

business solutions

Develop your Microsoft Access
expertise instantly with
proven techniques

Microsoft® Office Access 2007 VBA

Edit and debug your code

Use looping and conditional
statements

Understand Access's
object- and event-driven
architecture

Automate data entry

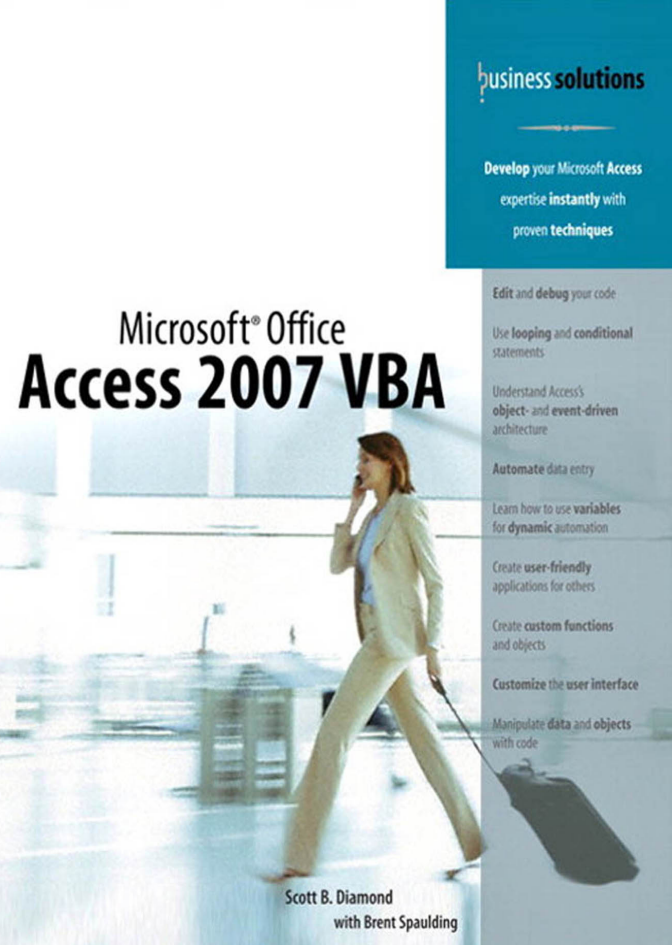
Learn how to use variables
for dynamic automation

Create user-friendly
applications for others

Create custom functions
and objects

Customize the user interface

Manipulate data and objects
with code



Scott B. Diamond
with Brent Spaulding

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

business solutions



Microsoft®
Office Access
VBA 2007

Scott B. Diamond

with Brent Spaulding

que®

800 E. 96th Street
Indianapolis, Indiana 46240

Contents at a Glance

Introduction	1
I The Building Blocks	
1 Advantages of Access and VBA	7
2 Using the Visual Basic Editor	11
3 Using Variables, Constants, and Data Types	25
4 Using Built-in Functions	37
5 Building Procedures	67
6 Conditional and Looping Statements	77
7 Working with Arrays	93
8 Object and Event-Driven Coding	101
9 Understanding Scope and Lifetime	117
II Working Within the User Interface	
10 Working with Forms	131
11 More on Event-Driven Coding	141
12 Working with Selection Controls	155
13 Working with Other Controls	177
14 Working with Reports	187
15 Menus, Navigation, and Ribbons	199
16 Application Collections	211
III Working with Data	
17 Object Models for Working with Data	227
18 Creating Schema	239
19 Data Manipulation	273
20 Advanced Data Operations	313
IV Advanced VBA	
21 Working with Other Data Files	323
22 Working with Other Applications	335
23 Working with XML Files	349
24 Using the Windows API	359
A A Review of Access SQL	373
Index	385

Microsoft® Office Access 2007 VBA

Copyright © 2008 by Que Publishing

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

ISBN-13: 978-0-7897-3731-1

ISBN-10: 0-7897-3731-0

Library of Congress Cataloging-in-Publication Data

Diamond, Scott B.

Microsoft Office Access 2007 VBA / Scott B. Diamond with Brent Spaulding.

p. cm.

ISBN 0-7897-3731-0

1. Microsoft Access. 2. Microsoft Visual Basic for applications. 3. Database management. I. Spaulding, Brent. II. Title.

QA76.9.D3D493 2008

005.75'65—dc22

2007044041

Printed in the United States of America

First Printing: November 2007 Corrections June 2010

This product is printed digitally on demand.

Trademarks

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

Microsoft is a registered trademark of Microsoft Corporation.

Warning and Disclaimer

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

Bulk Sales

Que Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside the United States, please contact

International Sales

international@pearsoned.com

This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- Go to <http://www.quepublishing.com/safarienabled>
- Complete the brief registration form
- Enter the coupon code SUJ2-L9NK-HUBD-WHKU-HCC6

If you have difficulty registering on Safari Bookshelf or accessing the online edition,

please email customer-service@safari-booksonline.com.

Associate Publisher

Greg Wiegand

Acquisitions Editor

Loretta Yates

Development Editor

Todd Brakke

Managing Editor

Patrick Kanouse

Senior Project Editor

Tonya Simpson

Copy Editor

Geneil Breeze

Indexer

Tim Wright

Proofreader

Paula Lowell

Technical Editor

Truitt L. Bradley

Publishing Coordinator

Cindy Teeters

Contents

Introduction	1
What's in the Book	1
This Book's Special Features	3
The Examples Used in the Book	4

I THE BUILDING BLOCKS

1 Advantages of Access and VBA	7
Understanding Where Access Fits in Office	7
Understanding Access Programming Choices	8
Macros	8
Using SQL	9
Using VBA	10
2 Using the Visual Basic Editor	11
First Look at the Visual Basic Editor	11
Explaining VBA Modules	15
Entering and Running Code	16
Debugging Code	17
Saving Code	19
Getting Help on Code	19
Coding Shortcuts	20
Good Coding Habits	21
Using a Naming Convention	21
Indenting	22
Documenting	23
3 Using Variables, Constants, and Data Types	25
Declaring Variables and Constants	25
Declaring Variables	25
Using <code>Option Explicit</code>	26
Naming Variables	27
Constants	27
Declaring Constants	28
VBA Data Types	28
Referencing Syntax	31
Case Study: Using Form References	32

4 Using Built-In Functions	37
What Are Functions?	37
Converting Data Types	38
Converting to a Boolean Data Type	39
Converting to a Date Data Type	40
Converting to an Integer Data Type	40
Converting to a String Data Type	41
Converting to a Variant Data Type	41
Converting Null Values	41
Working with Date Functions	42
Returning the Current Date	42
Performing Date Arithmetic	42
Determining the Difference Between Two Dates	43
Extracting Parts of Dates	44
Creating Dates from the Individual Parts	45
Creating Dates from String Values	46
Extracting a Specific Date or Time Portion	46
A Conversion and Date Example	46
Using Mathematical Functions	48
The Abs Function	48
The Int Function	48
The Rnd Function	49
A Mathematical Functions Example	49
Using Financial Functions	50
The Ddb Function	50
The Fv Function	50
The Pmt Function	51
The Rate Function	51
A Financial Functions Example	51
Manipulating Text Strings	52
The Asc Function	52
The Chr Function	53
The Case Functions	53
The Len Function	53
The Left, Right, and Mid Functions	54
The Replace Function	54
The Split Function	55
The Trim Functions	55
Formatting Values	55
Applying User-Defined Formats	57
Domain Aggregate Functions	59
The DLookup Function	60
The DCount Function	60
The DMax/DMin Functions	60

Using the Is Functions	61
Interaction	61
The MsgBox Function	61
The InputBox Function	63
Case Study: Add Work Days	64
5 Building Procedures	67
Types of Procedures	67
Subroutines	67
Functions	69
Assigning a Data Type to a Function	70
Public Versus Private	70
Passing Arguments	71
Using Optional Arguments and Default Values	71
Passing Arguments By Reference	72
Passing Arguments By Value	72
Error Handling	73
Using On Error Resume Next	73
Using On Error Goto	74
6 Conditional and Looping Statements	77
Introducing Flow of Control Statements	77
Using If...Then...Else	77
A Simple If Statement	78
More Complex Conditions	78
Including an Else Clause	79
Including an ElseIf Clause	79
Using Select Case	80
Using For...Next	81
Using the Step Clause	82
Other Ways to Set the Counter	82
Nesting For...Next Loops	83
Aborting a For...Next Loop	84
Using Do Loops	86
A Simple Do Loop	86
Do Loop Flavors	87
Aborting a Do Loop	88
Using GoTo	89
Case Study: Calculating Bonuses	89
7 Working with Arrays	93
Introducing Arrays	93
Declaring a Fixed-Size Array	94

Understanding an Array's Index	94
Using Option Base	95
Working with Array Elements	95
Assigning Array Elements	95
Using Array Element Values	96
Arrays with Multiple Dimensions	96
Expanding to Dynamic Arrays	98
About ReDim	98
Erase Statement	98
8 Object and Event-Driven Coding	101
Understanding Objects	101
Creating Objects in Code	102
Reading and Setting Object Properties	104
Invoking Methods	105
Using Collections	105
Working with an Object Model	106
Using the Object Model	107
Using References	108
The Object Browser	109
Creating Objects	110
Working with Events	113
9 Understanding Scope and Lifetime	117
Scope Explained	117
Procedure-Level Variables	117
Module-Level Variables and Constants	119
Public Variables and Constants	120
Measuring the Lifetime of a Variable or Constant	121
The Lifetime of a Procedure-Level Variable	122
The Lifetime of a Module-Level Variable	122
The Lifetime of a Public Variable	124
Using Static Variables	124
Case Study: Tracking the Current User	126

II WORKING WITHIN THE USER INTERFACE

10 Working with Forms	131
Opening and Closing Forms	131
Opening a Form	131
Passing Arguments Using OpenArgs	133
Closing a Form	134

The Form Module	135
Form and Control Properties	135
Form Events	137
Case Study: Adding to a Combo Box	138
11 More on Event-Driven Coding	141
Responding to Events	141
The Event Sequence for Controls	143
Focus Events	144
Data Events	146
Control Specific Events	148
The Event Sequence for Forms	149
Navigation Events	149
Data Events	150
Behind the Scenes: Data Buffers	151
The Event Sequence for Reports	151
Cancelling Events	152
Case Study: Validating Data	153
12 Working with Selection Controls	155
Selection Controls	155
Populating a List Control	155
A Filtering List Control	156
Adding to the List—Or Not	159
Updating a Table/Query List	163
Working with Option Groups	167
Working with MultiSelect Controls	169
Determining What Is and Isn't Selected	170
Case Study: Selecting Multiple Items	172
13 Working with Other Controls	177
Working with Text Boxes	177
Key Properties of Text Boxes	177
Tracking the Focus	179
Working with Check Boxes, Radio Buttons, or Toggle Buttons	181
Working with Subforms	182
Working with the Tag Property	183
Case Study: An Audit Trail	184

14 Working with Reports	187
An introduction to the Report Module and Events	187
Opening and Closing Reports	188
Opening a Report	188
Closing a Report	189
Passing Argument Using openArgs	190
Populating the Report	191
Applying a Filter and Sort Order	193
Handling Report-Level Errors	194
What to Do When There Is No Data	195
Working with Subreports	195
Case-Study: Product Catalog	196
15 Menus, Navigation, and Ribbons	199
Introducing Menus	199
Creating Form-Based Menus	199
Managing the Navigation Pane	204
Using Custom Ribbons	208
16 Application Collections	211
Understanding Application Collections	211
Retrieving Lists of Objects	213
Working with Object Properties	214
Programmatically Determining Dependencies	217
Case Study: Version Control	221

III WORKING WITH DATA

17 Object Models for Working with Data	227
What They Are and Why We Need Them	227
Data Access Objects	229
ActiveX Data Objects	232
ActiveX Data Objects Extensions for Data Definition	234
Object Model Selection	236
18 Creating Schema	239
Overview	239
Creating Databases	241
Using the DAO Object Model	241
Using the ADOX Object Model	243

