

Web Downloads Available

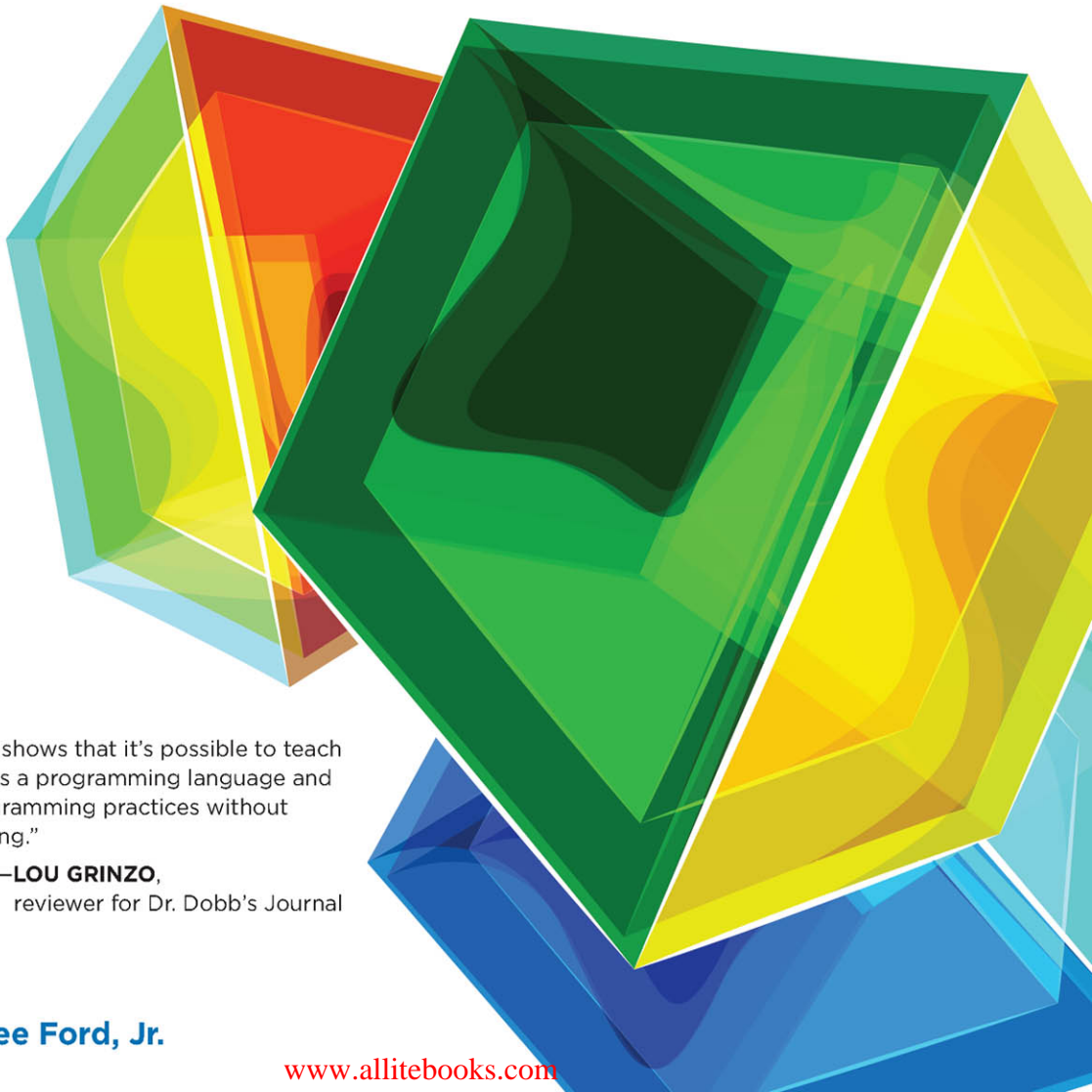


Microsoft®

WSH and VBScript Programming

for the Absolute Beginner

Fourth Edition



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

—**LOU GRINZO**,
reviewer for Dr. Dobb's Journal

Jerry Lee Ford, Jr.

www.allitebooks.com

Microsoft® WSH and VBScript Programming for the Absolute Beginner, Fourth Edition

Jerry Lee Ford, Jr.

