assigns a value to the property.
- **Property Set.** Sets the value of an object property.

**Property Get** procedures are often used in conjunction with both **Property Let** and **Property Set** procedures. When used together, a **Property Let** procedure with a **Property Get** procedure, or a **Property Set** procedure with a **Property Get** procedure must share the same name. **Property Let** and **Property Set** procedures, however, cannot be used together. They perform distinctly different roles in VBA object-oriented programming. Specifically, **Property Let** procedures are used for assigning data to scalar variables such as `String`, `Integer`, `Double`, or `Date` data types. **Property Set** procedures are used for assigning a reference to an object.

To add property procedures to your class module, select the **Add Procedure** dialog box from the **VBE Insert** menu to add property procedures, as demonstrated in Figure 11.3.



**FIGURE 11.3**

Use the **Add Procedure** dialog box to create property procedures.

VBA automatically adds a matching set of **Property Get** and **Property Let** procedures for you, as shown next:

```
Public Property Get Something() As Variant
```

```
End Property
```

---

```
Public Property Let Something(ByVal vNewValue As Variant)
```

```
End Property
```

The code required in each property procedure is short. You simply add a line to each respective procedure to assign a value and return a value.

Before adding code to your property procedures, you must first have a property (sometimes referred to as *member variables*) to manage. When working with property procedures, your properties are generally declared as `Private` in the general declarations area. By declaring the variable (property) in the general declarations area, you provide access to the property from any procedure in the class module. Declaring the variable (property) as `Private` provides encapsulation. Specifically, it forces your object's users to use the property procedures to access the member variable instead of accessing the member variable directly.

The concept of private properties and procedures is very important in OOP. Any procedure or property declared as `Private` is only accessible to the class module (not instances of your class module). Examine this concept further by studying the next block of VBA code.

```
Option Compare Database
```

```
Private privateSomething As Variant
```

---

```
Public Property Get something() As Variant
```

```
    something = privateSomething
```

```
End Property
```

---

```
Public Property Let something(ByVal vNewValue As Variant)
```

```
    privateSomething = vNewValue
```

```
End Property
```

Notice that the `Property Get` procedure behaves much like a `Function` procedure in that a value is assigned to the procedure's name. This type of assignment statement returns the property's value to the calling procedure. The `Property Let` procedure takes a single argument as a parameter and assigns its value to the `Private` property. This is how instantiated objects of this class access the `privateSomething` property without knowing how it's declared or what its name is.

The previous code blocks typify a *read/write property*. In other words, instantiated objects of this class can read this property and write data to it. It's common, however, to require *read-only properties* in OOP. To do so, simply use a single `Property Get` procedure by removing the corresponding `Property Let` procedure.

```
Option Compare Database
```

```
Private readOnlySomething As Variant
```

---

```
Public Property Get something() As Variant
```

```
    something = readOnlySomething
```

```
End Property
```

Using the `Private` property and a single `Property Get` procedure, instantiated objects of this class can only read the property.

You can also use the `Add Procedure` dialog box to create matching `Property Set` and `Property Get` procedures. After VBA has created the matching property procedures, simply change the keyword `Let` to `Set`.

```
Option Compare Database
```

```
Private employee As Employee
```

---

```
Public Property Get NewEmployee() As Variant
```

```
    NewEmployee = employee
```

```
End Property
```

---

```
Public Property Set NewEmployee(ByVal vNewValue As Employee)
```

```
    employee = vNewValue
```

```
End Property
```

Instead of a Variant data type (or any other data type, for that matter), the Property Set procedure called `NewEmployee` takes in a parameter of `Employee` type. The Property Set procedure then assigns the object reference from the argument to the property of the same object type. A matching Property Get procedure is used to return an object reference of the property.

You've probably noticed by now that these procedures are very simple. That's because they should be! The primary purpose of property procedures is to manage access to member variables. It may seem like overkill for what appears to be variable access, but in sections to come and through practice you see the power of property encapsulation through property procedures.

## Method Procedures

Method procedures expose methods internally to the class module or externally to an instance of the class. They are the meat and potatoes of object-oriented programming!

Creating methods for custom objects is quite easy. Simply create and place Sub or Function procedures in your class modules to represent methods.

Remember from earlier in the book that function procedures return a value and subprocedures do not.

An example of each type of method is shown next.

```
Option Compare Database
```

```
Private result1 As Integer
```

```
Private result2 As Integer
```

---

```
Public Sub AddTwoNumbers(num1 As Integer, num2 As Integer)
```

```
    result1 = num1 + num2
```

```
End Sub
```

---

```
Public Function MultiplyTwoNumbers(num1 As Integer, num2 As Integer) As Double

    MultiplyTwoNumbers = num1 * num2

End Function
```

The first method, `AddTwoNumbers`, takes two parameters and sets a property. If it's necessary for instances of this class to access this result, you should create a `Property Get` procedure that returns the value of the `result1` member variable.

The second method, `MultiplyTwoNumbers`, is a function that also takes two arguments and performs a simple calculation. The big difference is that this method is a function, which returns a value to the calling procedure by assigning a value to the method's name.

To get a better understanding of object methods, consider the `Connection` object from the ADO library. The `Connection` object has a method called `Open`. You and I both know that this method establishes a connection to a database. But do we know how that method is implemented? No, we don't. And believe it or not, that's a good thing. Think about all the programming that must be involved to implement the `Open` method of the `Connection` object. It's a sure bet that it contains complicated data structures and algorithms. This is encapsulation at its finest. Because the implementation detail is hidden, VBA programmers can simply call the method and pass it a few parameters to successfully open a database connection.

You can, of course, encapsulate the ADO library even further by writing your own classes to hide the dirty details of ADO programming. To demonstrate, imagine that a friend familiar with VBA but not with ADO asks for your expertise in developing database connectivity. You agree to help by creating a class that performs all details of ADO programming for connecting to a database. You start your program design by thinking about what would be easiest for your colleague to use. During design, you decide to create a new class called `DbConnection` that takes care of all facets of connecting to a database and providing connection objects. Your `DbConnection` class provides methods to connect and close the database connection and provides properties to access the ADO `Connection` object. After careful design, your class and its methods and properties look something like the following code.

```
Option Compare Database
```

```
Private cnn As New ADODB.Connection
```

---

```
Public Sub OpenConnection(dbPath_ As String)
```

```
On Error GoTo ConnectionError

'Assign OLEDB provider to the Provider property.
'Use the Open method to establish a connection to the database.
With cnn
    .Provider = "Microsoft.ACE.OLEDB.12.0"
    .Open dbPath_
End With

Exit Sub

ConnectionError:

MsgBox "There was an error connecting to the database. " & Chr(13) _
    & Err.Number & ", " & Err.Description

End Sub
```

---

```
Public Sub CloseConnection()

    On Error GoTo ConnectionError:

    'Close the database connection.
    cnn.Close

    Exit Sub

ConnectionError:

MsgBox "There was an error connecting to the database. " & Chr(13) _
    & Err.Number & ", " & Err.Description

End Sub
```

---

```
Public Property Get ConnectionObject () As Variant

    'Return an object reference of the Connection object.
    Set ConnectionObject = cnn

End Property
```

This simple class, which contains two methods (`CloseConnection` and `OpenConnection`) and one property (`ConnectionObject`), encapsulates the ADO programming required to manage a database connection. In the next section, you see how easy it is for your friend to use your class for managing a database connection.

## Creating and Working with New Instances

After you've created a new class module, it becomes an available object type for you to use when declaring variables. Using the `DbConnection` class from the preceding section, I can declare an object variable in a Form Class module of `DbConnection` type.

```
Private Sub Form_Load()  
  
    'Declare object variable as DbConnection type.  
    Dim db As New DbConnection  
  
    'Open the database connection.  
    db.OpenConnection ("C:\temp\myDatabase.accdb")  
  
End Sub
```

You can easily see how little code it takes to open a connection with the ADO programming encapsulated in the `DbConnection` class. Users of the `DbConnection` class need only know what methods and properties to utilize rather than concern themselves with the specific ADO implementation details.

Working with object methods and properties is pretty straightforward. If you've been working with VBA even a little, you've already had exposure to objects and their properties and methods. When object methods or properties return an object reference, you need to decide how the returned object reference is to be used. For example, the `DbConnection` class contains a Property Get procedure called `ConnectionObject` that returns a reference of the current ADO Connection object.

```
Public Property Get ConnectionObject() As Variant  
  
    'Return an object reference of the Connection object.  
    Set ConnectionObject= cnn  
  
End Property
```



This property procedure appears as a property of the object when an instance of the class is created. Because this property returns an object reference, I use a Set statement to retrieve and assign the object reference to another object.

```
Private Sub Form_Load()

    Dim db As New DbConnection
    Dim newConnection As New ADODB.Connection

    db.OpenConnection ("C:\temp\myDatabase.accdb")

    'Returns an object reference.
    Set newConnection = db.ConnectionObject

End Sub
```

Another example of using the DbConnection class's ConnectionObject property is to use it as an argument by passing it into methods for recordset processing. To demonstrate, I added a new method (function procedure) called ReturnAThing to my DbConnection class, which takes an ADO Connection object as an argument.

```
Public Function ReturnAThing(cnn_ As ADODB.Connection) As Variant

    Dim rs As New ADODB.Recordset
    Dim thing As Variant
    Dim sql As String

    On Error GoTo DbError

    'Generate SQL string.
    sql = "select thing from AThing"

    'Open the read only / forward only recordset using SQL and the
    'Connection object passed in.
    rs.Open sql, cnn_, adOpenForwardOnly, adLockReadOnly, _      adCmdText

    If rs.EOF = False Then
        thing = rs!thing
    End If
```

```
rs.Close

'Return the thing back to the calling procedure.
ReturnAThing = thing

Exit Function
```

```
DbError:
```

```
MsgBox "There was an error retrieving a thing from " & _
      " the database. " & Chr(13) _
      & Err.Number & ", " & Err.Description
```

```
End Function
```

I can now use this method and the `ConnectionObject` property in the form class module to return a thing.

```
Private Sub Form_Load()

    Dim db As New DbConnection
    Dim myThing As Variant
    Dim newConnection As New ADODB.connection

    db.OpenConnection ("C:\temp\myDatabase.accdb")

    myThing = db.ReturnAThing(db.ConnectionObject)

End Sub
```

Passing ADO Connection objects to methods allows me to be flexible in the type of connection (database) used in recordset processing. If users of my class are fluent in SQL or the database structure, I might add another parameter to the `ReturnAThing` method. This new parameter could be a SQL string or part of a SQL string, which allows users to define what they want from the database or from where they want it.

An important part of working with object instances is freeing and reclaiming resources when your objects are no longer required. When objects are instantiated, VBA reserves memory and resources for processing. To free these resources, simply set the object to `Nothing`.

```
Private Sub Form_Load()  
  
    Dim db As New DbConnection  
    Dim myThing As Variant  
    Dim newConnection As New ADODB.connection  
  
    db.OpenConnection ("C:\temp\myDatabase.mdb")  
  
    myThing = db.ReturnAThing(db.ConnectionObject)  
  
    'Reclaim object resources  
    Set db = Nothing  
  
End Sub
```

It's good programming practice to reclaim resources not only from custom objects, but also from built-in objects such as the ones found in the ADO library. If you neglect to free object resources, VBA does not remove them from memory until the application is terminated. If your application uses a lot of objects, this can certainly lead to performance problems.

## WORKING WITH COLLECTIONS

Collections are a data structure similar to arrays in that they allow you to refer to a grouping of items as one entity. Collections, however, provide an ordered means for grouping not only strings and numbers, but objects as well. As demonstrated next, the `Collection` object is used to create a collection data structure.

```
Dim myCollection As New Collection
```

Collections are popular data structures in object-oriented programming because they allow the grouping of objects using an ordered name/value pair. In fact, collections are objects themselves!

Items in a collection are referred to as *members*. All collection objects have one property and three methods for managing members, as described in Table 11.1.

**TABLE 11.1** COLLECTION OBJECT PROPERTIES AND METHODS

Type	Name	Description
Property	Count	Returns the number of members in the collection (beginning with 1).
Method	Add	Adds a member to the collection.
Method	Remove	Removes a member from the collection.
Method	Item	Returns a specific member in the collection.

## Adding Members to a Collection

Use the Add method of the Collection object to add members to a collection. The Add method takes four parameters:

object.Add item, key, before, after

- **item.** A required expression that identifies the member to be added.
- **key.** An optional expression (string-based) that uniquely identifies the member.
- **before.** An optional expression that adds the member before the member position identified.
- **after.** An optional expression that adds the member after the member position identified.



When adding a member to a collection, only the before or after parameter can be used, not both.

The following VBA code creates a new collection and adds three string-based members.

```
Dim myColors As New Collection
```

```
myColors.Add "red"
myColors.Add "white"
myColors.Add "blue"
```

As mentioned, collections are useful for grouping objects. The next VBA code creates three ADO Recordset objects and adds them to a Collection object.

```
Dim books As New ADODB.Recordset
Dim authors As New ADODB.Recordset
Dim publishers As New ADODB.Recordset
```

```
Dim myRecordsets As New Collection
```

```
myRecordsets.Add books  
myRecordsets.Add authors  
myRecordsets.Add publishers
```

By grouping objects in a collection, I can simplify code by accessing all objects through one Collection object.

## Removing Members from a Collection

Members are removed from a collection using the Collection object's Remove method. The Remove method takes a single parameter that identifies the index or key value of the member.

Removing a collection member using both the index value and key value is demonstrated here.

```
Dim myColors As New Collection
```

```
myColors.Add "red", "r"  
myColors.Add "white", "w"  
myColors.Add "blue", "b"
```

```
myColors.Remove 1 'using an index value  
myColors.Remove "w" 'using a key value
```

## Accessing a Member in a Collection

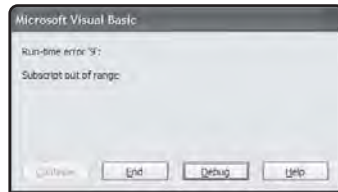
To access a member in a collection, use the Item method, which takes a single parameter that matches a member's index or key value.

```
Dim myColors As New Collection
```

```
myColors.Add "red", "r"  
myColors.Add "white", "w"  
myColors.Add "blue", "b"
```

```
MsgBox myColors.Item(1)  
MsgBox myColors.Item("b")  
MsgBox myColors.Item(4) 'Generates an error.
```

If the index or key value of the member is not found in the collection, an error like that in Figure 11.4 is generated.

**FIGURE 11.4**

An error is generated when trying to access a member's key value or index that does not exist.

## For Each Loops

VBA provides a looping structure specifically designed for iterating through members in a collection or an array. The `For Each` loop executes one or more statements inside the block as long as there is at least one member in the collection.

```
Dim myColors As New Collection
Dim vColor As Variant
```

```
myColors.Add "red", "r"
myColors.Add "white", "w"
myColors.Add "blue", "b"
```

```
For Each vColor In myColors
```

```
    MsgBox vColor
```

```
Next
```

Notice the syntax of the `For Each` statement. The statement basically says, “For every object in the collection, display the member name in a message box.”