Creating Tables	246
Using the DAO Object Model	246
Using the ADOX Object Model	249
Creating Fields	251
Using the DAO Object Model	252
Using the ADOX Object Model	255
Creating Indexes	259
Using the DAO Object Model	260
Using the ADOX Object Model	261
Creating Relationships	262
Using the DAO Object Model	263
Using the ADOX Object Model	265
Creating Queries	266
Using the DAO Object Model	267
Using the ADOX Object Model	269
Case Study: Updating an Existing Database Installation	271
19 Data Manipulation	273
Connecting to a Data Source	273
Using the DAO Object Model	273
Using the ADO Object Model	275
Opening a Recordset	278
Using the DAO Object Model	280
Using the ADO Object Model	284
Inserting Data	288
DAO'S Execute Method	288
ADO'S Execute Method	290
DAO'S AddNew Method	291
ADO'S AddNew Method	292
Finding Data	294
Limiting Records Retrieved	294
DAO'S FindFirst, FindNext, FindLast, and FindPrevious Methods	295
DAO'S Seek Method	297
Using DAO'S Filter Method	298
Using ADO'S Find Method	299
Using ADO'S Seek Method	301
Using ADO'S Filter Property	302
Updating Data	304
Deleting Data	306
DAO'S Delete Method for a Recordset Object	307
ADO'S Delete Method for a Recordset Object	307
Case Study: Backing Up Data	309

20 Advanced Data Operations	313
Creating Linked Tables	313
Data Definition Language	315
Schema Recordsets	317
Subqueries	319

IV ADVANCED VBA

21 Working with Other Data Files	323
Understanding File I/O	323
Opening Files	324
About mode	324
About access	324
About locking	325
Demonstrating Opening a File	325
Reading from Files	326
Using Input	326
Using Line Input #	327
Using Input #	328
Writing to Files	329
Printing to Files	331
Case Study: Using .ini Files	332
22 Working with Other Applications	335
Understanding Automation	335
Setting Object References	336
Creating Objects	338
Using CreateObject	338
Using GetObject	339
Using Early Binding	339
Working with Automation Servers	340
Talking To Excel	340
Talking to Word	342
Case Study: Using Excel Charts	345

23 Working with XML Files	349
Understanding XML	349
Using ExportXML	350
An Example of Exporting	352
Exporting a Web-Ready File	353
Exporting Related Data	353
Using ImportXML	354
An Import Example	355
24 Using the Windows API	359
Declaring API Calls	359
Using API Calls	360
API Calls You Can Use from Access	362
Check Whether an Application Is Loaded	362
Capture the Network Login ID	363
Retrieving the Name of the Program Associated with a Data File	364
Knowing When to Use the Windows API	364
Case Study: Capturing a Filename to Use for Processing	365
A A Review of Access SQL	373
Introduction to SQL	373
SQL Structure and Syntax	374
The SELECT Statement	376
The SQL Predicates	376
The SQL FROM Clause	377
The SQL WHERE Clause	378
The SQL ORDER BY Clause	379
The SQL GROUP BY Clause	379
The SQL HAVING Clause	380
The INSERT Statement	380
The UPDATE Statement	382
The SELECT INTO Statement	382
The DELETE Statement	383
Crosstabs	383
Index	385

About the Authors

Scott B. Diamond has been an information technology geek for more than 20 years. He has spent much of that time designing databases on various platforms. He started using Microsoft Access with Office 97 and has mastered all the subsequent versions. Besides developing database applications for the company where he's employed as an applications administrator, Scott also does freelance work, developing Access applications and consulting. He has always maintained that he's lucky his vocation is also his avocation, so he spends some of his free time helping people on web-based Q&A boards such as utteraccess.com (the premier support site for Access). He recently received Microsoft's MVP award for Access in acknowledgment of his contribution to the Access community. Scott, an avid bicyclist, lives on Long Island, New York, with his wife and daughter. You can reach Scott at AccessVBA@diamondassoc.com or visit his website, www.diamondassoc.com.

Brent Spaulding started writing applications about 20 years ago, generally focusing on data and data analysis. He has designed systems that have a wide range of focus: gymnastics class management, product assembly analysis, equipment fault logging, and manufacturing management systems. He has used Microsoft Access since version 2.0 and looks forward to using Access well into the future. In July 2007 Brent, who is employed in the automotive industry, received the Microsoft MVP award for Access, which recognizes his talent and contribution to the Access community. He spends much of his personal time learning and helping others on websites such as utteraccess.com, where he is known as [datAdrenaline](http://datAdrenaline.com). Brent lives in southern Indiana with his wife and children.

Dedication

To my wife and daughter, who have helped me realize one of my dreams.

—Scott B. Diamond

To my wife and our seven children who have loved me, encouraged me, and prayed with me throughout the entirety of this adventure.

—Brent Spaulding

Acknowledgments

A number of people contribute to a book like this, and this section is where I can reward their efforts by acknowledging them. First, thank you to Loretta Yates, who believed in me and gave me the opportunity to fulfill a longtime dream of mine. I also want to acknowledge the contributions of the editorial team: Todd Brakke and Geneil Breeze and technical editor, Truitt L. Bradly. Todd and Geneil made a major contribution to making sure my prose made sense to the reader. Truitt's contributions helped make this book possible as an editor and a friend. A special thanks also goes to Crystal Long for her encouragement and input.

Another thank you goes to my collaborator on this book, Brent Spaulding, who filled in some of the gaps in my knowledge and experience.

And finally, many thanks to my wife and daughter, who gave me support and, more importantly, put up with my hours of sitting in front of the computer to produce this book.

Personal note from Scott: I hope you enjoy using this book as much as I have enjoyed writing it. I truly believe that by following these lessons you will become an accomplished Access developer.

Personal note from Brent: Working with talented people like Scott and Truitt while I wrote Part III has been an incredible experience. The practical tips, advice, and code samples will provide a strong foundation as your knowledge and ability expand. With that said, grab a soda and a snack and start programming!

We Want to Hear from You!

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

As an associate publisher for Que Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the topic of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

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

Email: feedback@quepublishing.com

Mail: Greg Wiegand
Associate Publisher
Que Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.quepublishing.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

So, you've been using Access for a little while. Now that you have used Access to build databases for yourself, and maybe some friends and/or colleagues, you are ready for the next step: developing automated database applications. If you want quicker, easier, and more accurate data entry; faster searching; better reporting; the ability to manipulate data behind the scenes; and much more, this book is for you.

With *Microsoft Office Access 2007 VBA* we show you how to unleash the power of Access using Visual Basic for Applications (VBA). VBA is a superset of the Visual Basic programming language that combines Visual Basic command syntax and a rich assortment of functions with the capability to control objects in your application (hence the "A" in "VBA").

This book shows you VBA in action with real-world examples. We introduce you to programming and its use within Access and hold your hand every step of the way. With the information provided in this book you will explore commands, functions, properties, and methods and how to use them to make your applications dance to your tune and jump through hoops.

What's in the Book

This book isn't meant to be read from cover to cover, although you may find that you can't put it down! Instead, most of the chapters are set up as self-contained units that you can dip into and extract whatever nuggets of information you need at will. If you're a relatively new Access user, I suggest starting with the first one or two chapters in each of the book's four main parts to ensure you have a solid foundation in the fundamentals of working with data in Access tables by using queries, forms, and reports.

INTRODUCTION

IN THIS INTRODUCTION

What's in the Book	1
This Book's Special Features	3
The Examples Used in the Book	4



The book is divided into four main parts. To give you the big picture before diving in, here's a summary of what you find in each part:

- **Part I, “The Building Blocks”**—The nine chapters in Part I introduce you to the building blocks you use to build VBA modules. We start by explaining the advantages of using VBA. From there we introduce you to the Visual Basic Editor (VBE). This is where you enter, edit, and test all your code. In Chapter 3, “Using Variables, Constants, and Data Types,” we talk about the various ways you assign and identify data. And Chapter 4, “Using Built-in Functions,” moves on to a discussion of the many functions that Access and VBA provide. In Chapter 5, “Building Procedures,” the topic is procedures in their various forms and modules, the containers for your code. Chapter 6, “Conditional and Looping Statements,” gets into the meat of coding as we go over important syntax for branching using conditions and repeating code with looping. We follow that with a chapter on using arrays. And then Chapter 8, “Object and Event-Driven Coding,” explains how to launch your programs using object and event-driven coding. Part I closes with a chapter on scope, which covers lifetime and visibility of variables and procedures.
- **Part II, “Working Within the User Interface”**—This part shows you how to use VBA to create a great user interface. You learn to work with form and report design and understand their components such as controls and sections. We introduce you to the wide variety of different controls available and show you how to use them. You learn what events are and how they are triggered and discover how to create different menus and use VBA to navigate through your application. Part II ends with a discussion of collections and how to reference Access objects.
- **Part III, “Working with Data”**—This part of the book deals with working directly with data. You will learn the two main ways to get at data: Data Access Objects (DAO) and ActiveX Data Objects (ADO). DAO and ADO are presented in a comparative fashion. We cover these object models as they apply to the Access Connectivity Engine (ACE), the database engine under the hood of Access. ACE is the successor to the Joint Engine Technology (JET) database engine, so you will see JET and ACE terminology where appropriate. You will learn not only how to find, add, edit, and remove data, but also how to create and modify the structure of how data is organized (the schema). With DAO, ActiveX Data Objects extension (ADOX), and Data Definition Language (a subset of SQL), you can modify and create databases, tables, fields, indexes, relationships, and queries. This section also touches on some of the more advanced topics of database analysis, such as retrieving a list of all the relationships in your database or discovering all the computers that are connected to your database.
- **Part IV, “Advanced VBA”**—In this final section, you encounter working with other types of data files such as Excel spreadsheets, Word documents, and flat files. You also learn the basics of automation with other Office applications. Finally, we show you how to call on the Windows Application Programming Interface (API).
- **Appendix**—This includes a great reference on Structured Query Language (SQL) in its many different flavors and shows how to integrate SQL into your applications.

This Book's Special Features

Microsoft Office Access 2007 VBA is organized to give you a firm foundation for using VBA in a logical manner that builds your knowledge step by step. We have also made the book a functional reference for VBA techniques.

- **Steps**—Throughout the book, each Access task is summarized in step-by-step procedures.
- **Code lines**—Lines of VBA code, commands, and statements appear in a monospace typeface.
- **Required Text**—Any text you need to enter will be **boldfaced**.
- **Italics**—Technical terms being defined appear in italic, such as *RecordSet Property*.
- **Syntax**—Within code statements certain arguments will be *italicized* to denote where you will need to substitute values relevant to your task. Brackets ([]) are used to indicate optional arguments.

This book also uses the following elements to draw your attention to important (or merely interesting) information.

NOTE

Notes are used to provide sidebar information about the topic being discussed. They provide extra insights that help you understand the concepts being covered.

TIP

Tips tell you about Access methods that we have found to make coding with Access easier, faster, or more efficient.

CAUTION

Caution elements warn you about potential pitfalls waiting to trap your code, such as common errors that might occur, and how to avoid or fix them.

→ Cross-reference elements point you to related material elsewhere in the book.

CASE STUDY

You'll find case studies throughout the book based on an Inventory Tracking application. They are designed to show you how to apply what you've learned.

The Examples Used in the Book

All the tables, objects, and code samples referred to in this book can be found at <http://www.quepublishing.com>. There will be a folder for each chapter. The files build on the examples from the previous chapters. The Introduction file will be pretty bare bones and just have the objects that don't pertain to specific lessons we cover. You can build on the Introduction files or use the files that already have the examples coded for you.

This page intentionally left blank

Using Built-In Functions

4

What Are Functions?

Built-in functions are commands provided by Access and VBA that return a value. The value returned is dependent on the purpose of the function and the arguments, if any, passed to it. VBA is rich in functions that perform a variety of tasks and calculations for you. There are functions to convert data types, perform calculations on dates, perform simple to complex mathematics, make financial calculations, manage text strings, format values, and retrieve data from tables, among others.

Functions return a value, and most accept arguments to act on. A lot of your code will use functions to make calculations and manipulate data. You should familiarize yourself with the functions that VBA makes available to you, but don't expect to memorize their syntax. Between Intellisense and the VBA Help screens you can't go far off course, especially because Intellisense prompts you for each argument. If you need help understanding an argument, press F1 or look up the function in VBA Help.

Although this book was not meant to be a reference for VBA functions, this chapter explains many of the most used ones to give you an idea of VBA's power.

A point to remember when coding your functions: Be consistent in using data types. If you provide arguments of the wrong data type or assign a function to a different data type, you will cause an error.