Cengage Learning PTR



Australia, Brazil, Japan, Korea, Mexico, Singapore, Spain, United Kingdom, United States

**Microsoft® WSH and VBScript Programming
for the Absolute Beginner, Fourth Edition**
Jerry Lee Ford, Jr.

**Publisher and General Manager,
Cengage Learning PTR:**
Stacy L. Hiquet

Associate Director of Marketing:
Sarah Panella

Manager of Editorial Services:
Heather Talbot

Senior Marketing Manager:
Mark Hughes

Senior Acquisitions Editor:
Mitzi Koontz

Project and Copy Editor:
Kate Shoup

Technical Reviewer:
Zac Hester

Interior Layout:
Shawn Morningstar

Cover Designer:
Mike Tanamachi

Indexer:
Larry Sweazy

Proofreader:
Sam Garvey

© 2015 Cengage Learning PTR.

CENGAGE and CENGAGE LEARNING are registered trademarks of Cengage Learning, Inc., within the United States and certain other jurisdictions.

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at
Cengage Learning Customer & Sales Support, 1-800-354-9706

For permission to use material from this text or product,
submit all requests online at **cengage.com/permissions**

Further permissions questions can be emailed to
permissionrequest@cengage.com

Microsoft is a registered trademark of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.

Cover images: © antishock/Shutterstock.com

All images © Cengage Learning unless otherwise noted.

Library of Congress Control Number: 2014933749

ISBN-13: 978-1-305-26032-0

ISBN-10: 1-305-26032-5

eISBN-10: 1-305-26033-3

Cengage Learning PTR
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at:
international.cengage.com/region

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **cengageptr.com**.

Visit our corporate Web site at **cengage.com**.

*To my father;
to my children, Alexander, William, and Molly;
and to my beautiful wife, Mary.*

Acknowledgments

There are a number of individuals to whom I owe many thanks for their help and assistance in the development of the fourth edition of this book. I would start by thanking Mitzi Koontz, who served as the book's acquisitions editor. Special thanks also go out to Kate Shoup for serving as the book's project editor and copy editor. I also want to thank Zac Hester for all his valuable technical input and advice. In addition, I would like to thank everyone else at Cengage Learning for all their hard work.

About the Author

Jerry Lee Ford, Jr. is an author, educator, and IT professional with more than 24 years of experience in information technology, including roles as an automation analyst, technical manager, technical support analyst, automation engineer, and security analyst. He is the author of 38 books and co-author of two additional books. His published works include *Microsoft Windows PowerShell Programming for the Absolute Beginner*; *Microsoft Visual Basic 2008 Express Programming for the Absolute Beginner*; *HTML, XHTML, and CSS for the Absolute Beginner*; *XNA 3.1 Game Development for Teens*; and *VBScript Professional Projects*. Jerry has a master's degree in business administration from Virginia Commonwealth University in Richmond, Virginia, and has more than five years of experience as an adjunct instructor teaching networking courses in information technology.

Table of Contents

Introduction xv

Part I

Introducing the WSH and VBScript **1**

Chapter 1 Getting Started with the WSH and VBScript **3**

Project Preview: The Knock Knock Game. 3

What Is the WSH? 4

 WSH Scripting Engines 5

 Selecting a WSH Script Execution Host. 6

 Introducing the WSH Core Object Model. 6

 How Does the WSH Compare to Windows Shell Scripting? 7

 WSH Versus Windows PowerShell 8

 Understanding How the Windows Shell Works 9

 How Does It All Work? 15

 Operating System Compatibility 15

 How Do You Install It? 16

 How Does It Work with VBScript? 17

 What Other Scripting Languages Does the WSH Support? 20

Introducing VBScript 21

 VBScript Capabilities. 22

 VBScript's Roots. 22

 VBScript's Cousins: Visual Basic and VBA 23

Back to the Knock Knock Game. 25

 Designing the Game. 25

 The Final Result 29

Summary 29

Chapter 2 An Introduction to the Windows Script Host	31
Project Preview: The Rock, Paper, and Scissors Game	31
Examining Scripting Environments	32
An Examination of WSH Components	33
A Quick Introduction to the WSH Core Object Model	34
Working with the WScript Object	35
Configuring WSH Execution Hosts	35
Configuring WScript.exe and CScript.exe Command-Line Execution	36
Configuring WScript.exe Desktop Execution	37
Overriding Command-Line Host Execution Settings	38
Customizing WScript.exe Settings for Individual Desktop Scripts	39
Enabling and Disabling the Windows Script Host	41
Back to the Rock, Paper, and Scissors Game	43
Designing the Game	43
The Final Result	46
Summary	47