**The variable used in the `For Each` statement (called `vColor` in the preceding example) is always of variant type regardless of the collection's content. This is an important note because `For Each` statements cannot be used with an array of user-defined types; variants can't contain a user-defined type.**

The next program code loops through all control names on a form using the built-in VBA Controls collection.

```
Dim myControls As New Collection
Dim vControl As Variant

For Each vControl In Form_Form1.Controls

    MsgBox vControl.Name

Next
```



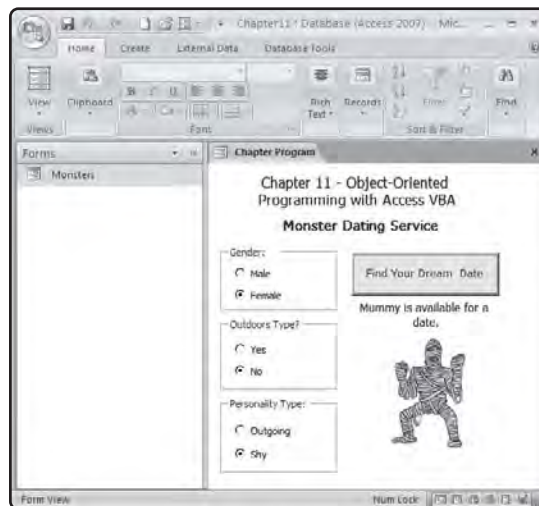
If you need to exit the For Each loop early, VBA provides the Exit For statement.

## CHAPTER PROGRAM: MONSTER DATING SERVICE

The Monster Dating Service program in Figure 11.5 uses chapter-based concepts to build a funny and simple application. Essentially, the program allows a user to find an available monster for a date by selecting character criteria.

**FIGURE 11.5**

Using chapter-based concepts to build the Monster Dating Service program.



Controls and properties of the Monster Dating Service program are shown in Table 11.2.

**TABLE 11.2 CONTROLS AND PROPERTIES OF  
THE MONSTER DATING SERVICE PROGRAM**

Control	Property	Property Value
Form	Name	Monsters
	Caption	Chapter Program
	Record Selectors	No
	Navigation Buttons	No
	Dividing Lines	No
Label	Name	lblHeading1
	Caption	Chapter 11--Object-Oriented Programming with Access VBA
Label	Name	lblHeading2
	Caption	Monster Dating Service
Frame	Name	fraGender
Frame Label	Name	lblGender
	Caption	Gender:
Option Button	Name	optMale
Option Button Label	Name	lblMale
	Caption	Male
Option Button	Name	optFemale
Option Button Label	Name	lblFemale
	Caption	Female
Frame	Name	fraOutdoors
Frame Label	Name	lblOutdoors
	Caption	Outdoors Type?
Option Button	Name	optYes
Option Button Label	Name	lblYes
	Caption	Yes
Option Button	Name	optNo
Option Button Label	Name	lblNo
	Caption	No
Frame	Name	fraPersonality
Frame Label	Name	lblPersonality
	Caption	Personality Type:
Option Button	Name	optOutgoing
Option Button Label	Name	lblOutgoing
	Caption	Outgoing
Option Button	Name	optShy
Option Button Label	Name	lblShy
	Caption	Shy



Control	Property	Property Value
Command Button	Name	cmdFindMonster
	Caption	Find Your Dream Date
Label	Name	lblMonsterName
	Caption	Mummy is available for a date.
Image	Name	imgPicture
	Picture	mummy.gif
	Size Mode	Stretch

The Monster Dating Service program uses object-oriented programming techniques split across two modules. The class module called `Monster` defines a monster object, which encapsulates all the functionality required to connect to the current database and retrieve monster attributes based on user input.

```
Option Compare Database
```

```
Option Explicit
```

```
Private name As String
```

```
Private picture As String
```

```
Private id As Integer
```

---

```
Public Sub FindMonster(sql_ As String)
```

```
    Dim rs As New ADODB.Recordset
```

```
    Dim sql As String
```

```
    'This method finds and sets all necessary monster details.
```

```
    'If does not return a value. Users of this method must use
```

```
    'the read-only property procedures to access the monster
```

```
    'attributes.
```

```
    On Error GoTo MonsterError
```

```
    'Open the recordset based on the SQL string passed
```

```
    'in as an argument.
```

```
    rs.Open sql_, CurrentProject.AccessConnection, _  
        adOpenForwardOnly, adLockReadOnly, adCmdText
```

```
If rs.EOF = False Then
```

```
    'Retrieve the monster's id, which will be used later.  
    id = rs!monsterId
```

```
Else
```

```
    'No monster found with those attributes.  
    'Raise a custom error.  
    Err.Raise vbObjectError + 512, , "No monster found."
```

```
End If
```

```
rs.Close
```

```
'Generate a new SQL string to retrieve the monster's  
'name and picture.  
sql = "select * from Monsters where MonsterId = " & id
```

```
rs.Open sql, CurrentProject.AccessConnection, _  
    adOpenForwardOnly, adLockReadOnly, adCmdText
```

```
If rs.EOF = False Then
```

```
    'Assign monster name and picture to properties.  
    name = rs!MonsterName  
    picture = Application.CurrentProject.Path & "\" & _  
        rs!picture
```

```
End If
```

```
rs.Close
```

```
Exit Sub
```

```
MonsterError:
```

```
MsgBox "Sorry, there was a problem finding the monster. " & _
```

```
Chr(13) & Err.Number & ", " & Err.Description
```

```
End Sub
```

---

```
Public Property Get MonsterName() As Variant
```

```
'This property procedure returns the monster's name.
```

```
MonsterName = name
```

```
End Property
```

---

```
Public Property Get MonsterPicture() As String
```

```
'This property procedure returns the path and file
```

```
'name of the monster's picture.
```

```
MonsterPicture = picture
```

```
End Property
```

**The form class module instantiates Monster objects to find an available monster for a date:**

```
Option Compare Database
```

```
Option Explicit
```

---

```
Private Sub cmdFindMonster_Click()
```

```
Dim aMonster As New Monster
```

```
Dim gender As String
```

```
Dim personality As String
```

```
Dim outdoors As Boolean
```

```
Dim sql As String
```

```
'Generate a SQL string based on user selection criteria.
```

```
If Me.fraGender.Value = 1 Then
```

```
    gender = "Male"
```

```
Else
```

```
    gender = "Female"
```

```

End If

If Me.fraOutdoors.Value = 1 Then
    outdoors = True
Else
    outdoors = False
End If

If Me.fraPersonality = 1 Then
    personality = "Outgoing"
Else
    personality = "Shy"
End If

sql = "select * from MonsterAttributes where Outdoors = " & _
      outdoors & " and Gender = '" & gender & "'" & _
      " and Personality = '" & personality & "'"

'Try to find a monster based on the search criteria.
aMonster.FindMonster sql

'If a monster was found, display their name and picture.
If aMonster.MonsterName = "" Then

    Me.lblMonsterName.Caption = _
        "Sorry, no one is available with " & _
        "that search criteria."

    Me.imgPicture.picture = _
        Application.CurrentProject.Path & "\" & "logo.gif"

Else

    Me.lblMonsterName.Caption = aMonster.MonsterName & _
        " is available for a date."

    Me.imgPicture.picture = aMonster.MonsterPicture

End If

End Sub

```


## SUMMARY

- Object-oriented programming maps data, structures, and business requirements to objects.
- Encapsulation allows programmers to reduce code complexity by hiding data and complex data structures in classes.
- Class modules contain member variables as well as property and method procedures.
- Class modules do not exist in memory.
- Multiple instances of a single class can be created.
- By default, VBA class modules contain two events called `Initialize` and `Terminate`.
- The `Initialize` event for a class module is triggered each time the class is instantiated (created) using the `New` or `Set` keywords.
- The `Terminate` event is triggered each time the class's instance is removed from memory.
- VBA provides property procedures for managing the attributes of a class.
- VBA provides three types of property procedures: `Property Get`, `Property Let`, and `Property Set`.
- `Property Get` procedures return the value of a property.
- `Property Let` procedures assign a value to a property.
- `Property Set` procedures set the value of an object property.
- Use a single `Property Get` procedure to create a read-only property.
- Method procedures are created in class modules with `Sub` and `Function` procedures.
- Setting objects to `Nothing` frees system resources.
- Collections are objects that contain an ordered list of items.
- Items in a collection are called members.
- Members in a collection can be referenced with an index or key value.
- VBA provides the `For Each` loop to iterate through members in a collection or an array.

## PROGRAMMING CHALLENGES

1. Create a new database called **BookStore** with one table called **Books**. Add the columns **ISBN**, **Title**, **PublishDate**, and **Price** to the **Books** table. Add a few records to the **Books** table. Create a new connection class called **CustomConnection** that connects to your **BookStore** database. The new class should have two methods—one method for opening an **ADO Connection** object and a second method for closing the **ADO Connection** object. The method that opens a database connection should take a single string argument, which represents the path and filename of the database.
2. In the same database application from Challenge 1, create a new class called **Books**. This class should have a read-only property for each column in the **Books** table. Create a method in the **Books** class called **FindBook**. The **FindBook** method should take in an **ISBN**. Build a **SQL** string based on the **ISBN** and use **ADO** programming techniques to open a recordset and assign the recordset field values to the class's matching properties. You should use the **CustomConnection** class to create and retrieve any **Connection** objects.
3. Enhance Challenges 1 and 2 by building a form with controls that allow a user to find a book by entering an **ISBN**. Use your **Books** class from Challenge 2 to find and retrieve book details.
4. Enhance the user interface from Challenge 3 to allow a user to add and remove books. To accomplish this, you need to modify the **Books** class from Challenge 2 by adding two methods called **AddBook** and **RemoveBook**.
5. In a new Access application, create a **Collection** object called **Friends**. Construct a user interface that allows a user to add and remove names of friends in the **Friends** collection.
6. Add a command button to Challenge 5's user interface that displays each friend in a message box. Hint: Use the **For Each** loop to iterate through members in the **Friends** collection.

*This page intentionally left blank*



## CHAPTER

# MACROS AND PERFORMANCE TUNING

**I**n this chapter, you learn about Macro objects in Access 2007, including how to create and debug stand-alone Macros and converting Macros to VBA code. You'll also learn about Form, VBA code, Query and Index performance considerations to build a well-tuned Access database application. I will also show you how to leverage out-of-the-box database optimization hints through Access's Performance Analyzer.

## MACROS

Macros are Access objects that you build and assign to Form, Report, or Control events to automate tasks and processes without VBA programming. Macros hide the implementation details of the VBA programming language much like a class definition encapsulates the complexities of an object's functionality. For example, you can create a Macro to open a Query when a user clicks a Command Button on a Form by associating the Macro name to the Command Button's `On Click` event property.

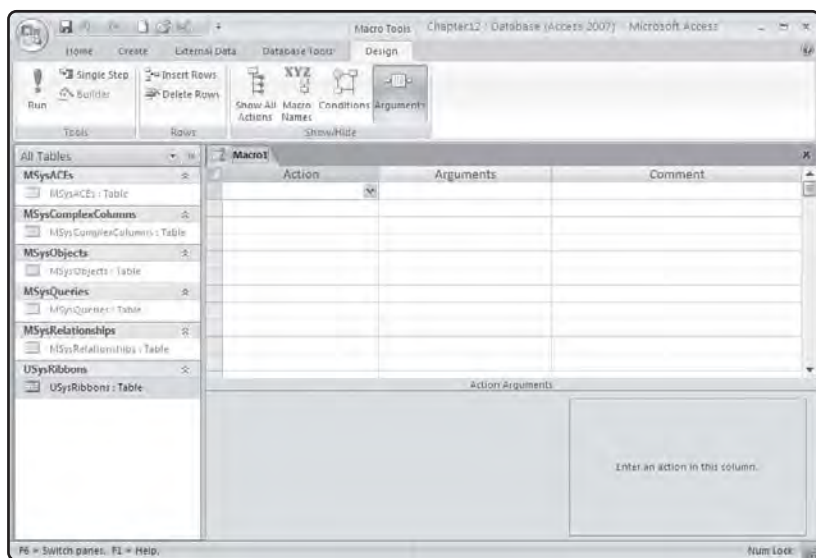
Access 2007 has added and enhanced many Macro features, including:



- **Embedded Macros** are neither stored nor accessible in the Access Navigation Pane, but rather are stored in the event properties of Form, Report, or Controls.
- **Increased security** options can be set in the Macro Builder to prevent non-trusted Macro options from running.
- **Error handling and debugging** which include new Macro Actions to handle errors and the ability to step through Macro Actions one at a time.
- **Temporary variables** can now be created and used in a Macro's conditional expression and passed among Forms and Reports. A Macro's temporary variables can also be accessed in VBA.

## Stand-Alone Macros

Stand-alone Macros (a single Macro or a Macro group) can be created using the Macro Builder by selecting the Macro icon from the Other group of the Create tab. Once created, Access 2007 displays an empty Macro in Design view, as revealed in Figure 12.1.



**FIGURE 12.1**

A newly created Macro using the Macro Builder.

To create your Macro, select one or more Macro Actions, as seen in Figure 12.2, and enter values for each Action's Arguments. Referenced in Table 12.1, Macro Actions contain common Access database processes that build automation.