IN THIS CHAPTER

What Are Functions?	37
Converting Data Types	38
Working with Date Functions	42
Using Mathematical Functions	48
Using Financial Functions	50
Manipulating Text Strings	52
Formatting Values	55
Domain Aggregate Functions	59
Using the Is Functions	61
Interaction	61
Case Study: Add Work Days	64



TIP

I don't know of any developer who knows every available function off the top of his or her head, so don't expect or think you need to. The more you code, the more you will remember, so feel free to use the references Microsoft provides. Use the Help option from the VBE menu to open the Developer Reference. In the search box type **functions list**, and one of the options is Functions (Alphabetical List). This gets you to a listing of all functions. Most of the function names are meaningful, so it shouldn't be difficult to find a function for the task you have.

NOTE

In this chapter we frequently use the term *expression*. In my use an expression can be as simple as a value or text string or as complex as a formula using multiple operators and functions. Just remember that an expression expresses a value.

Converting Data Types

At times you might find the need to import or link to data from external sources, or you might have to use data differently than the planned purpose. In such cases, the need may arise for you to convert from one data type to another. VBA includes several functions for this purpose. When you use a conversion function, the function returns the converted value but doesn't change the stored value.

→ For more on data types see, "VBA Data Types" p. 28.

This chapter goes over some of the more commonly used conversion functions. You can find a full list by opening the Developers Reference using the VBE Help menu and searching on *type conversion functions*.

- **CBool**—Converts a value to a Boolean data type.
- **CDate**—Converts a value to a Date data type.
- **CInt**—Converts a value to an Integer data type.
- **CStr**—Converts a value to a String data type.
- **CVar**—Converts a value to a Variant data type.

TIP

The most current conversion functions are prefixed with the letter C. It's better to use these functions in your conversions; however, you will also find included in VBA an older set of functions such as `Str` or `Val` for backward compatibility. The more current functions take your system's settings into account, whereas the older ones don't.

TIP

The `Val()` function has a use in addition to being a simple conversion function. It will return all numeric characters until it reaches a nonnumeric one. `CStr()` will return an error if you attempt to convert a string that contains nonnumeric data. For example, `Val("123abc")` will return the number 123 and `CInt("123abc")` will return a datatype mismatch error.

These functions have a simple syntax in common:

```
functionname(argument)
```

where *functionname* is the name of the function and *argument* is a value, variable, constant, or expression. The value of the argument is converted to a different data type depending on the function used, so it can be used elsewhere in your application. The value(s) used in the argument remain unchanged. It should be noted that not every data type can be converted to any other data type. The following sections explain the limitations.

Converting to a Boolean Data Type

A Boolean value is either `True` or `False`. The `False` value is either the number or character zero (0). Any other value is considered `True`. If the argument passed to the `CBool` function evaluates to a zero, `CBool` returns a `False`. If it evaluates to any other value, `CBool` returns a `True`. For example; all the following return a `True` because the arguments all evaluate to a nonzero value:

```
CBool("1")
```

```
CBool(1+0)
```

```
CBool(5)
```

```
CBool(-50)
```

Conversely, the following expressions return a `False` because each argument evaluates to zero:

```
CBool(0)
```

```
CBool("0")
```

```
CBool(15-15)
```

The argument passed to the `CBool` function must contain all numeric characters or operators. If you use alphabetic characters you get a type mismatch error. One place where using `CBool` becomes useful is in conditional statements. For example, you might need to determine whether two values match. In our Inventory application you might need to determine whether you are out of stock on an item. You could use the following expression, which would return a `False` if the incomings matched the outgoing:

```
CBool(Sum(Incoming) - Sum(Outgoing))
```

Converting to a Date Data Type

The `CDate` function converts any valid date/time value to a `Date/Time` data type. A valid date/time value can be either a number or a string that is formatted as a date or time. `CDate` determines valid date/time formats according to the regional settings you have chosen in Windows. You can use the following points to understand how dates are converted by `CDate`:

- If the argument is a numerical value, `CDate` converts the integer portion of the number according to the number of days since December 30, 1899. If the argument contains a decimal value, it's converted to a time by multiplying the decimal by 24 (for example, `.25` would be 6:00 a.m.).
- If the argument is a string value, `CDate` converts the string if it represents a valid date. For example; "1/16/51", "March 16, 1952", and "6 Jun 84" would all be converted to a date. However, "19740304" would result in a type mismatch error.
- Access recognizes dates from January 1, 100, to December 31, 9999. Dates outside that range result in an error.
- I recommend that you use four-digit years for clarity. However, Access will work with two-digit years. If you enter a year less than 30, Access assumes you want a date in the twenty-first century. If you use a year of 30 or higher, it is assumed to be a twentieth century date.
- Remember that the `/` is also the division operator and the `-` is used for subtraction. So, if you enter dates such as `12/3/04` you will get unexpected results. Entering `CDate(12/3/04)` returns December 31, 1899, because 12 divided by 3 divided by 4 = 1. So you need to put such dates within quotes.

Converting to an Integer Data Type

The `CInt` function takes a numeric or string value and converts it to an `Integer` data type. The *argument* is required and needs to represent a value within the range of `-32,678` to `32,767`. If the argument contains a decimal, Access rounds to the next whole number. A value of `.5` or higher is rounded up; anything lower is rounded down. Some examples of `CInt` functions follow:

```
CInt(10.5) = 11
```

```
CInt(25.333) = 25
```

```
CInt(10/3) = 3
```

```
CInt("1,000") = 1000
```

TIP

That last example illustrates one of the advantages of `CInt` over the older `Val` function. `CInt` uses the system's regional settings and, therefore, recognizes the thousands separator, whereas `Val` would convert "1,000" to 1.

The argument must evaluate to a numeric value; otherwise, it returns an error. If the argument evaluates to a value outside the range of the Integer data type, you get an overflow error.

Converting to a String Data Type

The `CStr` function converts just about every numeric value into a String data type. The required argument can be any variable, constant, expression, or literal value that evaluates to a string.

CAUTION

If you use a variable as the argument, make sure it's been initialized to a value. If you use `CStr` on an uninitialized variable, it returns a numeric value of 0.

Converting to a Variant Data Type

As I mentioned in the discussion of VBA data types in Chapter 3, “Using Variables, Constants, and Data Types,” the `Variant` data type is the most flexible because it can accept almost any value. With `CVar`, you can convert just about any numeric or text string to the `Variant` data type. With numeric values there is a constraint to the same range for the `Double` data type.

CAUTION

`CVar` should be used only when there is a doubt of the data type you are converting or when the data type isn't important.

Converting Null Values

If you try to use a Null value in many expressions, you will probably encounter an error. For example, the following expression results in a runtime error if either of the values contains a Null:

```
varTotal = ValueA * ValueB
```

To avoid such errors you can utilize the `Nz` function to convert the value to a non-Null. The `Nz` function uses the following syntax:

```
Nz(value, [valueifnull])
```

The `Nz` function works similarly to an Immediate If (IIF) function. The following expressions are functionally equivalent:

```
varTotal = IIF(IsNull(ValueA), 0, ValueA) * IIF(IsNull(ValueB), 0, ValueB)
```

```
varTotal = Nz(ValueA, 0) * Nz(ValueB, 0)
```

The `valueifnull` is an optional argument; it defaults to 0 or a zero-length string based on the value's data type.

Working with Date Functions

VBA has many functions that help you deal with dates. As long as you understand how Access stores Date/Time values, you should have no problem in working with date functions and values.

→ For a description of the Date/Time datatype see “VBA DataTypes,” p. 28.

In this section we go over most of the functions you use when dealing with dates.

Returning the Current Date

To return the current date (as stored on your system) use the following function, which gives you a number counting the days from 12/30/1899:

```
Date()
```

How this value is displayed depends on your regional settings. You can use the `Date$()` function to return a 10-character string representing the date. This string uses the format *mm-dd-yyyy*. The `Date()` function returns only the system date; if you need to include the time use the `Now()` function. As noted earlier, a date/time value is a number where the integer portion represents the date and the decimal portion represents the time. So the `Now()` function will return an integer and decimal that represents the current date and time. The `Now()` function defaults to displaying its value according to the regional settings on your PC. On my PC it displays 7/25/2007 5:06:34 PM.

Performing Date Arithmetic

Because dates are stored as numbers, you can do date arithmetic simply by adding or subtracting date values. However, VBA gives you a better way, the `DateAdd` function. Using this function, you can add 14 days, 14 weeks, 14 months, or 14 years to any date. Or you can find a time 60 hours earlier than the specified date and time.

The following is the syntax for `DateAdd`, where *interval* is a string that indicates the type of time period that you want to calculate:

```
DateAdd(interval, value, date)
```

Table 4.1 shows the various strings that can be entered as intervals. The *number* argument is a value or expression that specifies the number of intervals you want to calculate. The number used is an integer. If a decimal value is included, it's rounded to the nearest whole number, before performing the calculation. The *date* argument is a Date/Time value that is the base value to use in the calculation.

Table 4.1 Interval Settings

String Setting	Description
yyyy	Years
q	Quarters
m	Months
y	Day of year
d	Days
w	Weekdays
ww	Weeks
h	Hours
n	Minutes
s	Seconds

The *y*, *d*, and *w* intervals work interchangeably in the `DateAdd` function but have more meaning in other Date/Time functions. If the interval evaluates to a negative number, it returns an earlier date/time; a positive number returns a future date/time.

Determining the Difference Between Two Dates

The `DateDiff` function is used to determine the number of intervals between two date/time values. The following is the syntax for the `DateDiff` function, where *interval* is a string that indicates the type of time period used to calculate the difference between the first and second dates represented by *date1* and *date2* (refer to Table 4.1):

```
DateDiff(interval, date1, date2[,firstdayofweek[, firstweekofyear]])
```

Also included in the `DateDiff` function are two optional arguments: *firstdayofweek* and *firstweekofyear*. These are numerical constants that can be used to adjust the first day of a week or year when using the `DateDiff` function. Tables 4.2 and 4.3 show a list of the values for each constant. The default values are Sunday and January 1, respectively.

Table 4.2 First Day of Week Constants

Constant	Description	Integer Value
vbSunday	Sunday (the default)	1
vbMonday	Monday	2
vbTuesday	Tuesday	3
vbWednesday	Wednesday	4
vbThursday	Thursday	5
vbFriday	Friday	6
vbSaturday	Saturday	7

Table 4.3 First Week of Year Constants

Constant	Description	Integer Value
vbFirstJan1	Use the week in which January 1 occurs (the default).	1
vbFirstFourDays	Use the first week that has at least four days in the new year.	2
vbFirstFullWeek	Use the first full week of the new year.	3

The results from this function might not always be as expected:

- If *date2* falls before *date1*, the function yields a negative value.
- The `DateDiff` function calculates a year has passed when a new year falls between the two dates, even if there are fewer than 365 days. So when using 12/31 and 1/1 as *date1* and *date2*, respectively, the function returns a 1.

Figure 4.1 shows how these guidelines affect the function in the Immediate window.

Figure 4.1

The `DateDiff` function in action.

```

? DateDiff("d",#4/1/07#,#3/1/2007#)
-31
? DateDiff("yyyy",#12/31/07#,#1/1/2008#)
1

```

NOTE

Notice that the dates in Figure 4.1 are enclosed by *octothorpes* (#—commonly known as a *pound sign*). This character is used to delimit date values, similarly to the way quotation marks are used with text strings. Access may recognize a date value and automatically insert the octothorpes, but it's a good practice to insert them yourself.

Extracting Parts of Dates

The `DatePart` function is used to extract a portion of a date from a date value. A Date/Time data type contains several components that correspond to the intervals listed in Table 4.1. For example, the following expressions return the values 4, 1, and 2007, respectively:

```
DatePart("m",#4/1/2007#)
```

```
DatePart("d",#4/1/2007#)
```

```
DatePart("yyyy",#4/1/2007#)
```

The `DatePart` function uses the following syntax, where *interval* is a String value that defines the part of the date you want to extract and *date* is a valid Date/Time value (refer to Table 4.1 for a list of interval values):

```
DatePart(interval, date[,firstdayofweek[,firstweekofyear]])
```

Also included in the `DatePart` function are two optional arguments: *firstdayofweek* and *firstdayofyear*. These are numerical constants that can be used to adjust the first day of a week or year when using the `DatePart` function. Tables 4.2 and 4.3 show a list of the values for each constant. The default values are Sunday and January 1, respectively.