Part II

Learning VBScript and WSH Scripting **49**

Chapter 3 VBScript Basics	51
Project Preview: The Math Game	51
VBScript Statements	53
VBScript Syntax Rules	54
Reserved Characters	56
Adding Comments	57
Mastering the VBScript Object Model	58
Working with VBScript Run-Time Objects	59
Properties	61
Methods	62
Using VBScript Run-Time Objects in Your Scripts	64
Examining Built-in VBScript Functions	66
Demo: The Square-Root Calculator	66
Demo: A New and Improved Square-Root Calculator	67
Displaying Script Output	68
The WScript Object's Echo() Method	69
The WshShell Object's Popup() Method	69
The VBScript InputBox() Function	71
The VBScript MsgBox() Function	72

Back to the Math Game	74
A Quick Overview of the WshShell Object's SendKeys() Method	74
Designing the Game	77
The Final Result	81
Summary	81
Chapter 4 Constants, Variables, Arrays, and Dictionaries	83
Project Preview: The Story of Captain Adventure	83
Understanding How Scripts View Data	84
Working with Data That Never Changes	85
Assigning Data to Constants	86
VBScript Run-Time Constants	88
Storing Data That Changes During Script Execution	91
VBScript Data Types	91
Defining Variables	93
Variable Naming Rules	95
Variable Scope	96
Modifying Variable Values with Expressions	96
Using the WSH to Work with Environment Variables	100
Working with Collections of Related Data	103
Single-Dimension Arrays	103
Multiple-Dimension Arrays	105
Processing Array Contents	105
Getting a Handle on the Size of Your Arrays	108
Resizing Arrays	109
Building Dynamic Arrays	111
Erasing Arrays	112
Storing Data in Dictionaries	113
Keys and Values	113
Adding Dictionary Items	113
Retrieving Dictionary Items	114
Deleting Dictionary Items	114
Processing Data Passed to a Script at Run-Time	114
Passing Arguments to Scripts	114
Designing Scripts That Accept Argument Input	115
Back to the Story of Captain Adventure	117
Designing the Game	117
The Final Result	120
Summary	121

Chapter 5 Conditional Logic	123
Project Preview: The Planet Trivia Quiz Game	123
Examining Program Data	124
The If Statement	125
The Select Case Statement	133
Performing More Complex Tests with VBScript Operators	135
Back to the Planet Trivia Quiz Game	136
Game Development	137
The Fully Assembled Script	142
Summary	142
Chapter 6 Processing Collections of Data	143
Project Preview: The Guess a Number Game	143
Adding Looping Logic to Scripts	144
The For...Next Statement	145
The For Each...Next Statement	147
The Do...While Statement	149
The Do...Until Statement	152
The While...Wend Statement	153
Back to the Guess a Number Game	154
Designing the Game	154
The Final Result	157
Creating Shortcuts for Your Game	158
A Complete Shortcut Script	163
Summary	164
Chapter 7 Using Procedures to Organize Scripts	165
Project Preview: The BlackJack Lite Game	165
Improving Script Design with Procedures	167
Introducing Subroutines	167
Creating Custom Functions	168
Improving Script Manageability	169
Writing Reusable Code	170
The Guess a Number Game Revisited	170
Working with Built-in VBScript Functions	174
Limiting Variable Scope with Procedures	174
Back to the BlackJack Lite Game	176
Designing the Game	176
The Final Result	184
Summary	184

Part III

Advanced Topics 185

Chapter 8 Storing and Retrieving Data	187
Project Preview: The Lucky Lottery Number Picker	187
Working with the Windows File System	189
Opening and Closing Files	191
Writing to Files	194
Writing Characters	194
Writing Lines	195
Adding Blank Lines	195
Reading from Files	196
Skipping Lines	198
Reading Files Character by Character	198
Reading a File All at Once	199
Managing Files and Folders	199
Copying, Moving, and Deleting Files	200
Copying One or More Files	201
Moving One or More Files	201
Deleting One or More Files	202
Creating a New Folder	202
Copying Folders	202
Moving Folders	203
Deleting Folders	203
Storing Script Configuration Settings in External Files	204
INI File Structure	204
A Working Example	205
Back to the Lucky Lottery Number Picker	207
Designing the Game	208
The Final Result	216
Summary	217
Chapter 9 Handling Script Errors	219
Project Preview: The Hangman Game	219
Understanding VBScript Errors	221
Understanding Error Messages	221
Preventing Logical Errors	222
Dealing with Errors	223
Letting Errors Happen	223