**TABLE 12.1 ACCESS 2007 MACRO ACTIONS**

Action	Short Description
AddMenu	Used to create menus
ApplyFilter	Apply a filter or query to a table, form, or report
Beep	Sounds a beep tone through the computer's speaker
CancelEvent	Cancels the event that caused the Macro to run
ClearMacroError	Clears information about an error stored in the MacroError object
Close	Closes a document tab
CloseDatabase	Closes the current database
FindNext	Finds the next record based on the FindRecord action
FindRecord	Finds the first record based on specified criteria
GoToControl	Moves focus to a specified field or control
GoToPage	Moves focus to the first control on a specified page
GoToRecord	Moves the specified record to the first record
Hourglass	Changes the mouse pointer to a picture of an hourglass
LockNavigationPane	Prevents database objects from being deleted in the Navigation Pane
Maximize	Enlarges the active window to fill the Access window
Minimize	Reduces the active window
MoveSize	Moves or resizes the active window
MsgBox	Displays a warning or informational message box
NavigateTo	Manages the display of database objects in the Navigation Pane
OnError	Specifies an action to happen on error occurrence
OpenForm	Opens a form in Form, Design, Print, or Datasheet view
OpenQuery	Opens a select or crosstab query in Datasheet, Design, or Print view
OpenReport	Opens a report in either Design or Print view
OpenTable	Opens a table in Design, Datasheet, or Print view
OpenView	Opens a view in Design, Datasheet, or Print view
OutputTo	Outputs data to several formats
Quit	Exits Microsoft Access 2007
RemoveAllTempVars	Removes all temporary variables
RemoveTempVar	Removes a single temporary variable
Rename	Renames a specified database object
RepaintObject	Completes pending screen updates for a specified object
Requery	Updates data for a specified control
Restore	Restores a window to its previous size
RunCode	Calls a VBA function procedure
RunCommand	Executes a built-in Access 2007 command
RunMacro	Runs a Macro
SearchForRecord	Searches for a record in a table, form, report, or query
SelectObject	Selects a specified database object
SendObject	Emails a specified Access 2007 object

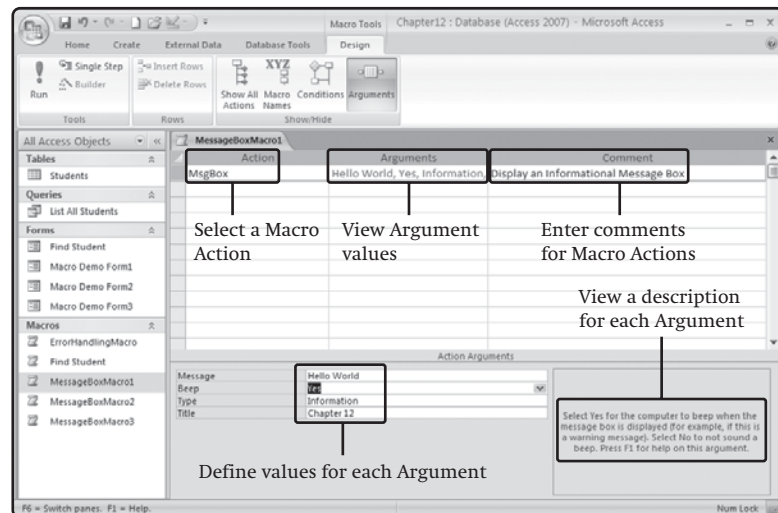
**Action**

SetDisplayedCategories  
 SetMenuItem  
 SetProperty  
 SetTempVar  
 ShowAllRecords  
 ShowToolbar  
 SingleStep  
 StopAllMacros  
 StopMacro

**Short Description**

Manages the categories displayed in the Navigation Pane  
 Manages menu items on custom or global menus in the Add-Ins tab  
 Sets a control's property  
 Creates and sets a temporary variable  
 Removes a filter from a table, query, or form  
 Displays or hides a group of commands in the Add-Ins tab  
 Pauses Macro execution  
 Stops all running Macros  
 Stops a running Macro

Some Actions have no Arguments, whereas others contain one or more. An Action's Argument enables you to customize how the Action will behave. A short description can be seen for each Argument at the bottom right of the screen by placing your cursor in the Action Arguments section for each value. Comments for each Action can also be added in the Comment column.

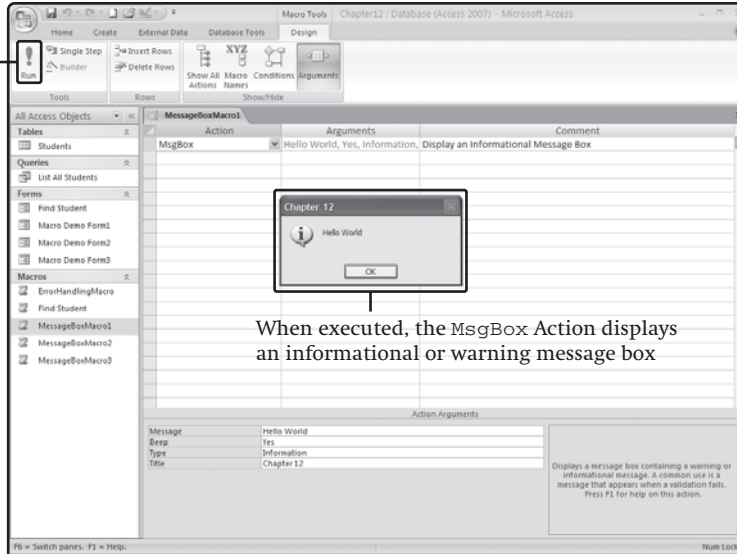
**FIGURE 12.2**

Selecting a Macro  
 Action and  
 defining the  
 Action's  
 Arguments.

After your Macro has been created and saved, the Actions can be executed by clicking the Run icon in the Tools section of the Design tab, as seen in Figure 12.3.

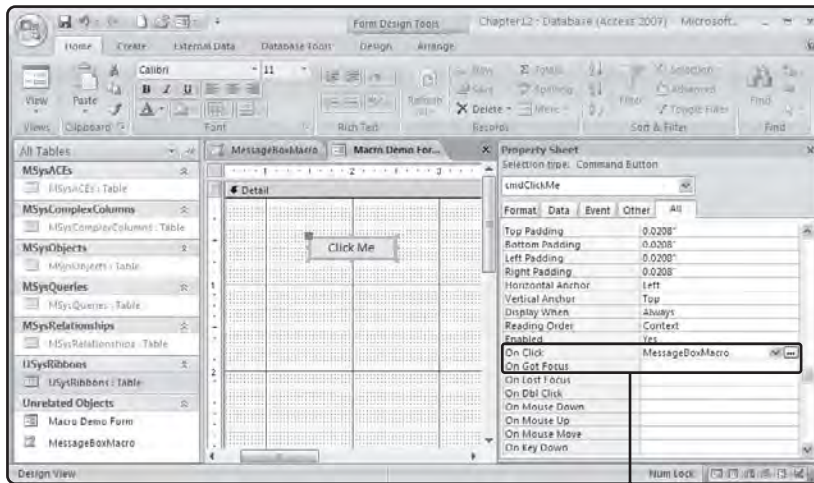
With my Macro created, I can assign the Macro to a Report, Form, or Control event property. For example, Figure 12.4 demonstrates assigning a Macro name to the On Click event property of a Command Button.

Click the Run icon to execute the active Macro



**FIGURE 12.3**

Running a Macro in Design mode.



**FIGURE 12.4**

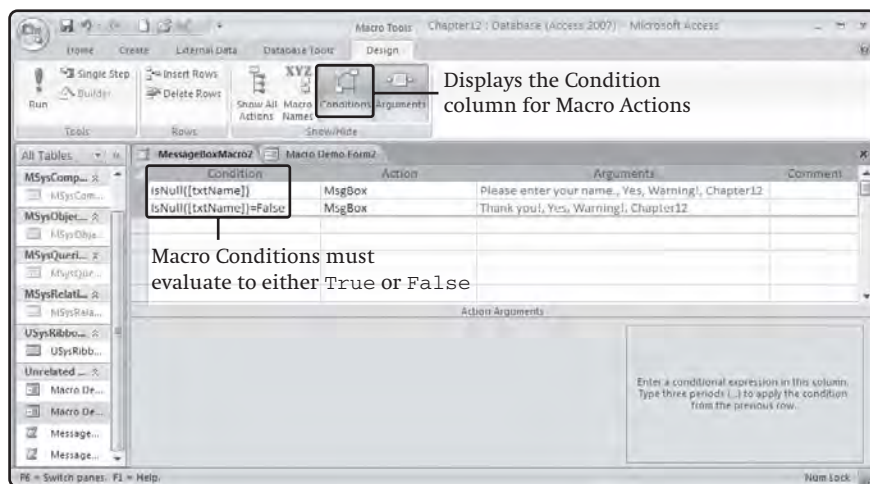
Assigning a Macro to a control's On Click event property.

Assign the name of a saved Macro to a control's event property

Macros can also be executed from a VBA event procedure via the RunMacro method of the DoCmd object. For example, the next VBA Form\_Load() event procedure runs the Macro called MsgBoxMacro by passing the Macro name (in double quotes) to the RunMacro method of the DoCmd object.

```
Private Sub Form_Load()
    DoCmd.RunMacro "MessageBoxMacro"
End Sub
```

Conditions can be added to Macros to ensure a set of criteria is met before the Macro Action occurs. Macro conditions are much like VBA conditions in that they must evaluate to either `True` or `False`. If the Macro Condition evaluates to `False`, the Macro Condition will not execute. If the Macro Condition evaluates to anything other than `False`, the Macro Action will execute. To place Conditions in your Macros, simply click the Conditions icon in the Show/Hide area of the Design tab as shown in Figure 12.5.



**FIGURE 12.5**

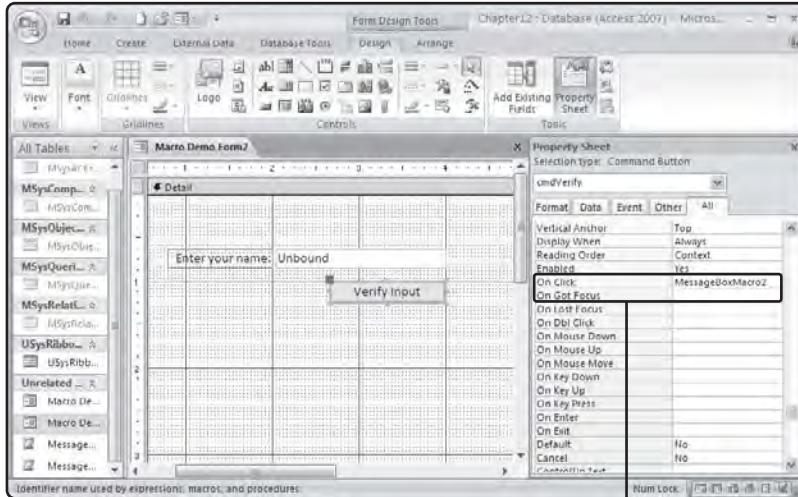
Enabling and creating Macro Conditions.



Enter `False` as a Condition to temporarily have Access ignore a Macro Action.

As seen in Figure 12.5, I created two Actions for my Macro. The first Action uses the `IsNull` function to check for text in the `txtName` Text Box. If the Condition evaluates to `True`, the Action will execute, in this case alerting the user to enter his or her name. The second Condition also uses the `IsNull` function, but in this case the Action will only execute if the Condition evaluates to `False`.

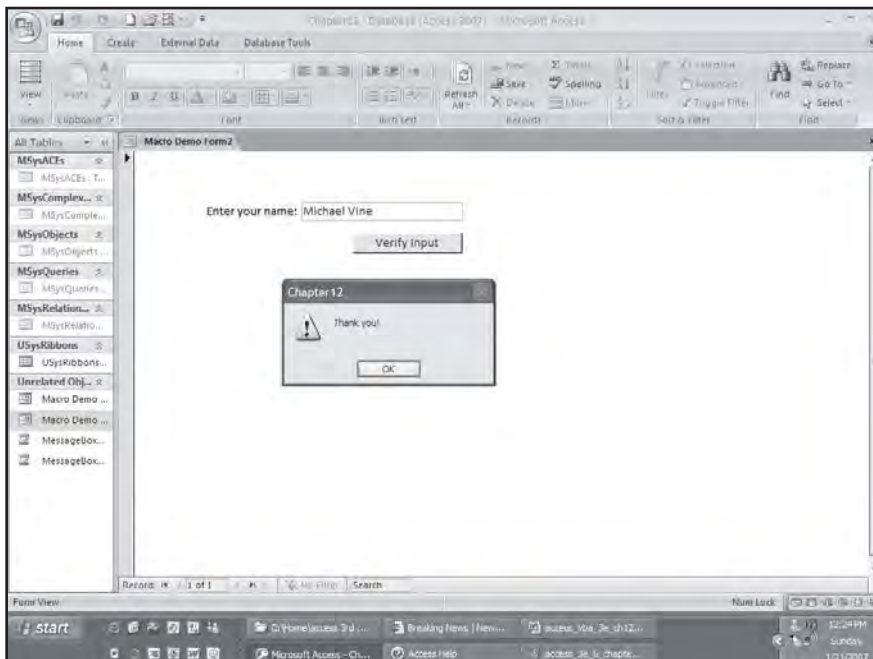
I'll try my new Macro with Conditions by creating a simple Form containing one Text Box with corresponding Label, and one Command Button. Next, I will assign my new Macro called `MessageBoxMacro2` to the Command Button's `On Click` event property, as shown in Figure 12.6.



Assign Macro name to On Click event

**FIGURE 12.6**  
Assigning the  
MessageBoxMacro2  
to a Command  
Button's On  
Click event.

After entering my name (Michael Vine) into the Text Box control, I click the Command Button that has the associated Macro name assigned to the On Click event property with output shown in Figure 12.7.



**FIGURE 12.7**  
Sample output  
using a Macro with  
Conditions.

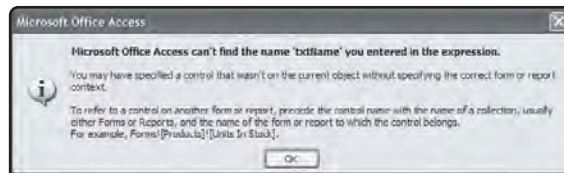


Access objects (for example, Control or Form name) used in a Macro's Condition should have an explicit reference to the object's location. For example, the following Macro Condition will generate an error, shown in Figure 12.8, if the Macro is run in the Macro Design view.

```
IsNull([txtName])
```

**FIGURE 12.8**

Attempting to execute a Macro in Macro Design view that has an improperly referenced control in a Condition.



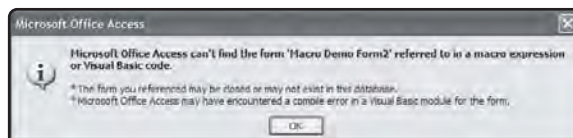
To successfully execute a Macro in Macro Design view that has a reference to a Control name anywhere in the Macro (for example, Macro Condition or Argument value), you must explicitly tell the Access Macro what Report or Form the Control belongs to using the Forms or Reports Collection, followed by the Form or Report name, then the Control name, with an exclamation mark (!) in between each object. Each object referenced should also be surrounded by brackets ([ ]), as demonstrated in the following code.

```
IsNull([Forms]![Macro Demo Form2]![txtName])
```

With Forms, Reports, and/or Controls properly referenced, it's important to know that a Macro still cannot execute by itself in Macro Design view without the associated Form or Report also opened in Design, Report, or Layout view. Attempting to execute a stand-alone Macro in Design view without the associated Form or Report opened will generate an error, as shown in Figure 12.9.

**FIGURE 12.9**

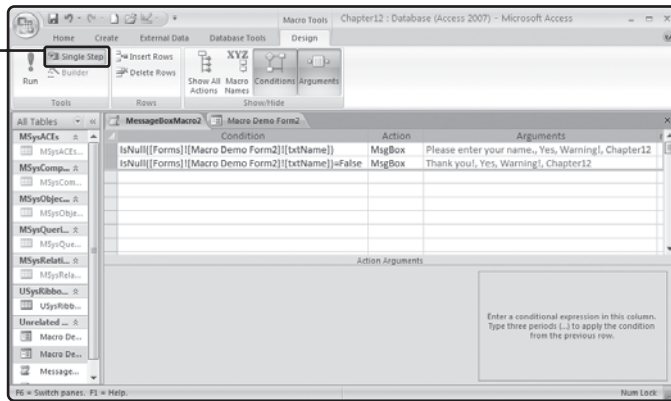
Attempting to execute a Macro in Macro Design view without the associated Form or Report also opened.