TIP

Because you can extract any portion of a Date/Time value, it makes the most sense to store a date or time once as a valid Date/Time value. For example, even if you need to show only the month and year for a date, it would make sense to store a full date even if it's just the first or last day of the month.

Creating Dates from the Individual Parts

With `DatePart` you extract part of a date; conversely, with the `DateSerial` function you combine the parts of a date to return a date value. The `DateSerial` function uses the following syntax, where *Year*, *Month*, and *Day* can be any expression that evaluates to an integer value that represents the respective date part:

```
DateSerial(Year, Month, Day)
```

There are some rules for each of the arguments:

- *Year* is required and must be equal to an integer from 100 to 9999.
- *Month* is required, and integers from 1 to 12 (positive or negative) are considered.
- *Day* is required, and integers from 0 to 31 (positive or negative) are considered.

The `DateSerial` function can take integer values outside those ranges and calculate the difference to return a date value. This makes it very powerful if you use expressions for the arguments. For example, the following expression returns June 5, 2008 because the 18th month from the start of 2007 is June:

```
DateSerial(2007, 18, 5)
```

Similarly, the following returns May 15, 2007, by using the 30 days in April and adding the difference of 15 days to the next month:

```
DateSerial(2007, 4, 45)
```

Although this shouldn't be used as a substitute for `DateAdd` or `DateDiff`, it can make it easy to create dates from calculated values.

TIP

The expression `DateSerial(2007, 5, 0)` returns 4/30/07. Using 0 for the *Day* value can then be used to get the last day of a month. If you use `DateSerial(Year, Month+1, 0)` you get the last day of the *Year* and *Month* used as arguments passed to the function.

Creating Dates from String Values

The `DateValue` function can be used to return a date value from a string value; it uses the following syntax, where *stringexpression* must conform to the formats used by the system's Regional settings:

```
DateValue(stringexpression)
```

The following three expressions return the date June 1, 2007:

```
DateValue("6/1/2007")
```

```
DateValue("June 1, 2007")
```

```
DateValue("1 Jun 07")
```

TIP

The functions `TimeSerial` and `TimeValue` perform similarly to the `DateSerial` and `DateValue` functions with time values.

Extracting a Specific Date or Time Portion

Table 4.4 lists several functions that return a specific portion of a date or time value. The syntax for these functions is simple:

```
Functionname(date/time)
```

Table 4.4 Date Component Functions

Function	Result
<code>Day(<i>date</i>)</code>	Returns the day of the month as an integer between 1 and 31
<code>Hour(<i>time</i>)</code>	Returns the hour as an integer between 0 and 23
<code>Minute(<i>time</i>)</code>	Returns the minute as an integer between 0 and 59
<code>Second(<i>time</i>)</code>	Returns the second as an integer between 0 and 59
<code>Month(<i>date</i>)</code>	Returns the month as an integer between 1 and 12
<code>Year(<i>date</i>)</code>	Returns the year as an integer between 100 and 9999

A Conversion and Date Example

Sometimes you might need to round a time value to the nearest quarter hour or hour. This example uses some of the conversion and date/time functions previously discussed to accomplish that task.

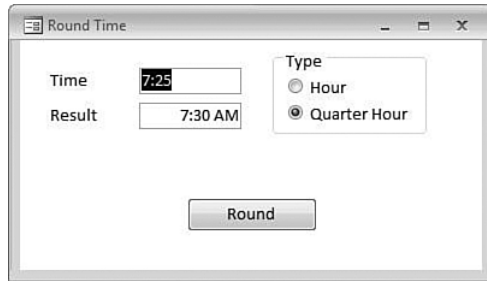
1. Create a blank form and put two text boxes on it. Label the boxes `txtTime` and `txtResult`.
2. Add an option group to the form with the options `Hour` and `Quarter Hour`. Name the group `optType`.
3. Add a button to the form (turn off the wizard first). Name the button `cmdRound`.
4. Set the Record Selectors and Navigation buttons to `No`. Set Scroll Bars to `neither`.
5. In the `On Click` event of the button use the following code:

```
Private Sub cmdRound_Click()  
Dim intHrs As Integer, intMin As Integer  
Dim dteTime As Date  
' convert entered time to Time value  
  
dteTime = CDate(Me.txtTime)  
'extract parts of time  
  
intHrs = DatePart("h", dteTime)  
intMin = DatePart("n", dteTime)  
  
If Me.optType = 1 Then 'test for nearest type  
    'Round to nearest hour  
    If intMin >= 30 Then  
        dteTime = DateAdd("h", 1, dteTime)  
        dteTime = DateAdd("n", -intMin, dteTime)  
    Else  
        dteTime = DateAdd("n", -intMin, dteTime)  
    End If  
Else  
    'Round to quarter hour  
    Select Case intMin  
        Case Is < 8  
            intMin = 0  
        Case 8 To 23  
            intMin = 15  
        Case 24 To 38  
            intMin = 30  
        Case 39 To 53  
            intMin = 45  
        Case Else  
            intHrs = intHrs + 1  
            intMin = 0  
        End Select  
    dteTime = TimeSerial(intHrs, intMin, 0)  
End If  
  
'Populate Result control  
Me.txtResult = dteTime  
  
End Sub
```

6. Save form as `frmRound` (see Figure 4.2).

Figure 4.2

The completed `frmRound` showing an example of input and result.



Using Mathematical Functions

VBA provides a rich, broad set of functions to perform mathematical and financial calculations. There are too many to cover in this section, so we provide an overview of the most commonly used functions.

The Abs Function

The `Abs` function returns the absolute value of a number, removing the sign. The following is the syntax for the `Abs` function, where *number* is any expression that evaluates to a numerical value:

```
Abs(number)
```

For example; this expression returns a 7:

```
Abs(-7)
```

The Int Function

The `Int` function removes any decimal value from a number, returning the integer portion. The function uses the following syntax, where *number* is any expression that evaluates to a numerical value:

```
Int(number)
```

For example; this expression returns 15 because it truncates the value, removing the decimal portion:

```
Int(15.9)
```

However, if the numerical value is negative, `Int` returns the nearest negative integer, so the following returns -16:

```
Int(-15.9)
```

Although seemingly the same, `Int` and `CInt` can't be used interchangeably. The `Int` function doesn't convert the data type of the argument. Using `CInt` is often the better option, but it doesn't always return the same result. So be careful in determining which one to use.

The Rnd Function

The `Rnd` function is used to generate a random number. It can be used with an optional argument represented as any valid numerical expression. The following is the syntax for the function:

```
Rnd(seed)
```

seed can be used to control the generated number as indicated in the following:

- If *seed* is a negative value, `Rnd` generates the same number.
- If *seed* is a positive number (other than 0) `Rnd` generates the next number in an internally determined sequence of numbers.
- If *seed* equals 0, `Rnd` generates the most recently generated number.
- If *seed* is omitted, `Rnd` generates the next number in an internally determined sequence of numbers.

The `Rnd` function generates a number in the range of 0 to 1, so if you need a whole number, you will have to multiply the generated value by a power of 10 and use the `Int` function to get your whole number.

TIP

Use the `Randomize` statement to reset the internal sequence so that `Rnd` generates apparently unique values that are repeated.

4

A Mathematical Functions Example

To illustrate mathematical functions, let's create a function to generate a number between 1 and 99.

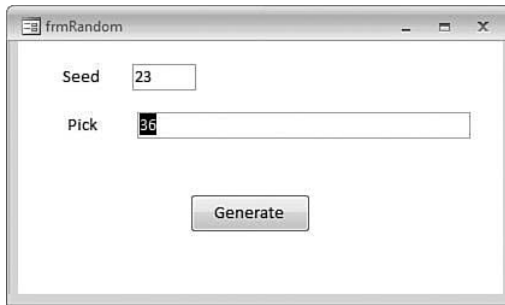
1. Create a blank form and put two text boxes on it. Label the boxes `txtSeed` and `txtPicks`.
2. Add a button to the form (turn off the wizard first). Name the button `cmdGenerate`.
3. Set Record Selectors and Navigation buttons to No. Set Scroll Bars to neither.
4. In the `On Click` event of the button use the following code:

```
Private Sub cmdGenerate_Click()  
'Generate number between 1 and 99  
Me.txtPicks = Int(Rnd(Me.txtSeed) * 100)  
End Sub
```

5. Save form as `frmGenerate` (see Figure 4.3).

Figure 4.3

The completed frmGenerate showing an example of a generated number.



Using Financial Functions

Financial functions are used to perform many standard financial calculations such as interest rates, annuity or loan payments, and depreciation. Following are some financial functions you might find useful.

The Ddb Function

The *Ddb* function calculates the depreciation of an asset for a specified time using the pre-defined double-declining balance method. The following is the syntax for this function, where *cost* is an expression representing the asset's opening cost and *salvage* is an expression that specifies the value of the asset at the end of *life*, an expression representing the term of the asset's lifespan.

```
Ddb(cost, salvage, life, period[, factor])
```

The *period* argument represents the time span for which the depreciation is being calculated. All these arguments use *Double* data types. There is an optional *factor* argument that specifies the rate of decline of the asset. If omitted, the double-declining method is used.

The FV Function

The *FV* function is used to calculate the future value of an annuity. The *FV* function returns a *double* data type and uses the syntax

```
FV(rate, nper, pmt[, pv [, type]])
```

where *rate* is an expression resulting in a *Double* data type that represents the interest rate per period, *nper* is an expression resulting in an *Integer* data type that represents the number of payment periods in the annuity, and *pmt* is an expression resulting in a *Double* value that specifies the payment being made for each period. There are two optional arguments, *pv* and *type*, which are *Variant* data types that specify the present value of the annuity and whether payments are made at the start or end of each period.

The Pmt Function

The `Pmt` function is used to calculate the payment for an annuity or loan. This function uses the syntax

```
Pmt(rate, nper, pv[, fv[, type]])
```

where *rate* is an expression resulting in a `Double` data type that represents the interest rate per period, *nper* is an integer expression that defines the number of payments to be made, and *pv* identifies the present value and is also a `Double` data type. There are two optional arguments, *fv* and *type*, which are `VARIANT` data types that represent the future value of the payments and whether payments are made at the start or end of each period.

The Rate Function

The `Rate` function is used to calculate the periodic interest rate for an annuity or loan. The syntax for this function is

```
Rate(nper, pmt, pv[, fv[, type[, guess]])
```

Where *nper* is an expression resulting in a `Double` data type that represents the number of period, *pmt* is an expression resulting in a `Double` data type that represents the payment per period, and *pv* is an expression resulting in a `Double` data type that defines the present value. There are also three optional arguments: *fv*, *type*, and *guess*, which identify the future value, determine whether payments are made at the start or end of each period, and allow you to give an estimate of the rate, respectively.

A Financial Functions Example

In keeping with the Inventory application theme of the sample file, this example looks at a scenario where you want to expand to cover a new product line. Because this new product line is from a new vendor, the vendor requires you to make a significant purchase the first time around. You don't have the \$10,000 to make the initial purchase, so you need to figure out different loan scenarios to see whether you can afford a loan.

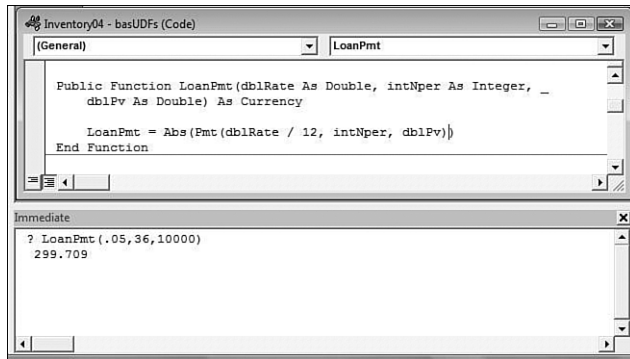
1. Open the `basUDFs` module or one of your own.
2. Enter the following procedure:

```
Public Function LoanPmt(dblRate As Double, intNper As Integer, _  
    dblPv As Double) As Currency  
    LoanPmt = Abs(Pmt(dblRate/12, intNper, dblPv))  
End Function
```

3. In the Immediate window enter the following statement and press Enter:
? LoanPmt(.05,36,10000)

Figure 4.4 shows the code and the result. This loan would cost you \$300 per month for 36 months. You can now try out different scenarios with combinations of rate and term to see what your payments might be.

Figure 4.4
The LoanPmt function
and its results.