Ignoring Errors	224
Creating Error Handlers	225
Reporting Errors	227
Creating a Custom Log File	228
Recording an Error Message in the Application Event Log	229
Back to the Hangman Game	230
Designing the Game	230
The Final Result	242
Summary	243
Chapter 10 Using the Windows Registry to Configure Script Settings	245
Project Preview: Part 2 of the Hangman Game	245
Introducing the Windows Registry	247
How Is the Registry Organized?	248
Understanding How Data Is Stored in the Registry	249
Accessing Registry Keys and Values	250
Creating a Key and Value to Store Script Settings	251
Creating or Modifying Registry Keys and Values	252
Accessing Information Stored in the Registry	252
Deleting Keys and Values	252
Retrieving System Information Stored in the Registry	253
Back to Part 2 of the Hangman Game	254
Creating the Setup Script	255
Updating the Hangman Game	257
Summary	262
Chapter 11 Working with Built-in VBScript Objects	265
Project Preview: The Tic Tac Toe Game	265
Leveraging VBScript's Built-in Collection of Objects	267
Built-in Object Properties	268
Built-in Object Methods	268
Creating Custom Objects	269
Defining a Custom Object	270
Defining Object Properties and Methods	270
Creating Event Procedures	271
Working with the Err Object	274
Working with Regular Expressions	275
Replacing Matching Patterns	276
Testing for Matching Patterns	279

Creating Matches Collections	279
Back to the Tic Tac Toe Game.	280
Designing the Game	281
The Final Result	293
Summary	294
Chapter 12 Combining Different Scripting Languages	295
Project Preview: The VBScript Game Console	295
Introducing Windows Script Files	297
Examining WSH-Supported XML Tags.	297
Using the <?job ?> Tag	298
Using the <?xml ?> Tag	299
The <comment> and </comment> Tags	299
The <job> and </job> Tags	300
The <package> and </package> Tags	301
The <resource> and </resource> Tags	302
The <script> and </script> Tags	302
Executing Your Windows Script Files.	303
Back to the VBScript Game Console	304
Designing the Game Console	304
Using XML to Outline the Script's Structure	304
Writing the First JScript.	305
Developing the VBScript Game Console.	307
Writing the Second JScript	315
The Final Result	315
Summary	316
Chapter 13 Working with the Windows Management Instrumentation	317
Introducing the Windows Management Instrumentation.	317
WMI Infrastructure Overview	319
Identifying WMI Consumers	320
Examining the WMI Infrastructure	320
Identifying Managed Resources	323
Scripting the WMI.	323
Developing WMI Scripts.	323
Executing WMI Queries	327
Using the WMI to Manipulate Managed Resources	334
Locating CIM Information	336
Summary	336

Chapter 14 Adding a GUI to Your Scripts	339
Project Preview: The HTA Rock, Paper, Scissors Game	339
Introducing HTML Applications (HTAs)	340
How Do HTAs Compare to HTML Pages?	341
Creating and Executing an HTA.	342
Constructing an HTA	343
Introducing the <HTA:APPLICATION> Tag.	343
The <script> </script> Tags.	347
The <body> </body>Tags	348
The <style> </style> Tags.	349
Adding Interface Elements	352
Creating Interface Controls Using the <input> Tag.	352
Adding a Button Control Using the <button> Tag.	360
Adding a Multi-Line Text Control Using the <textarea> Tag	360
Working with List Controls.	361
Integrating WSH into Your HTAs	366
Starting Other Applications	366
Using WMI to Capture Process Information	367
Other HTA Examples.	370
Back to the Rock, Paper, Scissors Game.	370
Game Development	371
The Fully Assembled Script.	375
Summary	375

Part IV

Appendices

377

Appendix A WSH Administrative Scripting	379
Desktop Administration	380
Configuring the Desktop Background.	380
Configuring the Screensaver	381
Network Administration.	383
Mapping Network Drives	383
Disconnecting Mapped Drives	385
Printer Administration	386
Connecting to a Network Printer	386
Disconnecting from a Network Printer	387