## Macro Troubleshooting and Error Handling

As you learned in Chapter 8, “Debugging, Input Validation, File Processing, and Error Handling,” Access VBA provides mechanisms for debugging code such as stepping through one line of code at a time. A similar offering is given to Macro developers to allow them to step through Macro Actions and isolate any defects or errors encountered. This is accomplished by clicking the Single Step icon in the Tools area of the Macro Design tab, as shown in Figure 12.10.

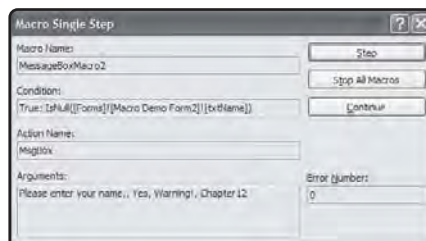
Step through Macro Actions



**FIGURE 12.10**

Observe a Macro's flow by stepping through each Action.

Stand-alone Macro Actions can be executed successfully all at once or one at a time from within the Macro Design view by clicking the Run icon in the Tools area. If the Single Step option has been selected, a Macro Single Step window will appear (as shown in Figure 12.11) that allows you to view each Macro's Condition, Action name, Arguments, and any associated errors.



**FIGURE 12.11**

Stepping through one Action at a time using the Macro Single Step window.

Once selected, the Single Step functionality can also be invoked outside of the Macro Design view by assigning the Macro name to a Report, Form, or Control event property.



Just as with VBA code, it's always a good idea to implement error-handling routines for those unexpected results or defects or, as I like to say, "undocumented features." Macro developers can implement error handling routines using the `OnError` Action, which has two Arguments, which are described next.

- **Go to**—Denotes the behavior to occur when an error is generated. Default values include:
  - **Next**—Moves Macro execution to the next Action.
  - **Macro Name**—Stops Macro execution and runs the Macro identified in the Macro Name Argument.
  - **Fail**—Stops Macro execution and displays an error message.
- **Macro Name**—Identifies the Macro to be used in error handling if the Macro Name **Go to** value is selected. For example, the error handling Macro could be titled `ErrorHandler`, and include a `MsgBox` Action to customize error communication to the user.

Generally speaking, the `OnError` Action should be placed at the top of the Macro Actions (before any other Action) so that error handling is in effect for subsequent Actions.

Access 2007 also includes the `MacroError` object that incorporates properties for determining specific information about run-time Macro errors. The next bulleted list describes properties of the `MacroError` object.

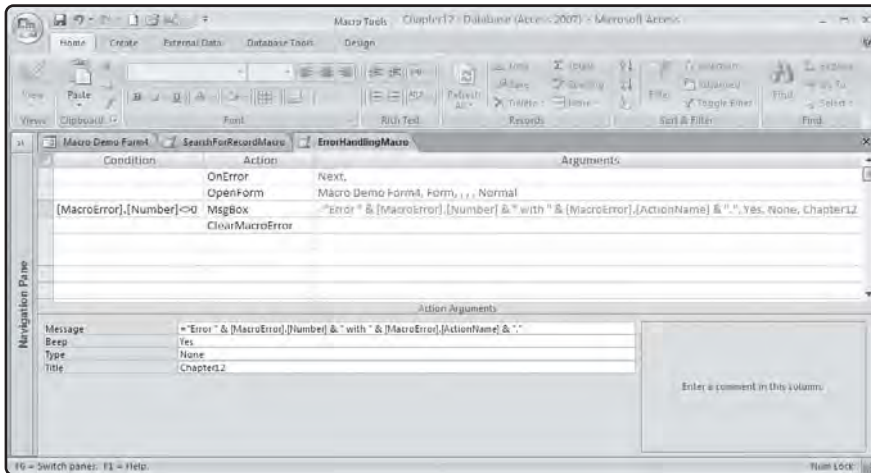
- **ActionName**—The Macro Action that was running when the error occurred.
- **Arguments**—The specified Arguments in the Macro Action that was running when the error occurred.
- **Condition**—The Condition the Macro Action was using when the error occurred.
- **Description**—Represents the error message text.
- **MacroName**—The name of the Macro executed when the error occurred.
- **Number**—The current error number.

The `MacroError` object and its properties can be used in Macro Actions to customize and display information about Macro errors. Because the `MacroError` object contains information about a single error only, the `MacroError` object will hold information about the last error encountered in a series of errors. The `MacroError` object and its properties can be used in Macro Actions (for example, the `Message` argument of the `MsgBox` Action) to customize and display information about errors as the next line of code demonstrates.

```
"Error " & [MacroError].[Number] & " with " & [MacroError].[ActionName] & "."
```

Although all property information in the `MacroError` object is reset (cleared) when a Macro ends, Access 2007 does provide the `ClearMacroError` Action to clear property information

for the MacroError object. Figure 12.12 demonstrates error handling using the OnError and ClearMacroError Actions in conjunction with the MacroError object.



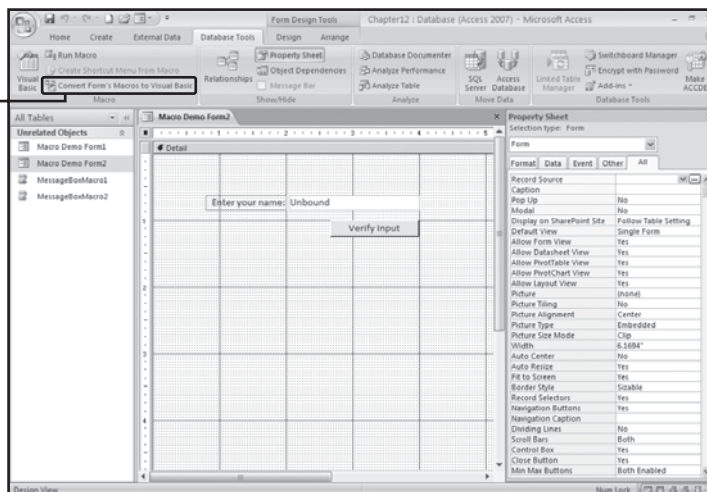
**FIGURE 12.12**

Incorporating error handling into Macros using the OnError and ClearMacroError Actions.

## Converting Macros to VBA

Microsoft Access 2007 can convert stand-alone Macros to VBA procedures, which will in turn perform similar actions as the original Macro. This conversion process is fully automated and is easily performed by simply opening the Form or Report in Design view and clicking the Convert Form's (or Report's) Macros to Visual Basic icon from the Macro area of the Database Tools tab, as shown in Figure 12.13.

Converts a Form's stand-alone Macros to VBA code



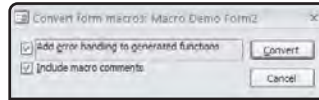
**FIGURE 12.13**

Viewing the Convert Form's Macros to Visual Basic option.

Once selected, Access prompts you with two conversion options, which are to add VBA error handling to generated functions and include Macro comments as revealed in Figure 12.14.

**FIGURE 12.14**

Selecting Macro to VBA code conversion options.



After the Macro conversion has completed, Access creates all necessary VBA procedures and code to produce similar functionality, and re-assigns values in event properties (for example, On Click) from a Macro name to the [Event Procedure] value.

**Stand-alone Macro objects are not deleted from the Access database after a Macro to VBA conversion completes.**

In the case of my Macro created earlier titled MsgBoxMacro2 (shown in Figure 12.10), the Macro conversion process creates the following VBA code.

```
'-----
' cmdVerify_Click
'
'-----

Private Sub cmdVerify_Click()
On Error GoTo cmdVerify_Click_Err

    If (IsNull(Forms![Macro Demo Form2]!txtName)) Then
        Beep
        MsgBox "Please enter your name.", vbExclamation, "Chapter12"
    End If
    If (IsNull(Forms![Macro Demo Form2]!txtName) = False) Then
        Beep
        MsgBox "Thank you!", vbExclamation, "Chapter12"
    End If
```

```
cmdVerify_Click_Exit:
    Exit Sub

cmdVerify_Click_Err:
    MsgBox Error$
    Resume cmdVerify_Click_Exit

End Sub
```

Notice the two `If` conditions in the preceding Command Button `Click` event procedure, one checking for a `True` value and the other for a `False` value. These VBA code conditions contain the same conditional logic and subsequent actions as did the converted Macro! In addition to VBA conditions, the conversion process also created VBA error-handling routines for my event procedure, as specified in the conversion options window (refer to Figure 12.13).

## ACCESS DATABASE PERFORMANCE CONSIDERATIONS

There comes a time in every developer's life when an application's performance has become undesirable. Generally speaking, performance problems are the result of one or more oversights that may include an unexpected increase in data volume that slows query and data operation responses, not having enough time to tune the application properly during or after application development, or not having the knowledge about what performance optimizations to consider in the first place. Whether you created the application yourself or inherited the support of an existing application, performance issues can lead to end-user dissatisfaction and ultimately a lack of trust and use of the application. By learning and leveraging key performance considerations and techniques, you'll be sure to keep your Access database application humming.

The most common areas one should consider Access performance optimizations are with Forms, VBA code, Queries, and Indexes. I will show you common optimization techniques in each of these areas that you should consider during your application design, construction, and maintenance. Before moving into area-specific performance considerations, consider the following general recommendations for Access installed on a local computer:

- For non-shared databases, install Access and all databases on a local disk drive rather than on a network shared drive.
- Open an Access database for "Exclusive Use" if the database has only one user. This is accomplished one of two ways: First, in the Access Open dialog box, click the arrow next to the Open button and select Open Exclusive. Or, use the `/exc1` command line switch in

a program shortcut or via the Windows Start/Run procedure, as shown in the following line of code.

```
"c:\Access 2007\msaccess.exe" /exc1 "c:\database_folder\database_name.accdb"
```

- Ensure the Access program has enough memory to successfully run and execute processes by closing all unnecessary programs and ensuring available memory in your system meets minimal requirements, as described in the “System Requirements” section from Chapter 1.
- Keep Access databases on uncompressed disk drives.
- Regularly delete unneeded files from the Windows Recycle Bin and run the Windows Disk Defragmenter.
- Use blank screen savers, or none at all, and solid color desktop wallpapers.

## Forms

Though the performance of Forms may not be an obvious consideration during database tuning, the following recommendations can shave seconds off of the loading and processing of graphical controls and improve the overall performance of your Access application.

- Close all non-used Forms to reclaim memory.
- Set a Form’s `Data Entry` property to `Yes` for Forms that will be used primarily for entering new records. Otherwise, Access will read all records before showing the empty record at the end of the recordset.
- Use graphical objects sparingly, or convert graphics from color to black and white.
- Remove code from Forms that do not require VBA, and set the Form’s `Has Module` property to `No`. Instead of using VBA code for simple event procedures, consider using Macros. Forms without code modules can still have controls that call functions and procedures from a Standard module. Forms without code modules load more quickly.
- Only include fields in Sub Forms that are absolutely necessary. Base a Sub Form’s data on a saved Query rather than a Table.
- Fields used in a Sub Form that are part of criteria or are linked to the main Form should be indexed.
- The source of List and Combo Boxes should be based on a saved Query rather than a SQL statement or Table by changing the List or Combo Box’s `Row Source` property.
- Fields displayed in a List or Combo Box should be indexed.

## VBA Code

The following VBA code performance considerations should be evaluated during the design and development of new Access applications and can be applied to existing applications without changing the application's external behavior.

- Ensure variables are explicitly declared by checking the Require Variable Declaration option from the VBA code window's Tools/Options menu item.
- Remember to remove unused procedures and variables to conserve memory.
- Leverage Constants whenever possible.
- Use the `Variant` data type sparingly.
- Compile VBA code by either saving an Access databases in the ACCDE file format or by using the Compile option from the Debug menu in the VBA code window.
- When possible, leverage the `Integer` or `Long` data types for mathematical operations.
- Use indexed fields for the `FindRecord` and `FindNext` methods.
- Create object variables to store Control or Form property values rather than identifying and accessing the Control or Form property numerous times.
- Leverage the `Me` keyword for Form references within event procedures.
- Avoid using the `IF` function for return expressions that take a long time to process.
- Leverage the `Erase` or `ReDim` statements with dynamic arrays to reclaim memory.
- Encourage Access VBA to load modules as needed and conserve memory by putting related procedures in the same modules.

## Queries and Indexes

The most common approach to tuning a database's performance or solving an existing performance problem is to analyze your queries and indexes because optimizing queries and indexes often results in the largest gains in database performance. On the other hand, a poorly written query or the lack of proper index utilization can bring the system the database resides on to its knees. Consider the following recommendations to prevent a system lockup or a query that seems to never return any results.

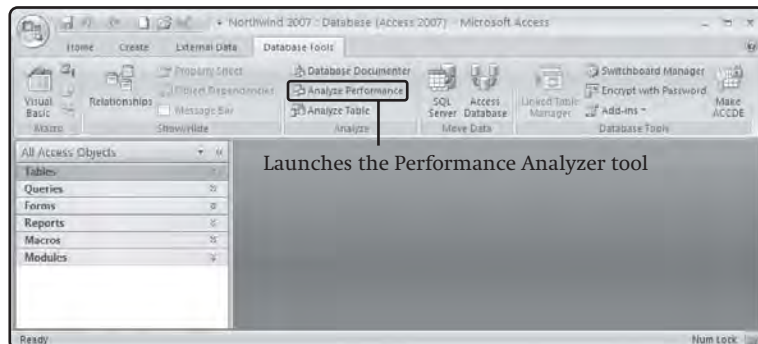
- Before creating a query, know how many rows are in the table or tables being queried and approximately how many rows you expect back from the query.
- When appropriate, leverage conditions in a SQL `WHERE` clause to minimize the number of rows returned by a query.
- When possible, avoid sorting hefty result sets, which can consume large amounts of memory.

- Leverage **GROUP BY** instead of **DISTINCT** to eliminate duplicate rows. The **GROUP BY** clause eliminates duplicate rows sooner in the query process than does the **DISTINCT** clause.
- Leverage indexes to find and sort data quicker. Indexes store record locations by the field or fields indexed. When a query finds the location from an index, it is able to move directly to the record's location rather than scanning through the entire table.
- As a general rule of thumb, indexes should be used on Primary and Foreign Keys.
- The **WHERE** clause in large queries should reference indexed fields.
- Create indexes on other columns that are used in table joins.
- Avoid indexing small tables to save the cost of loading and processing indexes.
- Avoid indexing columns that are frequently updated, which increases the amount of database write time.
- Only use indexes as required. Each index consumes disk space and must be managed by the database system.

## Performance Analyzer

Now that you've seen performance considerations and optimization best practices for Access databases, let me show you how to leverage the built-in Performance Analyzer for receiving feedback and hints directly from Access itself!

The Performance Analyzer can be accessed via the Analyze Performance icon in the Analyze area of the Database Tools tab, as shown in Figure 12.15.

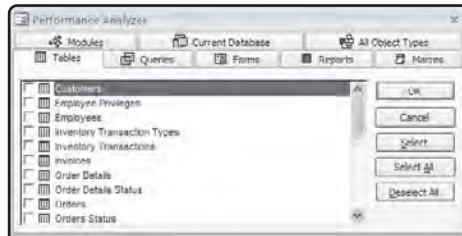


**FIGURE 12.15**

Accessing the Performance Analyzer tool from the Database Tools tab.