Manipulating Text Strings

You use string functions to manipulate groups of text data. The following are some examples of where you might use string functions:

- Checking to see whether a string contained another string
- Parsing out a portion of a string
- Replacing parts of a string with another value

The following string functions help you do all these tasks and more.

The Asc Function

Every individual character can be represented by a number value. These values are listed in the American Standard Code for Information Interchange (ASCII). To return the ASCII value of a character use the following syntax, where *string* is an expression that results in a Text data type. It returns an integer value between 0 and 255.

`Asc(string)`

CAUTION

The Asc function reads only the first character of the string if there are multiple characters.

With any text string you must use apostrophes or quotation marks to define and delineate the text string; otherwise, Asc returns an error. However, if the string is a numeric, the delimiters can be eliminated. For example, the following two functions both return the value 51:

`Asc("3")`

`Asc(3)`

The Chr Function

The `Chr` function is the reverse of the `Asc` function. Whereas `Asc` returns the numerical value from character, `Chr` returns the character from a number. The following is the syntax for this function, where *charactercode* is an integer value between 0 and 255:

```
Chr(charactercode)
```

As you saw previously, the character 3 is represented by the number 51. So the following functions returns a 3:

```
Chr(51)
```

NOTE

The numbers 0–255 represent the values of characters according to the ASCII table. An example of that table can be found at <http://www.asciitable.com>.

TIP

Many of the string functions return a value as a variant of the `String` subtype. An alternative set of string functions add a `$` to the function name (for example, `Chr$`). These alternative functions return a literal string value. This provides a better performance because VBA doesn't have to evaluate the data type during processing.

The Case Functions

There is actually no case function. There are two functions, `LCase` and `UCase`, that can be used to change the case of a text string. They use the following syntax, where *string* is an expression that returns a string value. Both functions return the *string* in either lowercase or uppercase, respectively.

```
LCase(string)
```

```
UCase(string)
```

TIP

You can use the `UCase` function to convert entered data so that the data entry person doesn't have to concern himself with entering the proper case.

The Len Function

The `Len` function is used to determine the number of characters in a text string. This function uses the following syntax, where *string* is an expression that results in a `Text` data type. The function returns a long integer except where the string is `Null`, in which case it returns a `Null` value.

```
Len(string)
```

The Left, Right, and Mid Functions

Among the most used functions, these three return a portion of a string depending on the function and the arguments provided. All three result in a Variant Long subtype but support a \$ version, which forces a String data type.

The Left and Right functions use a similar syntax:

```
Left(string, length)
```

```
Right(string, length)
```

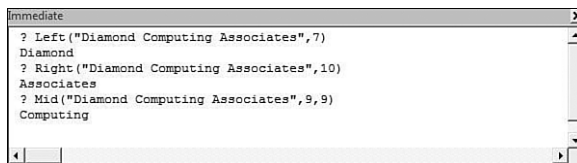
Here, *string* is an expression that results in a Text data type to be parsed and *length* is an expression that results in an Integer data type that specifies the number of characters from either the left or right end of the string to return.

The Mid function can parse a text string from any part of the string. It uses the following syntax, where *string* is a required argument that represents an expression resulting in a Text data type and *start* is a required argument that specifies where to start parsing the string:

```
Mid(string, start[, length])
```

An optional argument, *length*, specifies how many characters from the *start* point to parse. If *length* is omitted or is greater than the number of characters to the end of the string, all characters from *start* are returned. Figure 4.5 shows the three functions parsing various parts of the same string.

Figure 4.5
The Left, Right, and Mid functions parsing the same text.



The Replace Function

The Replace function is used to replace one or more characters within a string with a different character or characters. This function takes the following syntax, where *string* is an expression representing the text string to be searched, *stringtoreplace* is an expression representing the string to be searched for, and *replacementstring* represents the string you want in place of *stringtoreplace*

```
Replace(string, stringtoreplace, replacementstring[, start[, count[, compare]])
```

In addition, there are three optional arguments: *start*, which specifies where to start searching within the string; *count*, which specifies the number of replacements to process; and *compare*, which is a constant indicating the method used to compare *stringtoreplace* with *string*. Table 4.5 lists the constants that can be used.

Table 4.5 Comparison Constants

Constant	Value	Description
<code>vbUseCompareOption</code>	-1	Performs a comparison using the setting of the <code>Option Compare</code> statement.
<code>vbBinaryCompare</code>	0	Performs a binary comparison.
<code>vbTextCompare</code>	1	Performs a textual comparison.
<code>vbDatabaseCompare</code>	2	Microsoft Access only. Performs a comparison based on information in your database.

The Split Function

The `Split` function takes a delimited string and populates an array with the parts. The following is the syntax for the `Split` function, where *string* is a delimited string of values:

```
Split(string[, delimiter[, count[, compare]])
```

This is the only required argument. The first optional argument is *delimiter*, which specifies the delimiting character separating the values. If you omit *delimiter* a space is assumed to be the *delimiter*. The second optional argument is *count*, which limits the number of values parsed. For example, there might be five values separated by commas in the string, but a *count* argument of 3 parses only the first three values. The final optional argument is *compare*. See Table 4.5 for the comparison constants.

The Trim Functions

Three functions can be used to trim leading or trailing spaces from a string. All three use the same syntax, where *string* is an expression that results in a `Text` data type:

```
Trim(string)
```

```
LTrim(string)
```

```
RTrim(string)
```

The `Trim` function removes both leading and trailing spaces, `LTrim` removes the leading spaces, and `RTrim` removes the trailing spaces. All three functions return a `Variant String` subtype and support the `$` format to force a `Text` data type.

Formatting Values

Often data is stored differently from the way it's displayed on forms and in reports. The `Format` function is your tool to change how data is displayed. Access provides many predefined formats for you to use and allows you to customize your own formats. For example, a phone number might be stored as 10 digits but you can display it like (111) 222-3333 by applying a format. Another example are Date/Time values. As previously noted, they are stored as a `Double` number. However, the `Format` function can display the number in a variety of date or time formats.

CAUTION

Keep in mind that the `Format` function returns a `Variant String` subtype, which will probably be different from the original value's data type, and that the original data remains unchanged. This means that you should use `Format` only for display purposes; you don't want to use it in calculations.

The `Format` function uses the following syntax, where *expression* can be either a `String` or `Numeric` data type that results in the value you want to format:

```
Format(expression[, format[, firstdayofweek[, firstweekofyear]])
```

There are three optional arguments, the first of which determines how the data is formatted. The other two optional arguments, *firstdayofweek* and *firstweekofyear*, are numerical constants that can be used to adjust the first day of a week or year when using the `DatePart` function. Tables 4.2 and 4.3 show a list of the values for each constant. The default values are Sunday and January 1, respectively.

Tables 4.6 and 4.7 show some of the predefined formats you can use.

Table 4.6 Numeric Named Formats

Format	Example	Result
General Number	<code>Format(12345.6789, "General Number")</code>	12345.6789
Currency	<code>Format(12345.6789, "Currency")</code>	\$12,345.68
Fixed	<code>Format(0.1, "Fixed")</code>	0.10
Standard	<code>Format(12345.6789, "Standard")</code>	12,345.68
Percent	<code>Format(6789, "Percent")</code>	67.89%
Scientific	<code>Format(12345.6789, "Scientific")</code>	1.23E+03
Yes/No	<code>Format(0, "Yes/No")</code>	No
	<code>Format(3, "Yes/No")</code>	Yes
True/False	<code>Format(0, "Yes/No")</code>	False
	<code>Format(3, "Yes/No")</code>	True
On/Off	<code>Format(0, "Yes/No")</code>	Off
	<code>Format(3, "Yes/No")</code>	On

The result for `Currency` is based on the United States regional settings; if you use a different regional setting, the `Currency` format uses those settings. For the `Boolean` types a zero results in a `No`, `False`, or `Off` result. Any other value gives the opposite result.

Table 4.7 Date/Time Named Formats

Format	Example	Result
General Date	Format("04/01/07", "General Date")	4/1/2007
Long Date	Format("04/01/07", "Long Date")	Sunday April 1, 2007
Medium Date	Format("04/01/07", "Medium Date")	01-Apr-07
Short Date	Format("04/01/07", "Short Date")	4/1/2007
Long Time	Format('13:13:13', "Long Time")	1:13:13 PM
Medium Time	Format('13:13:13', "Medium Time")	1:13 PM
Short Time	Format('13:13:13', "Short Time")	13:13

Applying User-Defined Formats

Although the predefined formats listed in Tables 4.6 and 4.7 cover many situations, at times you'll need to create your own formats. You can use a number of special characters and placeholders to define your own formats. Tables 4.8, 4.9, and 4.10 list these formats.

Table 4.8 Numeric User-Defined Formats

Format	Explanation	Example	Result
0	Display actual digit or 0 for each 0 used. Rounds if more digits than shown.	Format(12.3456, "000.00000") Format(12.3456, "000.00")	012.34560 012.35
#	Display actual digit or nothing. Rounds if more digits than shown.	Format(12.3456, "###.#####") Format(12.3456, "###.##")	12.3456 12.35
%	Multiples by 100 and adds percent sign	Format(.3456, "##%")	35%
E- E+ e- e+	Display scientific notation.	Format(1.234567, "###E-###")	123E-2
- + \$ ()	Display a literal character.	Format(123.45, "\$####.##")	\$123.45
\	Display following character as a literal.	Format(.3456, "##.##\%")	.35%

Table 4.9 Date User-Defined Formats

Format	Explanation	Example	Result
d	Display day of month without leading zero	Format ("04/04/07", "d")	1
dd	Display day of month with leading zero where needed	Format ("04/04/07", "dd")	01
ddd	Display abbreviated day of week	Format ("04/01/07", "ddd")	Sun
dddd	Display full day of week	Format ("04/01/07", "dddd")	Sunday
ddddd	Display short date	Format ("04/01/07", "ddddd")	4/1/2007
dddddd	Display long date	Format ("04/01/07", "dddddd")	Sunday, April 1, 2007
m	Display month without leading zero	Format ("04/01/07", "m")	4
mm	Display month with leading zero	Format ("04/01/07", "mm")	04
mmm	Display abbreviated month name	Format ("04/01/07", "mmm")	Apr
mmm	Display full month name	Format ("04/01/07", "mmm")	April
q	Display quarter of year	Format ("04/01/07", "q")	2
h	Display hours without leading zero	Format ("13:13:13", "h")	1
hh	Display hours with leading zero	Format ("13:13:13", "hh")	01
n	Display minutes without leading zero	Format ("13:07:13", "n")	7
nn	Display minutes with leading zero	Format ("13:07:13", "nn")	07
s	Display seconds without leading zero	Format ("13:13:07", "s")	7
ss	Display seconds with leading zero	Format ("13:13:07", "ss")	07
tttt	Display 12-hour clock	Format ("13:13:13", "tttt")	1:13:13 PM
AM/PM	With other time formats displays either upper- or lowercase AM/PM	Format ("13:13:13", "hh:nn AM/PM")	1:13 PM
am/pm		Format ("13:13:13", "hh:nn am/pm")	1:13 pm
A/P	With other time formats displays either upper- or lowercase A/P	Format ("13:13:13", "hh:nn A/P")	1:13 P
a/p		Format ("13:13:13", "hh:nn a/p")	1:13 p
ww	Display the number of the week (1-54)	Format ("04/01/07", "ww")	14

Format	Explanation	Example	Result
w	Display the number of the day of the week	Format ("04/01/07", "w")	1
y	Display the day of the year (1–366)	Format ("04/01/07", "y")	91
yy	Display 2-digit year (00–99)	Format ("04/01/07", "yy")	07
yyyy	Display 4-digit year (0100–9999)	Format ("04/01/07", "yyyy")	2007

These formats can also be combined to display different date or time formats. The following are some examples:

Format ("04/01/07", "yyyymmdd") = 20070401

TIP This format is useful when exporting data to other formats and still maintaining chronological sort.