Computer Administration	389
Managing Services	389
User Account Administration	391
Disk Management	392
Integrating VBScript with Other Applications	394
Automating the Generation of Microsoft Word Reports	394
Automating the Execution of Third-Party Applications	397
HTML Applications	399
Wrapping a GUI Around a WSH ping Script	399
Automating Windows Shutdown	402
Appendix B Introducing Remote WSH	405
Introducing Remote WSH	405
Understanding Remote WSH's Supporting Architecture	406
Executing Remote WSH Methods	407
Responding to WSH Remote Events	407
Accessing WSH Remote Properties	408
Working with Remote WSH: A Demonstration	409
Appendix C The WSH Core Object Model	411
WSH Objects and Their Properties and Methods	412
Examining Object Properties	414
Working with Object Properties	416
Examining Object Methods	417
Working with Object Methods	419
Appendix D Built-in VBScript Functions	421
Appendix E What's on the Companion Website?	427
Script Examples	427
Index	433

This page intentionally left blank

Introduction

Welcome to the fourth edition of *Microsoft WSH and VBScript Programming for the Absolute Beginner*. Visual Basic Scripting language (VBScript) is a member of the Visual Basic family of programming languages. Other members of this family include Visual Basic and Visual Basic for Applications (VBA). Visual Basic is a very powerful and complex programming language used by programming professionals all over the world. VBA is a programming language based on Visual Basic that is designed to provide a programming environment for Microsoft Office applications such as Excel and Access.

Like VBA, VBScript represents a subset of the Visual Basic programming language. VBScripts can be run on any computer running Windows 95 or later as long as the Windows Script Host (WSH) is installed. The WSH represents one of several environments in which VBScripts can be run. Other environments in which VBScripts can run include HTML pages processed by Internet Explorer-compatible Web browsers and within Microsoft Outlook or Active Server Pages (ASP). Of all the environments in which VBScripts can run, the WSH is the most commonly used. However, by learning to write VBScripts using the WSH, you are also learning much of the prerequisite knowledge required to write VBScripts that will run in each of these other environments.

The WSH provides VBScripts with the capability to execute on Windows computers and to directly access and manipulate Windows resources such as the Windows desktop, file system, Registry, printers, network resources, and so on. You can think of the relationship between VBScript and the WSH as follows: VBScript provides the capability to create scripts and apply logic to perform specific tasks that manipulate Windows resources, which are made available to the script via the WSH.

Why VBScript?

VBScript is an excellent first programming language to learn. Its simplicity makes learning basic programming concepts easy. Yet VBScript is a powerful scripting language from which you can learn even the most complex programming concepts such as how to perform object-based programming. Unlike Visual Basic, VBA, and many other programming languages, there is no complex development environment to learn. In fact, you can create all your VBScripts using a simple text editor such as Windows Notepad.

VBScript provides a foundation that will later make learning Visual Basic and VBA a lot easier. VBScript is a great language for developing small but powerful scripts that perform all sorts of tasks. In fact, you'll find that many VBScripts are not very big at all when compared to programs written using more traditional programming languages. As you read through this book, I think you will be amazed at just what you can do with only a handful of lines of VBScript code. This makes VBScript the perfect language for rapid development, meaning that you can often write a VBScript to perform a task in a fraction of the time that it might take to write a program that performs the same task using a different programming language. Best of all, VBScript is free.

Who Should Read This Book?

This book is designed to teach you how to begin developing VBScripts using the WSH. It does not assume that you have a programming background. However, a basic understanding of computers and Microsoft Windows is assumed.

If you are a first timer looking for a friendly language with which to begin a programming career or a more experienced programmer who is looking for a book that provides you with a quick WSH and VBScript learning curve, then give this book a try.

This book's games-based teaching approach makes it very different from other books. This approach is not only more fun, but is also an extremely helpful technique for learning a new programming language.

What You Need to Begin

To follow along and complete all the exercises that you'll find in this book, you'll need a number of things:

- A computer running Windows.
- The current version of the WSH, which is version 5.8. If your computer is running Windows 7, Windows 8, or Windows 8.1, then you already have the version of WSH that you need. If you are using Windows XP with Service Pack 3 or Windows Vista, you can download and install WSH 5.7 from www.microsoft.com/downloads/.
- A text editor that supports the creation of plain-text files to create and work with your VBScripts. For this book, you can use the Windows Notepad application. Alternatively, you may prefer to download and install a VBScript-compatible script editor. Specialized VBScript editors provide numerous advanced features not provided by Notepad, including statement color-coding, a built-in debugger, line and column numbering, script execution from within the editor, statement indentation, and more. A good example of a VBScript editor is Adersoft VbsEdit. It is distributed as shareware with a limited period of free trial and can be downloaded from www.vbsedit.com.