Once launched, the Performance Analyzer window, revealed in Figure 12.16, allows you to select which Access objects (Tables, Queries, Forms, Reports, Macros, and Modules), you'd like analyzed. Each tab allows you to select one or more object by category. You can also view all

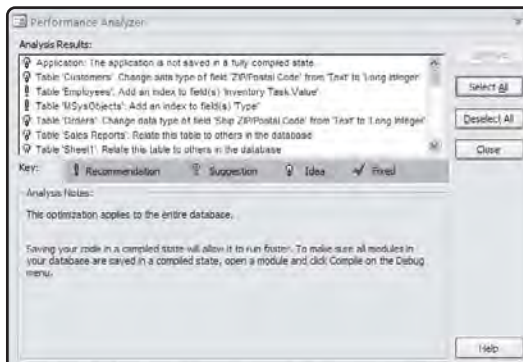
objects under the All Object Types tab. For the example, I ran Performance Analyzer against the Northwind database that comes with Access 2007, as shown in Figure 12.16.



**FIGURE 12.16**

The Performance Analyzer tool.

After Access objects have been selected for which you'd like analyzed, click the OK button. Using pre-determined performance-tuning considerations, the Performance Analyzer scans the identified objects and returns the results in a new window, as observed in Figure 12.17.



**FIGURE 12.17**

Viewing Performance Analyzer results.

Performance Analyzer considers and lists three types of results as defined next:

- **Recommendation**—Advice that will most likely improve the performance of the Access object analyzed. Performance Analyzer can perform these optimizations for you and mark the Recommendation as Fixed once complete.
- **Suggestion**—Advice that has potential trade-offs (good and bad) that should be considered before optimizing. Performance Analyzer can perform these optimizations for you and mark the Recommendation as Fixed once complete.
- **Idea**—General advice that may improve the performance of the object analyzed. Idea optimizations must be performed by you.



You can view information about each analysis by clicking a result in the list and viewing the details in the Analysis Notes box at the bottom of the window.

Although the Performance Analyzer tool does not provide suggestions for tuning the system your database is running on, it should be the first place you start when looking to optimize the performance of your Access database.

## SUMMARY

- Macros allow you to automate tasks and processes without VBA code.
- Stand-alone Macros can be created using the Macro Builder.
- Macro Actions contain common Access database processes that build automation.
- Macro Action Arguments allow you to customize how the Action will behave.
- Stand-alone Macros can be assigned to a Report, Form, or Control event property.
- Macros can also be executed from a VBA event procedure via the `RunMacro` method of the `DoCmd` object.
- Conditions can be added to Macros to ensure a set of criteria is met before the Macro Action runs.
- Macro Conditions evaluate to either `True`, causing the Action to execute, or `False`, preventing the Macro Action from executing.
- The Single Step option allows you view a Macro's Arguments, Conditions, and any associated errors one Action at a time.
- The `OnError` Action is used to implement error handling in Macros.
- Access 2007 includes the `MacroError` object that includes properties for determining specific information about run-time Macro errors.
- The `ClearMacroError` Action resets property information for the `MacroError` object.
- Microsoft Access 2007 is capable of converting stand-alone Macros to VBA code.
- The most common Access database areas to consider while performance tuning are with Forms, Queries, Indexes, and VBA code.
- Indexes store record locations by the field or fields indexed.
- Access 2007 includes the Performance Analyzer tool that can aid in optimizing your database application by reviewing a database's design and providing recommendations for change.

- Performance Analyzer lists three types of optimization advice, including Recommendations, Suggestions, and Ideas.
- Optimization Recommendations and Suggestions can be performed by the Performance Analyzer. Idea optimizations must be performed by you.

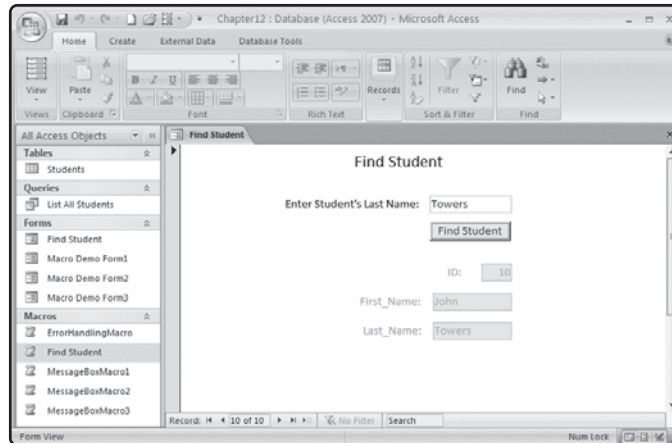
## PROGRAMMING CHALLENGES

1. Create a new Form with one Command Button titled cmdCloseForm. Create a new Macro called Close Form with the Close Action value. The Arguments for the Close Action should close the Form. Save both objects and test the Macro by assigning the Macro name to the Command Button's On Click Event property.
2. Create a new Form with one Text Box and one Command Button. The Form will allow a user to enter their age into the Text Box and click a Command Button to validate input. Create a new Macro called Test User Input with one MsgBox Action that uses the IsNumeric function in the Action's Condition to validate numeric (non-text) data was entered into the Text Box. If the user did not enter numeric data into the Text Box, use the MsgBox Action to display a Message Box to the user.
3. Create an Access table named Students with three columns, Student\_Id, Last\_Name, and First\_Name. Insert sample records into the Students table. Create a new Access Form called Find Student, similar to the one shown in Figure 12.18, with the controls and properties shown in Table 12.2.  
Create a Macro also called Find Student that uses the SearchForRecord Macro Action. The SearchForRecord Macro Action should use input from the txtLastName Text Box to locate a matching record in the Students table and in turn display the ID, First\_Name, and Last\_Name information in the corresponding Text Boxes. Hint: The *Where Condition* Argument of the SearchForRecord Action should resemble the following:

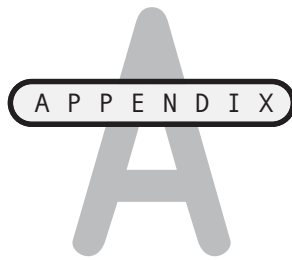
```
= "Last_Name = '" & [Forms]![Manage Students]![txtLastName] & "'"
```

**FIGURE 12.18**

The Find Student Form from Programming Challenge #3.

**TABLE 12.2 CONTROLS AND PROPERTIES OF THE FIND STUDENT FORM**

Control	Property	Property Value
Form	Name	Find Student
	Caption	Programming Challenge #3
	Record Source	Students
Text Box	Name	txtLastName
	Label	lblLastName
Text Box	Caption	Enter Student's Last Name:
	Name	txtID
	Control Source	Student_Id
	Enabled	No
Text Box	Name	lblID
	Caption	ID:
	Control Source	First_Name
Text Box	Enabled	No
	Name	lblFirstName
	Caption	First Name:
Text Box	Name	txtLastName
	Control Source	Last_Name
	Enabled	No
Text Box	Name	lblLastName
	Caption	Last Name:
	Control Source	cmdFindStudent
Text Box	Name	Find Student
	Caption	Find Student
	On Click	Find Student (Macro name)



# COMMON CHARACTER CODES

The items in this table represent the most common characters and associated character codes used in conjunction with the `Chr` and `Asc` functions.

Code	Character
8	Backspace
9	Tab
10	Line feed
13	Carriage return
32	Spacebar
33	!
34	"
35	#
36	\$
37	%
38	&
39	'

Code	Character
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q

Code	Character
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{

Code	Character
124	
125	}
126	~
127	Del (Delete key)

# KEYBOARD SHORTCUTS FOR THE CODE WINDOW

The items in the following table represent common keyboard shortcuts that can be used in the Visual Basic Environment's (VBE) Code window.

## Task

Beginning of module  
Clear all breakpoints  
Delete current line  
Delete to end of word  
End of module  
Find  
Find next  
Find previous  
Go to last position  
Indent  
Move one word to left  
Move one word to right

## Shortcut

Ctrl+Home  
Ctrl+Shift+F9  
Ctrl+Y  
Ctrl+Delete  
Ctrl+End  
Ctrl+F  
F3  
Shift+F3  
Ctrl+Shift+F2  
Tab  
Ctrl+Left Arrow  
Ctrl+Right Arrow

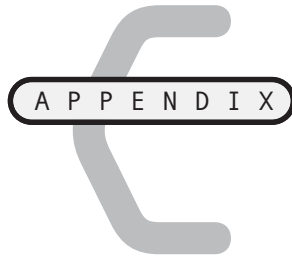


**Task**

Move to beginning of line  
Move to end of line  
Next procedure  
Outdent  
Previous procedure  
Replace  
Shift one screen down  
Shift one screen up  
Undo  
View Code window  
(control selected in Design view)  
View definition  
View Object Browser  
View shortcut menu

**Shortcut**

Home  
End  
Ctrl+Down Arrow  
Shift+Tab  
Ctrl+Up Arrow  
Ctrl+H  
Ctrl+Page Down  
Ctrl+Page Up  
Ctrl+Z  
  
F7  
Shift+F2  
F2  
Shift+F10



# TRAPPABLE ERRORS

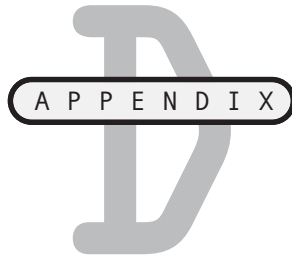
**M**icrosoft provides a comprehensive list of trappable errors that you can catch, display, and troubleshoot during runtime, development, or compile time.

Error Code	Error Description
3	Return without GoSub
5	Invalid procedure call
6	Overflow
7	Out of memory
9	Subscript out of range
10	This array is fixed or temporarily locked
11	Division by 0
13	Type mismatch
14	Out of string space
16	Expression too complex
17	Can't perform requested operation
18	User interruption (Ctrl + Break) occurred

Error Code	Error Description
20	Resume without error
28	Out of stack space
35	Sub, Function, or Property not defined
47	Too many code resource or DLL application clients
48	Error in loading code resource or DLL
49	Bad code resource or DLL calling convention
51	Internal error
52	Bad filename or number
53	File not found
54	Bad file mode
55	File already open
57	Device I/O error
58	File already exists
59	Bad record length
61	Disk full
62	Input past end of file
63	Bad record number
67	Too many files
68	Device unavailable
70	Permission denied
71	Disk not ready
74	Can't rename with different drive
75	Path/file access error
76	Path not found
91	Object variable or With block variable not set
92	For loop not initialized
93	Invalid pattern string
94	Invalid use of Null
97	Can't call Friend procedure on an object that is not an instance of the defining class
98	A property or method call cannot include a reference to a private object, either as an argument or as a return value
298	System resource or DLL could not be loaded
320	Can't use character device names in specified filenames
321	Invalid file format
322	Can't create necessary temporary file
325	Invalid format in resource file
327	Data value named not found
328	Illegal parameter; can't write arrays

Error Code	Error Description
335	Could not access system registry
336	Component not correctly registered
337	Component not found
338	Component did not run correctly
360	Object already loaded
361	Can't load or unload this object
363	Control specified not found
364	Object was unloaded
365	Unable to unload within this context
368	The specified file is out of date; this program requires a later version
371	The specified object can't be used as an owner form for Show
380	Invalid property value
381	Invalid property-array index
382	Property Set can't be executed at runtime
383	Property Set can't be used with a read-only property
385	Need property-array index
387	Property Set not permitted
393	Property Get can't be executed at runtime
394	Property Get can't be executed on write-only property
400	Form already displayed; can't show modally
402	Code must close topmost modal form first
419	Permission to use object denied
422	Property not found
423	Property or method not found
424	Object required
425	Invalid object use
429	Component can't create object or return reference to this object
430	Class doesn't support Automation
432	Filename or class name not found during Automation operation
438	Object doesn't support this property or method
440	Automation error
442	Connection to type library or object library for remote process has been lost
443	Automation object doesn't have a default value
445	Object doesn't support this action

Error Code	Error Description
446	Object doesn't support named arguments
447	Object doesn't support current locale setting
448	Named argument not found
449	Argument not optional or invalid property assignment
450	Wrong number of arguments or invalid property assignment
451	Object not a collection
452	Invalid ordinal
453	Specified code resource not found
454	Code resource not found
455	Code resource lock error
457	This key is already associated with an element of this collection
458	Variable uses a type not supported in Visual Basic
459	This component doesn't support the set of events
460	Invalid Clipboard format
461	Method or data member not found
462	The remote server machine does not exist or is unavailable
463	Class not registered on local machine
480	Can't create AutoRedraw image
481	Invalid picture
482	Printer error
483	Printer driver does not support specified property
484	Problem getting printer information from the system; make sure printer is set up correctly
485	Invalid picture type
486	Can't print form image to this type of printer
520	Can't empty Clipboard
521	Can't open Clipboard
735	Can't save file to TEMP directory
744	Search text not found
746	Replacements too long
31001	Out of memory
31004	No object
31018	Class is not set
31027	Unable to activate object
31032	Unable to create embedded object
31036	Error saving to file
31037	Error loading from file



# VISUAL BASIC ENVIRONMENT OPTIONS

**M**icrosoft enables you to customize the look and feel of the VBE (Visual Basic Environment) development environment by way of the Options dialog box, which can be accessed via the VBE's Tools menu.

The Options dialog box contains four tabs, with each tab revealing customizable settings for the Editor, Editor Format, General, and Docking areas. The customizable options in each tab are described in the following tables.