Format ("4/01/07", "mmm dd") = Apr 01

Format ("04/01/07", "mmm yyyy") = Apr 2007)

Table 4.10 String User-Defined Formats

Format	Explanation	Example	Result
@	Display actual character or space	Format ("VBA", "@@@@@")	VBA
&	Display actual character or nothing	Format ("VBA", "&&&&")	VBA
<	Display character as lowercase	Format ("VBA", "<<<<")	vba
>	Display character in uppercase	Format ("VBA", ">>>>")	VBA

Domain Aggregate Functions

Domain Aggregate functions are specific to Microsoft Access because they are used to retrieve data from tables. Because you can't assign the results of a query directly to a variable, you must use Domain Aggregate functions to retrieve that data. There are other ways besides Domain Aggregate functions that will be covered later in this book. The advantages of the Domain Aggregate functions are that they can accept a set of criteria to retrieve just the data needed. All the Domain Aggregate functions use a similar syntax, where *expression* is the name of the field in a table or query, *domain* is the name of the table or query, and *criteria* is a comparison to define which record to extract the value from:

Function("[*expression*]", "*domain*", *criteria*)

Notice that the *expression* is usually surrounded by quotes and brackets and that the *domain* is also surrounded by quotes. I'll list some of the more commonly used Domain Aggregate functions.

The DLookup Function

The DLookup function is used to retrieve a value from a single field in a table or query. The following example returns the last name of the contact for Supplier G from tblSuppliers in the Inventory application:

```
DLookup("[LastName]", "tblSuppliers", "[Company] = ' ' & "Supplier G" & "'")
```

The DLookup function retrieves that value from the first record matching the criteria. Because *Company* is a Text data type, you must concatenate the single quotes around the company name. If you are comparing a Numeric data type, no quotes are needed, and a Date/Time data type requires octothorpes (#) to delimit the value to be searched for.

The DCount Function

The DCount function is used to count the number of records in a table or query that match your criteria. An example of the DCount function follows:

```
DCount("*", "tblEmployees", "[Jobtitle] = 3")
```

This returns a result of 6 because there are six employees whose job title is Sales Representative.

The DMax/DMin Functions

The DMax and DMin functions return the highest or lowest values in the domain according to the criteria listed. An example of the DMin function follows:

```
DMin("[CreatedDate]", "tblTransactions")
```

This returns 3/22/2006 4:02:28 PM, which is the earliest transaction in the Transactions table.

TIP

The DMax function is often used to produce a sequential numbering system that can be dependent on some other value. For example, say you wanted to number each transaction for each employee and start the numbering each time a new employee is added. In such a case you could use the following code snippet in the After Update event of the control where the employee is selected on your form:

```
Me.txtIncrement =  
Nz(DMax("[Increment]", "tblTransactions", "[EmployeeID] = " &  
Me.cboEmployee), 0) + 1
```

This sets the control named txtIncrement to the highest value of the field Increment plus 1 for the selected employee. If no record for the employee is found, the NZ function causes the expression to return a 0, which is then incremented to 1.

Using the Is Functions

VBA provides a series of functions to help you trap errors that might arise from data type mismatches. These functions test a value to see whether it's a specified type.

- **IsArray**—Checks whether the value is an array
- **IsDate**—Checks whether the value is a Date/Time data type
- **IsEmpty**—Checks whether the value hasn't been initialized with a value
- **IsError**—Checks whether an expression results in an error
- **IsMissing**—Checks whether an optional argument has been passed to a procedure
- **IsNull**—Checks whether the value contains a Null
- **IsNumeric**—Checks whether the value is a number
- **IsObject**—Checks whether a variable contains a reference to an object

→ We cover arrays in more detail in Chapter 7, "Working with Arrays."

All these functions use the same syntax, where *value* is a value or expression being tested:

```
IsFunction(value)
```

The functions all return a Boolean data type, either `True` if the value meets the condition being checked or `False` if it doesn't.

Interaction

At times you need to provide information to the application's user or get information from them. This is interacting with the users. Two functions that can perform such an action are the `MsgBox` and `InputBox` functions.

The MsgBox Function

You use the `MsgBox` function to present information to users with an opportunity to respond to the information. You have control over how the message box appears and what response the user can make. The `MsgBox` function uses the following syntax, where *prompt* is the only required argument and represents a text string that constitutes the message presented by the message box:

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

The users can respond through a choice of one or more buttons. Table 4.11 lists various button options you can use. You can supply a string value for *title* that displays in the title bar of the message box. The other two optional arguments—*helpfile* and *context*—are seldom used and go together. The *helpfile* argument is a string that points to a help file to be used if the user clicks the message box's Help button. The *context* argument is a numeric value that specifies a number to be used within the help file. (Note: Creating help files is outside the scope of this book.)

Table 4.11 MsgBox Button Constants

Constant	Description	Integer Value
vbOkOnly	OK button	0
vbOKCancel	OK and Cancel buttons	1
vbAbortRetryIgnore	Abort, Retry, and Ignore buttons	2
vbYesNoCancel	Yes, No, and Cancel buttons	3
vbYesNo	Yes and No buttons	4
vbRetryCancel	Retry and Cancel buttons	5

Table 4.12 lists constants for the icons that can be displayed in the message box. You can display both icons and buttons using the following syntax:

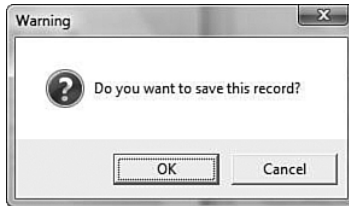
buttonconstant + iconconstant

As an example, the following function displays the message box shown in Figure 4.6. There are two buttons—OK and Cancel—and a question mark icon.

```
MsgBox("Do you want to save this record?", vbOKCancel + vbQuestion, "Warning")
```

Figure 4.6

A message box asking whether the user wants to save a record.

**CAUTION**

Besides the MsgBox function there is also a MsgBox action. The action displays the MsgBox without returning a value as a response.

Table 4.12 Icon Constants

Constant	Description	Integer Value
vbCritical	Critical message	16
vbQuestion	Warning message	32
vbExclamation	Warning message	48
vbInformation	Information message	64

When the user clicks one of the buttons, the function returns its value. Table 4.13 shows the values returned for each button.

Table 4.13 Button Values

Button	Returned Value	Integer Value
OK	vbOK	1
Cancel	vbCancel	2
Abort	vbAbort	3
Retry	vbRetry	4
Ignore	vbIgnore	5
Yes	vbYes	6
No	vbNo	7

The following code snippet is built around the message box function previously shown:

```
Private Function cmdSave_OnClick()
Dim strMsg As String
strMsg = "Do you want to save this record?"
If MsgBox("strMsg, vbOKCancel + vbQuestion,"Warning") = vbOK Then
    DoCmd.RunCommand acCmdSaveRecord
Else
    Me.Undo
End If
```

The InputBox Function

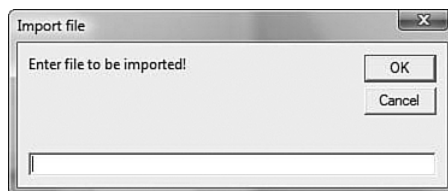
The Inputbox function displays a dialog box with a prompt that allows the user to enter a value that can then be assigned to a variable (see Figure 4.7). The following is the syntax for this function, where *prompt* is a String that displays a message for the user and is the only required argument:

```
InputBox(prompt[, title][, default][, xpos][, ypos][, helpfile, context])
```

The message is used to let the user know what data should be input. The *title* is a String that is displayed in the title bar of the window. The *default* is used to set a default value when the box opens. The *xpos* and *ypos* arguments allow you to precisely position the box in terms of the top and left of the screen. The *helpfile* and *context* arguments are the same as for the MsgBox.

Figure 4.7

An input box asking the user to enter a filename.



You usually use `InputBox` to retrieve a value from the user during processing of code. An example follows:

```
Dim strFilename As String
strFilename = InputBox("Enter file to be imported!", "Import file")
DoCmd.TransferText acExportDelim, , "Import", strFilename
```

TIP

I rarely use the `InputBox` function, preferring to use a form to allow the user to supply input. Using a form gives you much greater control over the input. With a form you can use interactive controls such as combo boxes or option groups to ensure that the correct data is entered. We'll deal with this more in later chapters.

CASE STUDY**Case Study: Add Work Days**

Sometimes you might need to figure a delivery or follow-up date for a shipment. You want to calculate such dates based on business days, not calendar days. The following function allows you to enter a start date and the number of business days and returns the date equal to that number of business days:

1. Open the UDFs module or a new one.
2. Enter the following code into the module:

```
Public Function AddWorkdays(dteStart As Date, intNumDays As Integer) As Date

    Dim dteCurrDate As Date
    Dim i As Integer

    dteCurrDate = dteStart
    AddWorkdays = dteStart
    i = 1

    Do While i < intNumDays
        If Weekday(dteCurrDate, vbMonday) <= 5 AND _
            IsNull(DLookup("[Holiday]", "tblHolidays", "[HolDate] = #" & _
                dteCurrDate & "#")) Then
            i = i + 1
        End If

        dteCurrDate = dteCurrDate + 1

    Loop
    AddWorkdays = dteCurrDate

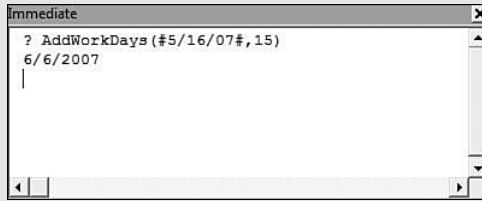
Exit_AddWorkDays:
End Function
```

3. Test the code by entering the following into the Immediate window. Figure 4.8 shows the results.


```
? AddWorkDays(#5/16/07#, 15)
```


Figure 4.8

The results from using the AddWorkDays function.



```
Immediate
? AddWorkDays(#5/16/07#,15)
6/6/2007
```

This page intentionally left blank

INDEX

Symbols

(octothorpes), 44
.ini files, case study, 332-333

A

aborting
Do loop, 88
For...Next loop, 84-86

Abs() function, 48

absolute value of a number, returning, 48

Access object class modules, 15

Access SQL. See SQL

accessing VBE, 11

AccessObject object, properties, 214-217

ACE (Access Connective Engine), 229

active control, 144

adding
to combo box, case study, 138-140
unlisted values to combo boxes, 159-167

AddNew method, data, inserting
ADO object model, 292-293
DAO object model, 291-292

ADO (ActiveX Data Objects) object model
AddNew method, inserting data, 292-293

connecting to data source, 275-278
Delete method, 307-308
Execute method, inserting data, 290-291
Filter property, 302-304
Find method, 299-301
object hierarchy, 232-234
Recordsets, opening, 284-288
Seek method, 301-302

ADOX (ActiveX Data Objects Extensions for Data Definition Language and Security), 234
databases, creating, 243-246
fields, creating, 255-259
indexes, creating, 261-262
object hierarchy, 235-236
queries, creating, 269-271
relationships, creating, 265-266
tables, creating, 249-251

AllForms collection, 213

annuities
payments, calculating, 51
periodic interest, calculating, 51

ANSI (American National Standard Institute), 373

API calls, 359-362
files, browsing, 365-372
FindExecutable API, 364
FindWindow API, 362-363
Network Login ID, capturing, 363

application case study, tracking current users, 126-127

application collections, 211-212

application lifetime, 122

arguments
delimiting, 72
ExportXML method, 350-352
Open function, 324-325
OpenArgs, 133
OpenDatabase method, 273
OpenRecordset method, 280
OpenReport method, 188-189
OpenArgs, 190-191
optional, 71-72
passing by reference, 72
passing by value, 72-73
specifying, 135
syntax, 71

arithmetic, performing on date values, 42-43

arrays, 93
dynamic, 98
Erase statement, 98-99
ReDim statement, 98
elements, 93, 95
assigning, 95-96
specifying, 96
fixed-size, declaring, 94
index values, 94
Option Base statement, 95
multidimensional, 96-98

Asc() function, 52

assigning

- array elements, 95-96
- data type to functions, 70
- values to object properties, 104

attachment fields, 229**audit trail case study, 184-185****automation, 335**

- objects, creating
 - CreateObject function, 338
 - early binding, 339-340
 - GetObject function, 339
- required components, 335
- type libraries, 336
- references, setting, 336-337

automation servers, 340

- transferring data to Excel, 340-342
- transferring data to Word, 342-345

B**backing up data, case study, 309-311****bands, events, 152****base table, 319****best practices for coding, 21-22**

- documenting, 23-24
- indenting, 22

bonuses, calculating, 89-92**Boolean data type, 30**

- controls, 181
- converting to, 39

branching, 77

- For...Next, 81-82
 - aborting, 84-86
 - counters, 83
 - nesting, 83-84
 - Step clause, 82
- If...Then...Else, 77-80