How This Book Is Organized

The fourth edition of *Microsoft WSH and VBScript Programming for the Absolute Beginner* has been improved in a number of ways. For starters, it has been updated to cover WSH 5.8 and VBScript 5.8, both of which were updated with the release of Windows 7. All scripts have been tested and their execution verified on both Windows 7 and Windows 8.1. In addition, a new chapter has been added that provides an introduction to HTML Applications (HTAs), which provide a mechanism for creating scripts that feature a graphical user interface (GUI). Lastly, I have streamlined coverage of many topics spread throughout the book to provide an even better learning experience.

This book is organized into four parts with the intention that you read it sequentially from beginning to end. If you are a new or inexperienced programmer, you will want to read this book in this manner. However,

if you already know another programming language and feel that you have a strong enough background in basic programming concepts, you might want to skip around and tackle each chapter in the order that best suits your particular requirements.

Part I, “Introducing the WSH and VBScript,” consists of two chapters and provides an introduction to both VBScript and the WSH. Part II, “Learning VBScript and WSH Scripting,” contains five chapters, which cover the programming statements that make up the VBScript scripting language. In addition, you’ll find coverage of the WSH woven throughout these chapters. The seven chapters in Part III, “Advanced Topics,” are dedicated to covering a collection of advanced topics that include file and folder administration, error handling, interaction with the Windows Registry, working with built-in VBScript objects, using XML to create WSH files, working with Windows Management Instrumentation, and adding graphical user interfaces to your scripts. Part IV, “Appendices,” is a collection of five appendices that provide you with additional avenues of exploration, including examples of real-world scripts, an introduction to Remote WSH, documentation of built-in VBScript functions, and a look at this book’s companion website.

The basic outline of the book is as follows:

- **Chapter 1, “Getting Started with the WSH and VBScript.”** This chapter provides a high-level introduction to both the WSH and VBScript. This includes how to install the WSH and how to create and execute your first VBScript.
- **Chapter 2, “An Introduction to the Windows Script Host.”** This chapter provides an introduction to the WSH core object model and the objects that comprise it. Particular attention is paid to the `WScript` root object. You’ll also learn how to configure the WSH and how to specify a default script execution host.
- **Chapter 3, “VBScript Basics.”** This chapter begins your VBScript education. You’ll learn about VBScript’s core and run-time objects and their properties and methods. You’ll learn about other VBScript elements including VBScript’s built-in functions, syntax rules, and output methods. You’ll also learn about various WSH output functions.
- **Chapter 4, “Constants, Variables, Arrays, and Dictionaries.”** This chapter shows you how to create and reference data stored in the computer’s memory using constants, variables, and arrays. You’ll learn about VBScript’s built-in collection constants. This chapter also presents the rules for variable creation and the enforcement of variable use as well as the techniques required to store and retrieve collections of data in arrays.
- **Chapter 5, “Conditional Logic.”** This chapter expands your scripting background to include an understanding of how to add conditional logic to your scripts to provide alternative execution paths for script execution. You’ll examine both the VBScript `If` and `Select Case` statements. In addition, you’ll learn about VBScript operators and operator precedence.
- **Chapter 6, “Processing Collections of Data.”** This chapter teaches you how to process collections of data and resources using various VBScript looping statements (`For...Next`, `Do While`, `Do...Until`, `While...End`, and `For Each...Next`). You’ll learn how to write small scripts that can add shortcuts to your scripts on the Windows desktop and Start menu.