**TABLE D.1** EDITOR OPTIONS

Option	Option Description
Auto Syntax Check	Verifies correct syntax after a line of code is entered
Require Variable Declaration	Requires explicit variable declaration in modules
Auto Indent	After the first line of code is tabbed, all subsequent lines start at that tab location
Tab Width	Sets the tab width, ranging from 1 through 32 spaces (4 spaces is default)
Default to Full Module View	New module procedures are displayed in the Code window as a single scrollable list, or one procedure at a time
Procedure Separator	Displays separator bars in the Code window at the end of each procedure
Auto List Members	Displays information that logically completes a VBA statement
Auto Quick Info	Provides information about functions and arguments as you type
Auto Data Tips	In break mode only, displays values of variables during a mouse over
Drag-Drop in Text Editing	Allows code to be dragged from the Code window to the Immediate or Watch windows

**TABLE D.2** EDITOR FORMAT OPTIONS

Option	Option Description
Foreground, Background, and Indicator	The color of different categories of text
Font	The font for displaying code
Size	The font size used for code
Margin Indicator Bar	Displays the Margin Indicator Bar

**TABLE D.3    GENERAL OPTIONS**

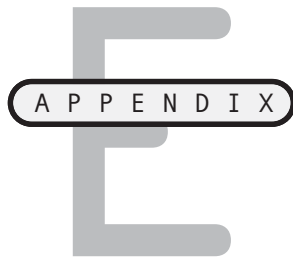
<b>Option</b>	<b>Option Description</b>
Show Grid	Displays the form grid
Grid Units	The unit of measurement for the grid
Width	The form's grid cell width
Height	The form's grid cell height
Align Controls to Grid	Automatically places the outer edge of controls on the nearest grid lines
Show ToolTips	Displays ToolTips for toolbar buttons
Collapse Proj. Hides Windows	Closes the project, form, object, or module windows when a project is collapsed in the Project Explorer
Notify Before State Loss	Displays message that an action will cause all module-level variables to be reset
Break on All Errors	All errors cause the project to enter break mode
Break in Class Module	Unhandled errors from a class module cause the project to enter break mode
Break on Unhandled Errors	Unhandled errors cause the project to enter break mode
Compile on Demand	Code is fully compiled before a project starts or as needed
Background Compile	During runtime, idle time is used to finish compiling the project in background

**TABLE D.4    DOCKING OPTIONS**

<b>Option</b>	<b>Option Description</b>
Dockable	Allows a window to be anchored to an adjacent Dockable window or the VBE window



*This page intentionally left blank*



# RESERVED WORDS AND SYMBOLS

**R**eserved words and symbols have significance to Access 2007 and the Access 2007 database engine. Errors can be encountered when using reserved words and symbols during the naming and use of fields, tables, controls, variables, and other Access objects. These errors can be encountered in either design time or runtime but may not be informative enough to denote a reserved word or symbol as the cause of the error. Reserved word and symbol errors can be avoided by surrounding reserved words and symbols with brackets ([ ]), but as a best practice, it's wise not to use reserved words nor symbols when naming Access objects.

**TABLE E.1    RESERVED WORDS AND SYMBOLS****Reserved Words and Symbols**

'	BEGIN
-	BETWEEN
!	BINARY
"	BIT
#	BIT_LENGTH
\$	BNOT
%	BOOLEAN
&	BOR
*	BOTH
.	BXOR
/	BY
:	BYTE
;	CASCADE
?	CASCADED
ABSOLUTE	CASE
ACTION	CAST
ADD	CATALOG
ADMINDB	CHAR
ALL	CHAR, CHARACTER
ALLOCATE	CHAR_LENGTH
Alphanumeric	CHARACTER
ALPHANUMERIC	CHARACTER_LENGTH
ALTER	CHECK
AND	CLOSE
ANY	COALESCE
Application	COLLATE
ARE	COLLATION
AS	COLUMN
ASC	COMMIT
ASSERTION	COMP
Assistant	CompactDatabase
AT	COMPRESSION
AUTHORIZATION	CONNECT
AUTOINCREMENT	CONNECTION
AVG	CONSTRAINT
Avg	CONSTRAINTS
BAND	CONTAINER

**Reserved Words and Symbols**

Container	DESC
CONTINUE	DESCRIBE
CONVERT	Description
CORRESPONDING	DESCRIPTOR
COUNT	DIAGNOSTICS
Count	DISALLOW
COUNTER	DISCONNECT
CREATE	DISTINCT
CreateDatabase	DISTINCTROW
CREATEDB	Document
CreateField	DOMAIN
CreateGroup	DOUBLE
CreateIndex	DROP
CreateObject	Echo
CreateProperty	Else
CreateRelation	ELSE
CreateTableDef	End
CreateUser	END
CreateWorkspace	END-EXEC
CROSS	Eqv
CURRENCY	Error
CURRENT	ESCAPE
CURRENT_DATE	EXCEPT
CURRENT_TIME	EXCEPTION
CURRENT_TIMESTAMP	EXCLUSIVECONNECT
CURRENT_USER	EXEC
CurrentUser	EXECUTE
CURSOR	EXISTS
DATABASE	Exit
DATE	EXTERNAL
DATETIME	EXTRACT
DAY	FALSE
DEALLOCATE	FETCH
DEC	Field, Fields
DECIMAL	FillCache
DECLARE	FIRST
DEFAULT	FLOAT
DEFERRABLE	FLOAT4
DEFERRED	FLOAT8
DELETE	FOR

**Reserved Words and Symbols**

FOREIGN	INSENSITIVE
Form, Forms	INSERT
FOUND	InsertText
FROM	INT
Full	INTEGER
FULL	INTEGER 1
FUNCTION	INTEGER2
GENERAL	INTEGER4
GET	INTERSECT
GetObject	INTERVAL
GetOption	INTO
GLOBAL	IS
GO	ISOLATION
GOTO	JOIN
GotoPage	KEY
GRANT	LANGUAGE
GROUP	LAST
GROUP BY	LastModified
GUID	LEADING
HAVING	LEFT
HOUR	LEVEL
IDENTITY	Level
Idle	Like
IEEEDOUBLE	LIKE
IEEESINGLE	LOCAL
If	LOGICAL
IGNORE	LOGICAL 1
IMAGE	LONG
IMMEDIATE	LONGBINARY
Imp	LONGCHAR
IN	LONGTEXT
INDEX	LOWER
Index	Macro
Indexes	MATCH
INDICATOR	Match
INHERITABLE	MAX
ININDEX	MEMO
INITIALLY	MIN
INNER	MINUTE
INPUT	Mod

**Reserved Words and Symbols**

Module	OWNERACCESS
MODULE	PAD
MONEY	Parameter
MONTH	PARAMETERS
Move	Partial
NAME	PARTIAL
NAMES	PASSWORD
NATIONAL	PERCENT
NATURAL	PIVOT
NCHAR	POSITION
NewPassword	PRECISION
NEXT	PREPARE
NO	PRESERVE
Not	PRIMARY
NOT	PRIOR
NOTE	PRIVILEGES
Note	PROC
NULL	PROCEDURE
NULLIF	Property
NUMBER	PUBLIC
NUMERIC	Queries
Object	Query
OBJECT	Quit
OCTET_LENGTH	READ
OFF	REAL
OFOLEBJECT	Recalc
OLEOBJECT	Recordset
ON	REFERENCES
ONONLY	Refresh
OPEN	RefreshLink
OpenRecordset	RegisterDatabase
OPTION	Relation
OR	RELATIVE
ORDER	Repaint
Orientation	RepairDatabase
ORORDER	Report
Outer	Reports
OUTER	Requery
OUTPUT	RESTRICT
OVERLAPS	REVOKE

**Reserved Words and Symbols**

RIGHT	TEXT
ROLLBACK	THEN
ROWS	TIME
SCHEMA	TIMESTAMP
SCREEN	TIMEZONE_HOUR
SCROLL	TIMEZONE_MINUTE
SECOND	TO
SECTION	TOP
SELECT	TRAILING
SELECTSCHEMA	TRANSACTION
SELECTSECURITY	TRANSFORM
SESSION	TRANSLATE
SESSION_USER	TRANSLATION
SET	TRIM
SetFocus	TRUE
SetOption	Type
SHORT	UNION
SINGLE	UNIQUE
SIZE	UNIQUEIDENTIFIER
SMALLINT	UNKNOWN
SOME	UPDATE
SPACE	UPDATEIDENTITY
SQL	UPDATEOWNER
SQLCODE	UPDATESECURITY
SQLERROR	UPPER
SQLSTATE	USAGE
StDev	USER
StDevP	USING
STRING	VALUE
SUBSTRING	VALUES
SUM	Var
Sum	VARBINARY
SYSTEM_USER	VARCHAR
TABLE	VarP
TableDef	VARYING
TableDefs	VIEW
TABLEID	WHEN
TableID	WHENEVER
TEMPORARY	WHERE

**Reserved Words and Symbols**

WITH	YEAR
WORK	Year
Workspace	YES
WRITE	YESNO
Xor	ZONE



*This page intentionally left blank*

# INDEX

## A

- About form, controls and properties of, 265
- AbsolutePosition property, 256
- ACCDB files, 6, 16
- ACCDE files, 16
- ACCDR files, 16
- ACCDT files, 16
- Access 2007 features, 6–16
- Access VBA. *See also* macros
  - arithmetic of, 73–74
  - comments, 56
  - event-driven paradigm, 49–50
  - introduction to, 55–56
  - Me keyword, 58–59
  - object-based programming, 50–51
  - procedures, 53–55
- AccessConnection property, 243
- Add method, 291
- Add Procedure dialog box, 156, 281
- Add Watch dialog box, 191–192
- AddItem method, 117–118
- addition operator, 74
- ADO (ActiveX Data Objects)
  - adding records, 261–262
  - browsing data, 249–257
  - connecting to database, 242–246
  - connection errors, troubleshooting, 246
  - cursor types in, 248–249
  - database locks, 247
  - deleting records, 262–264
  - encapsulating library, 285
  - freeing/reclaiming resources, 289–290
  - opening recordset with, 243–254
  - ordinal position of field returned, 254–255
  - overview of, 241–242
  - recordsets, working with, 246–264
  - remote database, connecting to, 243–244
  - retrieving data, 249–257
  - terminology, 242
  - updating records, 258–261
- ADO object model, 242
- Alt key, 65
- altering tables, 236–237
- ampersand (&) character, 65
- Analyze Performance icon, 318
- And operator
  - for SQL queries, 222
  - truth table for, 85–86
- and/or variables. *See* expressions
- ANSI SQL, 215–216
- ANSI values, checking range of, 196
- API (application programming interface), 242.
  - See also* ADO (ActiveX Data Objects)
- arguments, 158–161
  - ADO Connection object as, 288–289
  - arrays as, 169–170
  - for macros, 306
  - names, 159
  - parentheses with, 160
- arithmetic of VBA, 73–74
- arrays, 163–164
  - arguments, passing as, 169–170
  - declaring arrays, 168
  - dynamic arrays, 167–169
  - elements, 164
  - initializing elements, 165

- single-dimension arrays, 164–166
- two-dimensional arrays, 166–167
- of user-defined types, 173–176
- zero, use of, 164
- Asc function, 144
  - range of values, checking, 196
  - sorting in SQL with, 227–228
- ASP (Active Server Pages). *See* ADO (ActiveX Data Objects)
- Assert method with Debug object, 200–201
- Assets template, 10
- assignment operator, 59
- assignment statement, 55–56, 59
  - If block and, 82
  - for text box values, 65
- Attachment data
  - fields, 29
  - type, 15
- attributes
  - defined, 18
  - field attributes, 30
- AutoNumber
  - data fields, 29
  - INSERT INTO statement with, 233
- AVG function, 223–224

## B

- BackColor property, 63
  - values in assigning, 64
- BackStyle property, 63
- BASIC, GoTo keyword in, 197
- benefits of Access 2007, 2–3
- binary codes, 70
- Blank Database icon, 25
- Boolean data, 68
  - expressions and, 81
  - variables storing, 67
- brackets
  - for reserved words/symbols, 337–343
  - for table/column names, 219
- break mode, 187
  - Immediate window, use of, 188
  - re-executing program in, 188
- breakpoints, 187–188
- browsing data with ADO, 249–257
- buffers, 203
- bugs. *See also* debugging
  - defined, 186
- button controls, 38
  - adding, 40–41
  - naming conventions, 42
  - wizards for, 41
- buttons
  - option buttons, 95–98
  - toggle buttons, 99–101
- Buttons parameter for MsgBox function, 89
- ByVal keyword, 158

## C

- C language, 69. *See also* ADO (ActiveX Data Objects)
  - dynamic arrays in, 168
  - user-defined types, 170
  - void functions, 155
- calendar, 13
- Cartesian joins/results, 231
- Case Else statement, 87
- case-sensitivity
  - of constant name, 69
  - in SQL (Structured Query Language), 217
- character codes, 143–144
  - common character codes, 323–326
- character values, checking range of, 195–196
- check boxes, 98–99
  - naming conventions, 42
- Choose My Adventure program, 264–274
  - About form, controls and properties of, 265
  - Math form, controls and properties of, 266
- Chr function, 143–144
- class modules, 1671

- method procedures, 284–287
- in OOP, 279–281
- classes
  - in OOP, 278
  - property procedures, 281–284
- Click event procedure, 165–166
- Close function, 205
- closing data files, 205
- COBOL, GoTo keyword in, 197
- code. *See also* debugging
  - duplicate code, removing, 154–156
  - performance considerations, 317
  - reuse of, 153–161
  - stepping through, 186–187
- Code window, 53
  - keyboard shortcuts for, 327–328
  - procedures in, 54
- collections, 290–291
  - accessing members in, 292–293
  - adding members, 290–292
  - looping structures for members, 293–294
  - removing members from, 292
- Colors form, 62–63
- ColumnHead property, 124
- ColumnHeads property, 122–124
- columns and rows. *See also* ADO (ActiveX Data Objects); SQL (Structured Query Language)
  - brackets for names of columns, 219
  - with list and combo boxes, 122–124
- combo boxes. *See* list and combo boxes
- comma-delimited files, 201–202
- command buttons. *See* button controls
- command objects, 60–65
- commands. *See* SQL (Structured Query Language)
- comments for VBA statements, 56
- compound If blocks, 84–87
- computed fields, 222–223
- concatenation, 56
  - operator, 68
- conditional operators. *See* operators
- conditions. *See also* If blocks; looping structures
  - compound conditions, 84–87
  - expressions and, 81
  - macros, adding to, 308
  - nested conditions, 83–84
  - Select Case structures for, 87–88
  - for SQL queries, 220–222
  - on UPDATE statements, 234
- connecting to database, 242–246
- Connection object, 285–287
- ConnectionObject procedure, 287–288
- Const statement, 69
- constants, 69
  - use of, 90
- Contacts template, 10
- Control Wizards icon, 60
- controls
  - bound controls, creating, 37–39
  - check boxes, 98–99
  - in Choose My Adventures
    - program, 265–266
  - common controls, 37–42
  - for Dice program, 176–177
  - for Fruit Stand program, 75–76
  - for Hangman program, 101–103
  - Hungarian Notation for, 42
  - Locals window, use of, 190
  - for Math Quiz program, 127
  - for Monster Dating Service
    - program, 295–296
  - naming conventions, 42
  - option buttons, 95–98
  - option group controls, 93–95
  - properties of, 36–37
  - referencing, 57
  - for Remote Connection program, 251
  - for Secret Message program, 149
  - switching between properties, 39
  - for Trivial Challenge program, 209

conversion

- functions, 142–144

- macros to VBA procedures, 313–315

COUNT function, 224–225

- ordinal position with, 255

counters for looping structures, 111

CREATE TABLE statement, 235–236

currency data type, 68

cursors, 248–249

## D

DAO (Data Access Objects), 241

Data Access Pages, 7

data assignment, 59

data files. *See also* file processing

- defined, 201

data types, 15

- common data types, 68

- list of, 29

- for variables, 67

- working with, 28–29

Database Enhancement dialog box, 6

database locks, 247

database normalization, 17–25

- 1st normal form, 21–22

- 2nd normal form, 22–23

- 3rd normal form, 23–25

databases. *See also* ADO (ActiveX Data Objects)