Break mode, 17**browsing files, 365-372****buffers, 151****button values, MsgBox() function, 62****Byte data type, 30****ByVal keyword, 72****C****calculating bonuses, case study, 89-92****cancelling events, 152-153****cascading combo boxes, 156****case studies**

- adding to combo box, 138-140
- audit trail, 184-185
- backing up data, 309-311
- date functions, 64
- form references, 32-35
- .ini files, 332-333
- product catalog, 196
- updating database installation, 271-272
- version control, 221-223

CBool() function, 39**CDate() function, 40****charts (Excel), creating from Access data, 345-347****Chr() function, 53****CInt() function, 40****class properties, implementing, 112-113****clauses**

- FROM, 377
- GROUP BY, 379
- HAVING, 380
- ORDER BY, 379
- SQL, 374
- WHERE, 378-379

Close method, 189-190**closing**

- forms, 134-135
- reports, 189-190

cmdPreview button, 141**code**

- debugging, 17-18
 - Break mode, 17
 - single step through, 18-19

saving, 19

Code Builder, 114**coding**

- best practices
 - documenting, 23-24
 - identing, 22
 - naming conventions, 21-22
- shortcuts, Intellisense, 20

collections, 105-106

- AllForms, 213
- Application, 211-212

combo boxes

- adding to, case study, 138-140
- cascading, 156
- unlisted values, adding, 159-167

comments, adding to code, 23-24**common action keywords (SQL), 374-375****comparing public and private keywords, 70****complete functions list, displaying, 38****complex data fields, 229****conditional branching, 77**

- For...Next statement, 81-82
 - aborting, 84-86
 - counters, 83
 - nesting, 83-84
 - Step clause, 82
- If...Then...Else statement, 77-80

connecting to data source, 273

- using ADO object model, 275-278
- using DAO object model, 273-275

constants, 25

- data types, 28-30
- declaring, 28
- for LockTypeEnum method, 281

- for OpenRecordset method, 280-281
 - intrinsic, 28
 - lifetime, measuring, 121
 - module-level, 119
 - public, 120-121
 - system-defined, 27
 - control-specific events, 148**
 - controls, 32, 143**
 - active control, 144
 - Boolean, 181
 - combo boxes
 - cascading, 156
 - unlisted values, adding, 159-167
 - event sequence, 144-146
 - focus, tracking, 179-180
 - list controls
 - populating, 155-156
 - records, filtering, 156-158
 - multiselect, 169-172
 - option groups, 167-169
 - subforms, 182
 - Tag property, 183
 - text boxes, 177
 - properties, 177-179
 - conversion functions, 38**
 - CBool(), 39
 - CDate(), 40
 - CInt(), 40
 - CStr(), 41
 - CVar(), 41
 - example of, 46-48
 - Nz(), 41
 - syntax, 39
 - Val(), 39
 - creating**
 - custom properties, 215-217
 - custom ribbons, 209
 - databases
 - ADOX object model, 243-246
 - DAO object model, 241-243
 - dates from individual parts, 45
 - dates from string values, 46
 - fields
 - ADOX object model, 255-259
 - DAO object model, 252-255
 - form-based menus, 199
 - indexes
 - ADOX object model, 261-262
 - DAO object model, 260-261
 - link tables, 313-314
 - objects, 102-104, 110-112
 - objects from automation servers
 - CreateObject function, 338
 - early binding, 339-340
 - GetObject function, 339
 - relationships
 - ADOX object model, 265-266, 269-271
 - DAO object model, 263-264, 267-269
 - tables
 - ADOX object model, 249-251
 - DAO object model, 246-249
 - crosstab queries, 383-384**
 - CStr() function, 41**
 - Currency data type, 30**
 - current date, returning, 42**
 - cursors, 279-280**
 - custom class modules, 15**
 - custom properties, creating, 215-217**
 - custom ribbons, 208-209**
 - CVar() function, 41**
- ## D
- DAO (Data Access Objects) object model, 229**
 - AddNew method, inserting data, 291-292
 - connecting to data source, 273-275
 - databases, creating, 241-243
 - Delete method, 307
 - Execute method, inserting data, 288-290
 - fields, creating, 252-255
 - Filter method, 298-299
 - Find methods, 295-297
 - indexes, creating, 260-261
 - object hierarchy, 229-232
 - queries, creating, 267-269
 - Recordsets, opening, 280-283
 - relationships, creating, 263-264
 - Seek method, 297-298
 - tables, creating, 246-249
 - data**
 - backing up, 309-311
 - deleting with Delete() method, 306
 - ADO object model, 307-308
 - DAO object model, 307
 - inserting
 - ADO object model, 290-293
 - DAO object model, 288-292
 - updating, 304-306
 - data events, 146-151**
 - data interface object models, 228-229**
 - data sources, 227**
 - connecting to, 273
 - ADO object model, 275-278
 - DAO object model, 273-275
 - data types, 28-30**
 - assigning to functions, 70
 - data validation, case study, 153-154**
 - Database Splitter, launching, 313**
 - databases, creating**
 - ADOX object model, 243-246
 - DAO object model, 241-243

- Date data type, 30**
 - converting to, 40
 - date functions, 42**
 - case study, 64
 - date component functions, 46
 - Date(), 42
 - DateAdd() function, 42-43
 - DateDiff() function, 43-44
 - DatePart() function, 44-45
 - DateSerial() function, 45
 - DateValue() function, 46
 - example of, 46-48
 - Date\$() function, 42**
 - DateAdd() function, 42-43**
 - DateDiff() function, 43-44**
 - DatePart() function, 44-45**
 - DateSerial() function, 45**
 - DateValue() function, 46**
 - Day() function, 46**
 - DCount() function, 60**
 - DdB() function, 50**
 - DDL (Data Definition Language), 315-316, 374**
 - keywords, 315
 - debugging code, 17-18**
 - Break mode, 17
 - single step through, 18-19
 - Decimal data type, 30**
 - decimal values, removing from numbers, 48**
 - declaring**
 - arrays, fixed-size, 94
 - constants, 28
 - variables, 25-26
 - DELETE statement (SQL), 383**
 - deleting data, Delete method, 306**
 - ADO object model, 307-308
 - DAO object model, 307
 - dependencies**
 - tracking, 219-220
 - viewing, 217-218
 - depreciation of assets, calculating, 50**
 - derived table, 319**
 - difference between dates, determining, 43-45**
 - Dim statement, 25**
 - displaying complete functions list, 38**
 - DLookup() function, 60**
 - DMax() function, 60**
 - DMin() function, 60**
 - DML (Data Manipulation Language), 374-375**
 - Do loop, 86-88**
 - DoCmd.Close method, closing forms, 134-135**
 - DoCmd.OpenForm method, opening forms, 131**
 - documenting code, 23-24**
 - domain aggregate functions, 59-60**
 - Double data type, 30**
 - double-declining balance method, 50**
 - dynamic arrays, 98**
 - Erase statement, 98-99
 - ReDim statement, 98
- ## E
-
- early binding, 339-340**
 - elements, 93**
 - of arrays, 95
 - assigning, 95-96
 - specifying, 96
 - specifying in multidimensional arrays, 96
 - embedded macros, 8**
 - EOF function, 327**
 - Erase statement, 98-99**
 - error handling**
 - On Error Goto statement, 74-75
 - On Error Resume Next statement, 73
 - report-level errors, 194
 - event procedures, 67, 113**
 - events, 113-114, 144-146, 151-152**
 - cancelling, 152-153
 - control-specific, 148
 - data events, 146-151
 - data validation, case study, 153-154
 - form events, 137-138
 - navigation events, 149
 - responding to, 141, 143
 - examples**
 - of conversion function, 46-48
 - of date function, 46-48
 - of financial function, 51
 - of mathematical function, 49
 - Excel**
 - charts, creating from Access data, 345-347
 - transferring data to, 340-342
 - Execute method, data, inserting**
 - ADO object model, 290-291
 - DAO object model, 288-290
 - exporting**
 - data to XML files, 350-353
 - HTML files, 353
 - ExportXML method, 350, 353**
 - arguments, 350-352
 - example use of, 352-353
 - expressions, 38**
 - extracting parts of dates, 44-46**
- ## F
-
- fields, 143**
 - creating
 - ADOX object model, 255-259
 - DAO object model, 252-255

- file handle, 323**
- file I/O, 323**
 - opening files, 324-326
 - printing to files, 331
 - reading from files, 326-329
 - writing to files, 329-330
- Filter method (DAO object model), 298-299**
- Filter property (ADO object model), 302-304**
- filtering**
 - list control records, 156-158
 - report data, 193
- financial functions, 50**
 - DdB() function, 50
 - example of, 51
 - Pmt() function, 51
 - Rate() function, 51
- Find method**
 - ADO object model, 299-301
 - DAO object model, 295-297
- FindExecutable API, 364**
- finding data in recordsets, 294**
 - Filter method
 - ADO object model, 302-304
 - DAO object model, 298-299
 - Find method
 - ADO object model, 299-301
 - DAO object model, 295-297
 - results, limiting, 294-295
 - Seek method
 - ADO object model, 301-302
 - DAO object model, 297-298
- FindWindow API, 362-363**
- fixed-size arrays, declaring, 94**
- flow of control statements, 77**
 - Do loop, 87-88
 - For...Next, 81-82
 - aborting, 84-86
 - counters, 83
 - nesting, 83-84
 - Step clause, 82
 - GoTo, 89
 - If...Then...Else, 77-80
- focus, 144**
- For...Next statements, 81-82**
 - aborting, 84-86
 - counters, 83
 - nesting, 83-84
 - Step clause, 82
- foreign key, 262**
- foreign table, 262**
- form events, 137-138**
- form modules, 135**
- form objects, setting properties, 136-137**
- form-based menus, 200-203**
 - creating, 199
- Format() function, 55**
 - date/time named formats, 56
 - numeric named formats, 56
 - user-defined formats, 57-59
- forms**
 - closing, 134-135
 - controls, 32, 143
 - active control, 144
 - event sequence, 144-146
 - events, 149
 - data events, 150-151
 - navigation events, 149
 - multiple items, selecting, 172-174
 - OpenArgs argument, 133
 - opening, 131-133
 - properties, 135
 - setting, 136-137
 - referencing, case study, 32, 34-35
 - subforms, 182
- FROM clause, 377**
- functions, 37, 69-70**
 - API calls, 362
 - FindExecutable API, 364
 - FindWindow API, 362-363
 - Network Login ID, capturing, 363
 - complete list of, displaying, 38
 - conversion, 38
 - CBool(), 39
 - CDate(), 40
 - CInt(), 40
 - CStr(), 41
 - CVar(), 41
 - example of, 46-48
 - Nz(), 41
 - syntax, 39
 - Val(), 39
 - CreateObject, 338
 - data type, assigning, 70
 - date functions, 42
 - case study, 64
 - date component functions, 46
 - Date(), 42
 - DateAdd(), 42-43
 - DateDiff(), 43-44
 - DatePart(), 44-45
 - DateSerial(), 45
 - DateValue(), 46
 - example of, 46-48
 - domain aggregate functions, 59
 - DCount(), 60
 - DLookup(), 60
 - DMax(), 60
 - DMin(), 60
 - EOF, 327
 - financial functions, 50
 - DdB() function, 50
 - example of, 51
 - Pmt() function, 51
 - Rate() function, 51
 - Format(), 55
 - date/time named formats, 56
 - numeric named formats, 56
 - user-defined formats, 57-59

GetObject, 339
 Is() functions, 61
 IsTable(), 319
 LOF, 327
 mathematical functions, 48
 Abs(), 48
 example of, 49
 Int(), 48
 Rnd(), 49
 Open(), 324
 arguments, 324-325
 string functions
 Asc(), 52
 Chr(), 53
 Lcase(), 53
 Left(), 54
 Len(), 53
 Mid(), 54
 Replace(), 54
 Right(), 54
 Split(), 55
 Trim(), 55
 Ucase(), 53
 user interaction
 InputBox(), 63-64
 MsgBox(), 61-62

G-H

generating random numbers, 49
 GoTo statement, 89
 GROUP BY clause, 379
 grouping objects within
 NavPane, 204-206
 handling report errors, 194
 HAVING clause, 380
 hiding toolbars, 15
 hierarchical structure
 of access objects, 229-232
 of ADO objects, 233-234
 of ADOX objects, 235-236
 Hour() function, 46
 HTML files, exporting, 353
 Hungarian convention, 21-22