- connecting to, 242–246

- cursors, 248–249

- defined, 1–2

- Exclusive Use of, 315–316

- older formats, working with, 6

- ordinal position defined, 255

- performance considerations, 315–320

- remote database, connecting to, 243–244

Datasheet view, 11–12

- split forms option, 14

- tables, creating, 27

Date function, 139

date picker, 13

dates and times

- data fields, 29

- data type, 68

- formatting, 146–148

- functions, 139–142

- variables storing, 67

Day function, 140

DDL (data definition language), 215, 235–237

- ALTER TABLE statement, 236–237

- DROP statement, 237

Debug object, 200–201

debugging, 185–192

- breakpoints, 187–188

- error-handling, 196–201

- Immediate window, use of, 188–190

- Initialize event for, 280

- input validation, 192–196

- Locals window, use of, 190

- macros, 304

- stepping through code, 186–187

- Terminate event for, 280

- Watch window, use of, 190–192

declaration statements, 55–56, 67–68

declaring arrays, 168

Default parameter with input boxes, 91

DELETE statement, 234–235

deleting

- ADO records, 262–264

- collections, members from, 292

- ID fields, 27–28

- lines in forms, 62

- list and combo boxes, items from, 121–122

- scrollbars in forms, 62

DESC keyword, sorting in SQL with, 227–228

design time, 186

- Immediate window, use of, 188

Design view

- form properties, managing, 36–37

- forms, creating, 37

- for Fruit Stand program, 74–75

- macros, running, 307
- queries, creating, 43–44
- development environment, 333–335
- dialog boxes, 88–92
  - input boxes, 91–92
  - message boxes, 88–91
- Dice program, 176–183
- dice roll simulation, 124–126
- Dim keyword, 67
  - for arrays, 164
  - with user-defined types, 171
- Disk Not Ready error, 186
- DISTINCT function, 226
- Division by Zero error, 186
- division operator, 74
- DML (data manipulation language), 215
  - computed fields, 222–223
  - result set, 219
- Do Until loop, 112–113
- Do While loop, 111–112
- docking
  - Field List window, 37
  - options, 335
  - Property Sheet window, 37
- dot operator, 58–59
- doubles data type, 68
- drag-and-drop primary keys, 32
- DROP COLUMN keyword, 236–237
- DROP statement, 237
- dynamic arrays, 167–169

## E

- Edit List Items icon, 118
- Edit Relationships window options, 33
- elements
  - all elements, passing, 169–170
  - of arrays, 164, 169–170
  - of user-defined types, 173–176
- Else clause, 82
- ElseIf clause, 82–83

- embedded macros, 304
- empty quotes, 66
- encapsulation
  - of ADO library, 285
  - in OOP, 278
- End Functions statement, 158
- End Sub statement, 156–157
- End Type statement, 170–172
- Enforce Referential Integrity check box, 33
- entities
  - defined, 18
  - identifiers for, 21–22
- EOF function, 204
- equality testing, 59
- equals (=), 59
  - in expressions, 83
  - in SQL expressions, 221
- Err object, 199–200
- error handling, 196–201
  - for data files, 206–208
  - Debug object, 200–201
  - Err object, 199–200
  - for macros, 304, 311–313
- errors. *See also* debugging; error handling
  - ADO connection errors, troubleshooting, 246
  - common numbers/descriptions, 199
  - file access, error trapping for, 206–208
  - input validation, 192–196
  - off-by-one error, 164
  - reserved words/symbols, 337–343
  - trappable errors, 329–332
  - types of, 186
- event-driven paradigm, 49–50
- event procedures, 53
  - empty event procedures, 54
- events
  - in class modules, 280
  - template, 11
- exclamation mark operator, 58–59
- Exclusive Use of database, 315–316

- executable statements, 55–56
- Exit Function statement, 197
- Exit Sub statement, 120
  - with error-handling routine, 197
- exponentiation operator, 74
- expressions, 81–83
  - compound expressions, 84–87
  - in If blocks, 82
  - Watch window, use of, 190–192

## F

- Faculty template, 11
- Field List window, moving, 37
- fields, 26–31. *See also* file processing; queries
  - attributes of, 30
  - defined, 201
  - key symbol of, 30–31
  - multivalued fields, 14–15
  - ordinal position for, 254–255
  - primary key for, 30–31
- file formats, 16
- file I/O routines. *See* file processing
- File Not Found error, 186
- file processing, 201–208
  - access modes, 202–203
  - closing data files, 205
  - end of file, testing for, 204
  - error trapping, 206–208
  - opening sequential data files, 202–203
  - reading sequential data from file, 203–204
  - writing sequential data to file, 204–205
- Filename attribute, 202
- FIRST function, 225
- 1st normal form, 21–22
- For Each loop, 293–294
- For loop, 114–115
  - ListCount property, use of, 120
- ForeColor property, 64
- foreign keys
  - defined, 18
  - Enforce Referential Integrity check box, 33
- form class modules, 161
  - with ADO, 253
- Form object, 50
  - accessing properties of, 57–58
- Form tool, 35–42
- Form view split forms option, 14
- Format function, 144–148
- formatting, 144–148
  - dates and times, 146–148
  - numbers, 145–146
  - strings, 145
  - VBE format options, 334
- forms, 35–42. *See also* controls
  - event-driven paradigm and, 49–50
  - lines, deleting, 62
  - Me keyword, 58–59
  - naming conventions, 42
  - performance considerations, 317
  - properties of, 36–37
  - saving, 36
  - scrollbars, deleting, 62
- Forms collection, 57–58
- Fruit Stand program, 74–78
- function procedures, 155–156, 158
- Function statement, 158
- functions. *See also* specific functions
  - arrays, passing, 169–170
  - conversion functions, 142–144
  - date and time functions, 139–142
  - procedures, 53
  - SQL, built-in functions in, 223–227
  - string-based functions, 131–139

## G

- Getting Started with Microsoft Office Access
  - page, 25–26
- GoTo keyword, 197
- greater than (>)
  - in expressions, 83

- in SQL expressions, 221
- greater than or equal to ( $\geq$ )
  - in expressions, 83
  - in SQL expressions, 221
- GROUP BY clause, 229–230
- grouping. *See also* collections
  - in SQL, 229–230
- GUI (graphic user interface), 96

## H

- Hangman program, 101–107
- headers for columns, 122–123
- Help viewer, 16
- Hour function, 141–142
- Hungarian Notation, 42
- Hyperlink data fields, 29

## I

- IBM
  - ANSI SQL, use of, 215
  - SQL (Structured Query Language),
    - development of, 217
- ID fields, changing/removing, 28
- Idea results of Performance Analyzer, 319
- If blocks, 81–83
  - assignment statement and, 82
  - compound If blocks, 84–87
  - nested If blocks, 83–84
- image controls, 38
  - adding, 40
  - naming conventions, 42
- Immediate window, use of, 188–190
- indexes
  - form indexes, 58
  - performance considerations for, 317–318
- infinite loops, 110
- inheritance in OOP, 278
- Initialize event for class module, 280
- input boxes, 91–92

- input validation, 192–196
  - IsNumeric function, 192–194
  - range of values, checking, 194–196
- InputBox function, 91–92
- INSERT INTO statement, 232–233
- instances. *See* OOP (object-oriented programming)
- InStr function, 138–139
- Int function, 124
- integers data type, 68
- integrated development environment (IDE),
  - 51. *See also* VBE (Visual Basic Editor)
- Is keyword, 88
- IsNull function, 193–194
- IsNumeric function, 192–194
- Issues template, 11
- iterations. *See* looping structures

## J

- Java void functions, 155
- joins in SQL, 230–232

## K

- key symbol of field, 30–31
- keyboard shortcuts
  - with ampersand (&) character, 65
  - for Code window, 327–328

## L

- label controls, 38
  - binding, 37–38
  - naming conventions, 42
- label objects, 50, 60–65
- LAST function, 225
- Layout view, 12–13
  - Form tool in, 35
- LBound function, 165–166
- LCase function, 133



- Left function, 137
- left outer joins, 231–232
- Len function, 133–134
- less than (<)
  - in expressions, 83
  - in SQL expressions, 221
- less than or equal to (<=)
  - in expressions, 83
  - in SQL expressions, 221
- Light Switch program, 60–62
- limitations of Access 2007, 2–3
- LimitToList property, 120
- linefeed characters in message box, 90–91
- list and combo boxes, 116–124
  - adding items with, 117–120
  - columns and rows, managing, 122–124
  - deleting items from, 121–122
  - duplicate items, checking for, 118–120
  - Edit List Items icon, 118
  - LimitToList property, 120
  - naming conventions, 42
  - properties of, 116
- ListCount property, 120
- Load event, 117
- Locals window, use of, 190
- locking techniques, 247
- logic errors, 186
- long data type, 68
- Loop Until loop, 114
- Loop While loop, 113
- looping structures, 109–111. *See also* specific loops
  - for collection members, 293–294
  - counters, 111
  - introduction to, 109–110
  - nested looping structures, 167
  - in two-dimensional arrays, 167

## M

- Macro Builder, 304
- Macro Design view, 310
- MacroError object, 312–313
- macros, 303–315
  - conditions, adding, 308
  - converting to VBA procedures, 313–315
  - description of actions, 305–306
  - Design mode, running in, 307
  - embedded macros, 304
  - error handling, 304, 311–313
  - MacroError object, 312–313
  - OnError Action for, 312
  - security options, 304
  - stand-alone macros, 304–310
  - temporary variables, 304
  - troubleshooting, 311–313
  - VBA code, running, 54–55
- many-to-many relationship, defined, 18
- Marketing projects template, 11
- math operations, 73
- Math Quiz program, 126–129
- mathematical operators, 74
- matrices, two-dimensional arrays as, 166–167
- MAX function, 226
- MDB files, 6, 16
- MDE files, 16
- Me keyword, 58–59
- member variables, 282
- members. *See* collections
- memo data fields, 29
- memory requirements, 316
- menus, 52
- message boxes, 88–91. *See also* MsgBox
  - function
  - for errors, 198
- method procedures, 284–287
- methods
  - of collection objects, 291
  - for custom objects, 284
  - of objects, 50–51
- Microsoft Access SQL. *See* SQL (Structured Query Language)
- Microsoft ANSI SQL, use of, 215

## Microsoft Office

- Enterprise 2007 system requirements, 5
- Professional 2007 system requirements, 5
- Professional Plus 2007 system requirements, 5
- suites, 3–4
- Ultimate 2007 system requirements, 6
- Mid function, 137
- MIN function, 226
- Minute function, 141
- module-level variables, 71–72
- modules, 161–163. *See also* class modules
  - Watch expression value, 192
- Monster Dating Service program, 294–299
- Month function, 140
- MoveNext method, 254
- MovePrevious method, 254
- moving
  - drag-and-drop primary keys, 32
  - Field List window, 37
  - Property Sheet window, 37
- MsgBox function, 88–91
  - linefeed characters in, 90–91
  - return values, 89
- multiplication operator, 74
- multivalued fields, 14–15

## N

### names

- arguments, 159
- brackets for table/column names, 219
- error handling procedure, 196–197
- Hungarian Notation for, 42
- option controls, 95
- queries, 44
- tables, 27
- variables, 67, 70–71
- VBA procedures, 54
- natural joins, 231
- Navigation pane, 9–10

- nested If blocks, 83–84
- nested looping structures, 167
- new database, creating, 25–26
- Next keyword with For loop, 114
- nonvolatile memory areas, 70
- normalization process, 17–25
- Northwind 2007 database, 216
  - browsing/retrieving data in, 249–257
- not equal to <>
  - in expressions, 83
  - in SQL expressions, 221
- NOT keyword for SQL queries, 222
- Not operator, 86
- Now function, 142
- numbers
  - binary codes, 70
  - data fields, 29
  - formatting, 145–146
  - IsNumeric function, 192–194
  - random numbers, 124–126
  - Val function for converting, 69
  - variables storing, 67

## O

- object methods, 284–287
- object-oriented programming. *See* OOP (object-oriented programming)
- objects, 50–51. *See also* ADO (ActiveX Data Objects); macros; OOP (object-oriented programming)
  - accessing, 56–67
  - Me keyword, 58–59
  - variables storing, 67
- ODBC (Open Database Connectivity), 242. *See also* ADO (ActiveX Data Objects)
  - providers, 244
- off-by-one error, 164
- Office button, 9
- OLE DB providers, 244
- OLE Object data fields, 29

- On Error GoTo statement, 196–197
  - file access, error trapping for, 206
- one-to-many relationship, defined, 18
- one-to-one relationship, defined, 18
- OnError Action for macros, 312
- OOP (object-oriented programming), 50–51.
  - See also* collections
  - class modules, working with, 279–281
  - classes, use of, 278
  - custom objects, creating, 278–290
  - encapsulation in, 278
  - inheritance in, 278
  - instances, 279
    - creating and working with, 287–290
  - introduction to, 277–278
  - method procedures, 284–287
  - polymorphism in, 278
  - property procedures, 281–284
- Open function, 202–203
- opening
  - ADO recordset, 243–254
  - sequential data files, 202–203
  - VBE (Visual Basic Editor), 51–52
- operators
  - in expressions, 83
  - in SQL expressions, 221
  - in truth tables, 85
- Option Base statement, 73
- option buttons, 95–98
  - naming conventions, 42
  - Value property for, 96
- Option Compare Database statement, 72
- Option Explicit statement, 72–73
- option group controls, 93–95
  - check boxes with, 98
  - naming, 95
  - toggle buttons in, 99–101
- Option Group Wizard, 93–94
- option statements, 72–73
- Or operator, 85
  - for SQL queries, 222

- Oracle
  - ADO (ActiveX Data Objects) with, 242
  - ANSI SQL, use of, 215
- ORDER By clause, 227–228
- order of operations, 73–74
- ordinal position in ADO, 254–255
- outer joins, 231–232

## P

- parameters, 158–161
- parentheses
  - with arguments, 160
  - with dynamic arrays, 168
- percentage character, 146
- Performance Analyzer, 318–320
- performance considerations, 315–320
  - for code, 317
  - for forms, 317
  - for indexes, 317–318
  - for queries, 317–318
- polymorphism in OOP, 278
- Preserve keyword with dynamic arrays, 169
- primary keys
  - defined, 18
  - drag-and-drop, 32
  - Enforce Referential Integrity check
    - box, 33
  - for fields, 30–31
  - for tables, 27
- Private keyword
  - for arrays, 164
  - with user-defined types, 171–172
- procedure-level variables, 71–72
- procedures, 53–55
  - breakpoints in, 187–188
  - error-handling, 196–201
  - macros, converting, 313–315
  - user-defined procedures, 155–156
  - Watch expression value, 192
- Project Explorer window, 52

projects  
     template, 11  
     Watch expression value for, 192  
 Prompt parameter  
     with input boxes, 91  
     for MsgBox function, 88  
 properties  
     for Choose My Adventures program, 265–266  
     of collection objects, 291  
     for Dice program, 176–177  
     for Fruit Stand program, 75–76  
     for Hangman program, 101–103  
     of list and combo boxes, 116  
     for Math Quiz program, 127  
     for Monster Dating Service program, 295–296  
     of objects, 50–51  
     for Remote Connection program, 251  
     for Secret Message program, 149  
     for Trivial Challenge program, 209  
 Properties window in VBE (Visual Basic Editor), 53  
 Property Get procedure, 281–284  
 Property Let procedure, 281–284  
 property procedures, 53, 155, 281–284  
     private properties/procedures, 282  
 Property Set procedure, 281–284  
 Property Sheet window, 36–37  
     moving, 37  
     switching between control properties, 39  
 Public keyword  
     for arrays, 164  
     with user-defined types, 171–172  
 public variables, 71–72