I

I/O, 323
 opening files, 324-326
 printing to files, 331
 reading from files, 326-329
 writing to files, 329-330
 icon constants, MsgBox() function, 62
 identifiers, 31
 If...Then...Else statements, 77-80
 importing data into XML files, 354
 example of, 355-357
 ImportXML method, 354-357
 in place filtering, 302
 indenting code, 22
 index values, 94
 Option Base statement, 95
 indexes, creating
 ADOX object model, 261-262
 DAO object model, 260-261
 initiating procedures, 16-17
 InputBox() function, 63-64
 INSERT statement (SQL), 380-381
 inserting
 data
 ADO object model, 290-293
 DAO object model, 288-292
 modules, 15
 subroutines in modules, 68-69
 Int() function, 48
 Integer data type, 30
 converting to, 40
 Intellisense, 20
 interaction functions
 InputBox(), 63-64
 MsgBox(), 61-62

intrinsic constants, 28
 invoking methods, 105
 Is() functions, 61
 IsTable() function, 319

J-K-L

keywords
 public and private, comparing, 70
 SQL, 374-375
 subqueries, syntax, 320
 launching Database Splitter, 313
 LCase() function, 53
 Left() function, 54
 Len() function, 53
 lifetime, 117
 measuring, 121
 of module-level variables, measuring, 122-123
 of public variables, measuring, 124
 limiting results in searches, 294-295
 linked tables, creating, 313-314
 list controls
 combo boxes, adding unlisted values, 159-167
 populating, 155-156
 record, filtering, 156-158
 loans
 payments, calculating, 51
 periodic interest, calculating, 51
 local variables, 117
 LockTypeEnum method, constants, 281
 LOF function, 327
 Long data type, 30
 loops, Do Loop, 86-87
 aborting, 88

M

- macros, 8**
- mathematical functions**
 - Abs() function, 48
 - example of, 49
 - Int() function, 48
 - Rnd() function, 49
- measuring lifetime, 121**
 - of module-level variables, 122-123
 - of public variables, 124
- menu bar options (VBE), 12-13**
- menus, 199**
 - form-based, 199-203
- methods**
 - arguments, specifying, 135
 - Close, 189-190
 - DoCmd.Close, closing forms, 134-135
 - DoCmd.OpenForm, opening forms, 131
 - ExportXML, 350, 353
 - arguments, 350-352
 - example use of, 352-353
 - ImportXML, 354-357
 - invoking, 105
 - OpenForm, OpenArgs argument, 133
 - OpenReport, 188
 - arguments, 188-189
 - OpenArgs argument, 190-191
 - syntax, 105
- Microsoft Office Fluent UI, 208-209**
- Mid() function, 54**
- Minute() function, 46**
- module-level variables, 119**
 - lifetime, measuring, 122-123
- modules, 15**
 - subroutines, inserting, 68-69
- Month() function, 46**
- MsgBox() function, 61-62**
- multidimensional arrays, 96-98**
 - elements, specifying, 96
- multiselect controls, 169-172**

N

- naming conventions**
 - applying, 21-22
 - variables, 27
- navigation events, 149**
- NavPane, 204**
 - methods, 206-208
 - objects, grouping, 204-206
- nesting, 83-84**
- Network Login ID, capturing, 363**
- Now() function, 42**
- Null data types, converting, 41**
- Nz() function, 41**

O

- Object Browser, 109-110**
- Object data type, 30**
- object lifetime, 122**
- object models, 106-107**
 - ADO, 232
 - object hierarchy, 233-234
 - ADOX, 234
 - object hierarchy, 235-236
 - DAO, 229
 - object hierarchy, 229-230, 232
 - data interface object models, 228-229
 - referencing, 108
 - selecting, 236-237
- objects, 101-102**
 - collections, 105-106
 - controls, 32
 - creating, 102-104, 110-112
 - grouping within NavPane, 204-206
 - properties
 - events, 113-114
 - values, assigning, 104
 - referencing, 31-32
 - case study, 32-35
 - UI objects, 227
- octothorpes (#), 44**
- On Error Goto statement, 74-75**
- On Error Resume Next statement, 73**
- one-dimensional arrays, 96**
- Open function, 324**
 - arguments, 324-325
- OpenArgs argument, 133**
- OpenArgs argument (OpenReport method), 190-191**
- OpenDatabase method, 274-275**
 - arguments, 273
- OpenForm method, OpenArgs argument, 133**
- opening**
 - files, I/O method, 324-326
 - forms, 131-133
 - Recordsets, 278-280
 - ADO object model, 284-288
 - DAO object model, 280-283
 - reports, 188-189
 - schema recordsets, 317
- OpenRecordset method**
 - arguments, 280
 - constants, 280-281
- OpenReport method, 188**
 - arguments, 188-189
 - OpenArgs, 190-191
- operators, 31**
- Option Base statement, 95**
- Option Explicit statement, 26**

option groups, 167-169
optional arguments, 71-72
ORDER BY clause, 379

P

passing arguments
 by reference, 72
 by value, 72-73

payments for annuities, calculating, 51

periodic interest, calculating, 51

Platform SDK, 360

Pmt() function, 51

populating
 list controls, 155-156
 reports, 191-193

predefined formats, applying to data, 55-57

predicates (SQL), 376-377

primary key, 262

primary table, 262

printing to files, I/O method, 331

private keywords, 70

procedure-level variables, 117-119

procedures, 67
 arguments, 71
 optional, 71-72
 passing by reference, 72
 passing by value, 72-73
 functions, 69-70
 data type, assigning, 70
 initiating, 16-17
 public versus private, 70
 stubs, 16
 subroutines, 67
 inserting in modules, 68-69

product catalog case study, 196

properties
 AccessObject object, 214-215, 217

events, 113-114
 of classes, implementing, 112-113
 of controls, Tag property, 183
 of form objects, setting, 136-137
 of forms, 135
 of objects, values, assigning, 104
 of text boxes, 177-179
 read/write, 105
 write-only, 105

Public keyword, 70
public keywords, 70
public variables, 120-121
 lifetime, measuring, 124

Q-R

qualifiers, 32

queries
 creating
 ADOX object model, 269-271
 DAO object model, 267-269
 crosstab queries, 383-384

random numbers, generating, 49

Rate() function, 51

read/write properties, 105

reading from files, I/O method, 326-329

Recordsets
 cursors, 279-280
 data, deleting, 306
 Delete method (ADO object model), 307-308
 Delete method (DAO object model), 307
 data, finding, 294
 Filter method (DAO object model), 298-299
 Filter property (ADO object model), 302-304
 Find method (ADO object model), 299, 301

Find methods (DAO object model), 295, 297
 results, limiting, 294-295
 Seek method (ADO object model), 301-302
 Seek method (DAO object model), 297-298
 data, updating, 304-306
 opening, 278-280
 ADO object model, 284-288
 DAO object model, 280-283

ReDim statement, 98

referencing object models, 108

referencing objects, syntax, 31-35

relationships, creating
 ADOX object model, 265-266
 DAO object model, 263-264

Replace() function, 54

replacing characters within a string, 54

reports
 closing, 189-190
 error handling, 194
 event sequence, 151-152
 filtering data, 193
 opening, 188-189
 populating, 191-193
 subreports, 195
 troubleshooting, 195

responding to events, 141-143

ribbons, 208
 customizing, 209

Right() function, 54

Rnd() function, 49

S

saving code, 19
schema, 227, 239

- schema recordsets, 317-319
 - scheme definition files, 350
 - scope, 117
 - module-level, 119
 - procedure-level, 117-119
 - public, 120-121
 - Second() function, 46
 - Seek method
 - ADO object model, 301-302
 - DAO object model, 297-298
 - Select Case statement, 80-81
 - SELECT INTO statement (SQL), 382-383
 - SELECT statement (SQL), 376
 - FROM clause, 377
 - GROUP BY clause, 379
 - HAVING clause, 380
 - ORDER BY clause, 379
 - predicates, 376-377
 - WHERE clause, 378-379
 - selecting
 - files from File, Open dialog, 365-371
 - multiple form items, 172-174
 - object models, 236-237
 - setting breakpoints, 18
 - shortcuts for coding,
 - Intellisense, 20
 - Simonyi, Charles, 21
 - Single data type, 30
 - single stepping through code, 18-19
 - specifying
 - arguments for methods, 135
 - elements in multidimensional arrays, 96
 - specifying array elements, 96
 - split configuration,
 - implementing, 313
 - Split() function, 55
 - SQL (Structured Query Language), 9
 - clauses, 374
 - crosstab queries, 383-384
 - DDL, 315-316
 - keywords, 315
 - keywords, 374
 - common action
 - keywords, 374-375
 - statements, 374
 - DELETE, 383
 - INSERT, 380-381
 - SELECT, 376-380
 - SELECT INTO, 382-383
 - UPDATE, 382
 - subqueries, 319
 - keyword syntax, 320
 - standard modules, 15
 - Standard toolbar (VBE), 13-14
 - statements
 - Dim, 25
 - Option Explicit, 26
 - SQL, 374
 - DELETE, 383
 - INSERT, 380-381
 - SELECT, 376-380
 - SELECT INTO, 382-383
 - UPDATE, 382
 - static variables, 124-125
 - Steele, Doug, 365
 - String data type, 30
 - converting to, 41
 - string functions
 - Asc(), 52
 - Chr(), 53
 - LCase(), 53
 - Left(), 54
 - Len(), 53
 - Mid(), 54
 - Replace(), 54
 - Right(), 54
 - Split(), 55
 - Trim(), 55
 - UCase(), 53
 - structured files, 350
 - stub, 16
 - subforms, 182
 - subqueries, 319
 - keywords, syntax, 320
 - subreports, 195
 - subroutines, 67
 - inserting in modules, 68-69
 - syntax, 25
 - for Abs() function, 48
 - for arguments, 71
 - for Asc() function, 52
 - for Chr() function, 53
 - for conversion functions, 39
 - for DateAdd() function, 42
 - for DateDiff() function, 43
 - for DatePart() function, 44
 - for Dbd() function, 50
 - for Format() function, 56
 - for InputBox() function, 63
 - for Int() function, 48
 - for Is() functions, 61
 - for LCase() function, 53
 - for Left() function, 54
 - for Len() function, 53
 - for methods, 105
 - for Mid() function, 54
 - for MsgBox() function, 61
 - for Pmt() function, 51
 - for Rate() function, 51
 - for referencing objects, 31-32
 - case study, 32-35
 - for Replace() function, 54
 - for Right() function, 54
 - for Rnd() function, 49
 - for Split() function, 55
 - for static variables, 124
 - for Trim() function, 55
 - for UCase() function, 53
 - system-defined constants, 27
- ## T
- tables, creating
 - ADOX object model, 249-251
 - DAO object model, 246-249
 - Tag property, 183

text boxes, 177
properties, 177-179

toolbars, hiding, 15

tracking

current application users,
case study, 126-127
dependencies, 219-220

transferring data

to Excel, 340-342
to Word, 342-345

Trim() function, 55

troubleshooting reports, 195

two-dimensional arrays, 93

type libraries, 336

references, setting, 336-337

U

UCase() function, 53

UDFs (user-defined functions), 69-70

UI objects, 227

UPDATE statement (SQL), 382

updating

data, 304-306
database installation, case study, 271-272

user-defined formats,

applying to data, 57-59

Userforms, 14

V

Val() function, 39

values, formatting, 55

date/time named formats,
56
numeric named formats, 56
user-defined formats, 57-59

variables, 25

data types, 28-30
declaring, 25-26
lifetime, 117
measuring, 121
local, 117

module-level, 119

lifetime, measuring,
122-123

naming, 27

procedure-level, 117-119

public, 120-121

lifetime, measuring, 124

scope, 117

static, 124-125

Variant data type, 31

converting to, 41

VBA, 10

VBE (Visual Basic Editor), 11

menu bar options, 12-13
Standard toolbar, 13-14

version control, case study, 221, 223

viewing dependencies, 217-218

W

WHERE clause, 378-379
syntax, 374

window handles, 362

Windows API, 359

API calls, 359-362
FindExecutable API, 364
FindWindow API,
362-363
Network Login ID,
capturing, 363
files, browsing, 365-372
when to use, 364

Word, transferring data to, 342-345

write-only properties, 105

writing to files, I/O method, 329-330

X-Y-Z

XML, 349

exporting data to XML
files, 350-353
example of, 352-353
importing, 354-357
structured files, 350
supported file types,
349-350

Y

Year() function, 46