**Q**

queries, 43–46. *See also* SQL (Structured Query Language)  
     limiting results of, 45

    naming, 44  
     performance considerations for, 317–318  
     running, 44–45  
     saving, 44  
 Quick Access toolbar, 9  
     Save icon, 27–28  
 quitting Access application, 64  
 quotes for concatenating string/text values in SQL, 259

## R

radio buttons. *See* option buttons  
 raise method error condition, triggering, 200  
 random numbers, 124–126  
 Randomize function, 124–126  
 range of values, checking, 194–196  
 RDBMS (Relational Database Management Systems), 49  
 RDO (Remote Data Objects), 241  
 read/write property, 283  
 reading sequential data from file, 203–204  
 recommendation results of Performance Analyzer, 319  
 records, 27. *See also* file processing  
     defined, 201  
 Recordset object, 246  
 recordsets in ADO, 246–264  
 ReDim keyword with dynamic arrays, 168–169  
 reference, arguments passed by, 158, 160  
 Relationships icon, 31  
 Remote Connection program, 251  
 remote database, connecting to, 243–244  
 Remove method for collections, 292  
 RemoveItem method, 121–122  
 reserved words/symbols, 337–343  
 resizing control property values, 41  
 result set  
     in ADO, 247  
     in SQL, 219  
 Resume keyword with error-handling routine, 198

- retrieving data with ADO, 249–257
- reuse, procedures for, 153–161
- Ribbon, 8–9
- rich text, 13
- Right function, 136
- right outer joins, 231–232
- Rnd function, 124–126
- rows. *See* columns and rows
- RunMacro method, 307
- runtime environment, 186–187
- runtime errors, 186

## S

- Sales pipeline template, 11
- saving
  - forms, 36
  - queries, 44
  - tables, 27–28
- scrollbars, deleting, 62
- Second function, 141
- 2nd normal form, 22–23
- Secret Message program, 148–151
  - standard module code for, 161–163
- security
  - macros, options for, 304
  - VBA code, running, 54–55
- Select Case structures, 87–88
- SELECT clause for joins, 230–232
- Select statements in DML (data manipulation language), 218–220
- SET keyword, UPDATE statement
  - with, 233–234
- shortcuts. *See* keyboard shortcuts
- Show Table window, 31–32
- single data type, 68
- single-dimension arrays, 164–166
- SizeMode property, 124
- software-locking techniques, 247
- sorting in SQL, 227–228
- split forms option, 14
- SQL (Structured Query Language), 43. *See also*
  - ADO (ActiveX Data Objects); DDL (data definition language); DML (data manipulation language)
  - built-in functions, 223–227
  - compound conditions for queries, 222
  - computed fields, 222–223
  - conditions for queries, 220–222
  - CREATE TABLE statement, 235–236
  - DELETE statement, 234–235
  - grouping in, 229–230
  - history of, 217
  - INSERT INTO statement, 232–233
  - introduction to, 215–218
  - joins in, 230–232
  - opening queries, 216
  - order of columns, changing, 219–220
  - ordinal position with, 255
  - outer joins, 231–232
  - Select statements, 218–220
  - sorting techniques, 227–228
  - syntax nomenclature for, 217–218
  - UPDATE statement, 233–234
- stand-alone macros, 304–310
- standard modules, 161–163
- statements, 55–56. *See also* assignment statement; specific statements
  - subprocedures executing, 157
- Static keyword for arrays, 164
- Step keyword with For loop, 115
- stepping through code, 186–187
- Str function, 142–143
- StrComp function, 134–136
  - output values for, 135
- stretching with SizeMode property, 124
- strings
  - character sequences, comparing, 134–136
  - data type, 68
  - formatting, 145
  - functions, 131–139
  - length of, 133–134
  - string data/string variables, comparing, 69

- Val function for converting, 69
- variables storing, 67
- structures in C language, 170
- Student Form, 321–322
- Students template, 11
- Sub statement, 156–157
- subprocedures, 53, 155–157
  - arrays, passing, 169–170
- subscript of two-dimensional arrays, 167
- subtraction operator, 74
- Suggestion results of Performance Analyzer, 319
- SUM function, 226–227
- symbols, reserved, 337–343
- syntax errors, 186
- system requirements, 4–6

## T

- Tabbed documents window, 9–10
- table relationships, 30–34
  - editing, 33
- TableName attribute, 235–236
- tables, 26–31. *See also* ADO (ActiveX Data Objects);
  - fields;
- SQL (Structured Query Language)
  - altering tables, 236–237
  - brackets for names of, 219
  - database normalization, 17–25
  - in DDL (data definition language), 235–236
  - naming, 27
  - saving, 27–28
  - two-dimensional arrays as, 166–167
- Tasks template, 11
- templates, 9–11
  - benefits of, 25
- Terminate event for class module, 280
- text boxes, 38
  - assigning properties to, 39–40
  - binding, 37–38

- clearing contents of, 66
- naming conventions, 42
- user input with, 65–67
- Val function with, 66
- text data fields, 29
- Text property for text boxes, 66
- Then keyword, 82
- 3rd normal form, 23–25
- Time function, 140–141
- times. *See* dates and times
- Title argument for MsgBox function, 88
- Title parameter with input boxes, 91
- To keyword with Select Case structure, 88
- toggle buttons, 99–101
- toolbars, 52
- Toolbox for list and combo boxes, 116
- trappable errors, 329–332
- Trivial Challenge program, 208–213
- troubleshooting. *See* error handling
- truth tables, 85
  - for Not operator, 86
  - for And operator, 85
  - for Or operator, 85–86
- turning on/off control wizards, 60
- two-dimensional arrays, 166–167
  - subscript of, 167
- Type statement, 170–172

## U

- UBound function, 165–166
- UCase function, 132–133
- Update method or ADO records, 258–261
- UPDATE statement, 233–234
- updating ADO records, 258–261
- user-defined procedures, 155–156
- user-defined types, 170–172
  - arrays of, 173–176
  - elements, managing, 173–176
  - variables, declaring, 172
- user interface, 6–9
  - requirements for, 35

**V**

- Val function, 66, 142
  - strings to numbers, converting, 69
- Value property
  - for option buttons, 96
  - for text boxes, 66
- values
  - arguments passed by, 158–160
  - subprocedures and, 157
- variables, 67–70. *See also* arrays
  - explicit declaration of, 72–73
  - initializing, 67
  - Locals window, use of, 190
  - naming conventions, 56, 70–71
  - scope, 71–72
  - of user-defined type, 172
- variants data type, 68
- VBA (Visual Basic for Applications), 49. *See also* Access VBA; macros; VBE (Visual Basic Editor)
  - statements, 55–56
- VBE (Visual Basic Editor), 51–56. *See also* Code window; debugging
  - components of, 52–53
  - development environment, 333–335
  - format options, 334
  - general options, 335
  - Me keyword, 58–59
  - opening, 51–52

- Project Explorer window, 52
- Properties window, 53
- viewing forms, 41
- Visual Basic. *See* ADO (ActiveX Data Objects)
- Visual C++. *See* ADO (ActiveX Data Objects)
- void functions, 155
- volatile memory areas, 70

**W**

- Watch Type, 192
- Watch window, use of, 190–192
- WeekDay function, 140
- WHERE clause, 220–221
  - for joins, 230–232
- wizards for button controls, 41
- words, reserved, 337–343
- Write function, 204–205
- writing sequential data to file, 204–205
- WYSIWYG in Layout view, 13

**Y**

- Year function, 140
- Yes/No data fields, 29

**Z**

- zero-based index in arrays, 164



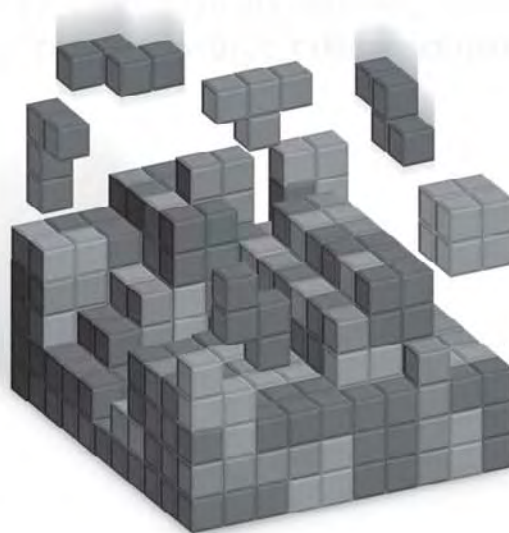
# the fun way to learn programming



## Let's face it.

C++, Java, and Perl can be a little intimidating. But now they don't have to be. The *for the absolute beginner* series gives students a fun, non-intimidating introduction to the world of programming. Each book in this series teaches a specific programming language using simple game programming as a teaching aid. All titles are

only \$29.99 and include source code on the companion CD-ROM or Web site.



### Programming for the Absolute Beginner

ISBN: 1-59863-374-0 ■ May 2007

Teaches programming fundamentals using a free implementation of BASIC called Just BASIC.



### ASP Programming for the Absolute Beginner

ISBN: 1-931841-01-2



### C++® Programming for the Absolute Beginner

ISBN: 1-931841-43-8



### Java™ Programming for the Absolute Beginner, Second Edition

ISBN: 1-59863-275-2



### JavaScript Programming for the Absolute Beginner

ISBN: 0-7615-3410-5



### Microsoft® Access VBA Programming for the Absolute Beginner, Third Edition

ISBN: 1-59863-393-7 ■ May 2007



### Microsoft® Excel VBA Programming for the Absolute Beginner, Third Edition

ISBN: 1-59863-394-5 ■ July 2007



### Microsoft® Visual Basic 2005 Express Edition Programming for the Absolute Beginner

ISBN: 1-59200-814-3



### Microsoft® Visual C++ 2005 Express Edition Programming for the Absolute Beginner

ISBN: 1-59200-816-X



### Microsoft® Visual C# 2005 Express Edition Programming for the Absolute Beginner

ISBN: 1-59200-818-6



### Microsoft® Windows Powershell Programming for the Absolute Beginner

ISBN: 1-59863-354-6



### Microsoft® WSH and VBScript Programming for the Absolute Beginner, Second Edition

ISBN: 1-59200-731-7



### PHP 5/MySQL Programming for the Absolute Beginner

ISBN: 1-59200-494-6



### Python Programming for the Absolute Beginner, Second Edition

ISBN: 1-59863-112-8



### Perl Programming for the Absolute Beginner

ISBN: 1-59863-222-1



*This page intentionally left blank*

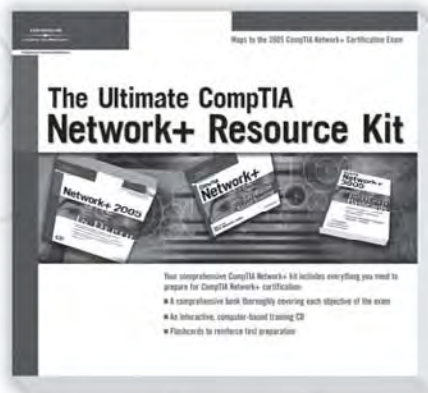
# COURSE TECHNOLOGY PTR...

## the ultimate source for all your certification needs.

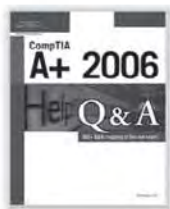
With step-by-step instructions and extensive end-of-chapter review questions, projects, and exercises, these learning solutions map fully to CompTIA certification exams. In-depth and well-organized—there isn't a better way to prepare!



**CompTIA A+ 2006 In Depth**  
ISBN: 1-59863-351-1 ■ \$39.99



**The Ultimate CompTIA Network+ Resource Kit**  
ISBN: 1-59863-349-X ■ \$59.99



**CompTIA A+ 2006  
Q&A**  
1-59863-352-X ■ \$19.99



**HTI+ In Depth**  
1-59200-157-2 ■ \$39.99



**Linux+ 2005  
In Depth**  
1-59200-728-7 ■ \$39.99



**Network+ 2005  
In Depth**  
1-59200-792-9 ■ \$39.99



**Network+ 2005  
Q&A**  
1-59200-794-5 ■ \$19.99



**Security+ In Depth**  
1-59200-064-9 ■ \$39.99

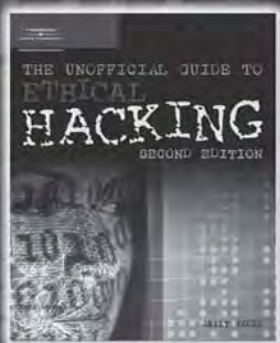
For more information on our offerings and to order, call **1.800.648.7450**,  
go to your favorite bookstore, or visit us at **www.courseptr.com**.

*This page intentionally left blank*

## hack•er (*n. Informal*)

1. One who is proficient at using or programming a computer; a computer buff.
2. One who programs enthusiastically—even obsessively—or who enjoys programming rather than just theorizing about programming.
3. A person who enjoys exploring the details of programmable systems and how to stretch their capabilities.
4. One who cracks a computer system for the sheer challenge of doing so!

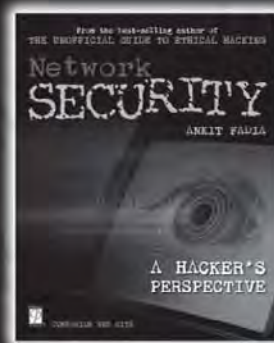
# Take your hacking skills to the limit!



**The Unofficial Guide to Ethical Hacking  
Second Edition**  
ISBN: 1-59863-062-8 ■ \$49.99  
Ankit Fadia



**Hacking Mobile Phones**  
ISBN: 1-59863-106-3 ■ \$24.99  
Ankit Fadia



**Network Security  
A Hacker's Perspective**  
ISBN: 1-59200-045-2 ■ \$49.99  
Ankit Fadia

Check out our complete collection of networking and security titles and order online at

# [www.courseptr.com](http://www.courseptr.com)

*This page intentionally left blank*

# Blackjack? Poker? Fantasy Baseball?

Wherever your online interests lie, we've got you covered...



## **Winning Secrets of Online Blackjack**

Catherine Karayanis  
ISBN: 1-59200-914-X  
U.S. List: \$19.99

## **Winning Secrets of Online Poker**

Douglas W. Frye and Curtis D. Frye  
ISBN: 1-59200-711-2  
U.S. List: \$19.99

## **Dominate Your Fantasy Baseball League**

David Sabino  
ISBN: 1-59200-684-1  
U.S. List: \$19.99

To view and order even more guides for today's digital lifestyle,  
visit us at [www.courseptr.com](http://www.courseptr.com) or call 800.648.7450.