- **Chapter 7, “Using Procedures to Organize Scripts.”** In this chapter, you learn how to improve the organization of your scripts using procedures. You’ll also be introduced to the concept of creating reusable procedures. This will help you create scripts that are more complicated and easier to modify.
- **Chapter 8, “Storing and Retrieving Data.”** This chapter teaches you how to create VBScripts that can write to and read from text files. In addition to learning how to create reports and log files, this chapter shows you how to store and retrieve script configuration settings in INI files, thus allowing you to externalize key script settings.
- **Chapter 9, “Handling Script Errors.”** This chapter focuses on teaching you how to deal with the errors that occur during script development and execution. This chapter introduces errors during script development and shows you how to troubleshoot them. In addition, you’ll learn how to bypass errors and to develop code that handles specific error conditions.
- **Chapter 10, “Using the Windows Registry to Configure Script Settings.”** This chapter provides an overview of the Windows Registry and shows you how to develop scripts that store and retrieve data in Registry keys and values. Because most Windows functionality is controlled from the Registry, this knowledge will provide the basic building blocks required to manipulate any number of Windows settings.
- **Chapter 11, “Working with Built-in VBScript Objects.”** This chapter expands your understanding of object-based programming by reviewing VBScript’s built-in collection of objects. Specifically, you’ll learn new techniques for parsing and extracting data from strings.
- **Chapter 12, “Combining Different Scripting Languages.”** In this chapter, you learn how to take advantage of the WSH’s support for Windows Script Files. Windows Script Files enable you to combine two or more WSH-supported scripting languages, such as VBScript and JScript, into a single script using XML. You’ll also learn a little about XML and the XML tags supported by the WSH.
- **Chapter 13, “Working with the Windows Management Instrumentation.”** This chapter was added to the third edition of this book. It provides an overview of the WMI and the WMI model. You will learn about WMI objects, namespaces, providers, and classes. You will also learn how to develop scripts that use WMI to collect and process systems information.
- **Chapter 14, “Adding a GUI to Your Scripts.”** This chapter, which is entirely new to this edition of the book, introduces HTML Applications (HTAs), which can be used to provide scripts with graphical user interfaces (GUIs). This chapter explains how HTAs work and teaches you how to add GUIs to your scripts, replete with radio buttons, checkboxes, text fields, drop-down lists, and other attributes associated with Windows applications.
- **Appendix A, “WSH Administrative Scripting.”** In this appendix, I show you some practical examples that demonstrate the use of VBScript and the WSH in real-world situations. This appendix will assist you in making a transition from the book’s game-based approach to real-world script development.
- **Appendix B, “Introducing Remote WSH.”** This appendix was added to the third edition of this book. In it, you will learn how to execute, monitor, and terminate the remote execution of scripts using Remote WSH. You will learn about the objects that make up Remote WSH and how to work with their properties and methods.

- **Appendix C, “The WSH Core Object Model.”** This appendix provides detailed information on the WSH core object model using material previously presented in Chapter 2. This includes a detailed examination of WSH objects’ methods and properties.
- **Appendix D, “Built-in VBScript Functions.”** In this appendix, I list and define all the functions that are available as you develop your VBScripts.
- **Appendix E, “What’s on the Companion Website?”** In this appendix, I provide more information about the sample scripts provided on the book’s companion website (www.cengageptr.com/downloads).

Conventions Used in This Book

To help make this book as easy as possible to read and understand, a number of conventions have been applied to help highlight critical information and to emphasize specific points. These conventions are as follows:

Hint

Whenever I can, I provide tips on how to do things differently and point out techniques that you can try to become a better programmer in “hint” boxes.

Trap

From time to time, I use “trap” boxes to point out areas where you are likely to run into problems and then provide you with advice on how to deal with these situations—or, better yet, to prevent them from happening in the first place.

Trick

Whenever I can, I share programming shortcuts that will help to make you a better and more efficient programmer. These appear in “trick” boxes.

In the Real World

Throughout the book, I’ll stop along the way to point out how the knowledge and techniques that you are learning can be applied to real-world scripting projects. These will appear in “real world” boxes.

Definition

Whenever a new term is introduced, I will provide you with an explanation of that term’s meaning in a “definition” box.

In addition, toward the end of each chapter, you will find instructions that guide you through the development of a new computer game. In most chapters, immediately following each game project, you will find a series of suggestions or challenges designed to provide you with ideas that you should be able to apply in order improve the game and further the development of your programming skills. These appear under a “Challenge” heading.

Companion Website Downloads

You may download the files for this book from www.cengageptr.com/downloads. For more information about what files are available, see Appendix E.

I

Introducing the WSH and VBScript

**Chapter 1: Getting Started with the
WSH and VBScript**

**Chapter 2: An Introduction to the
Windows Script Host**

This page intentionally left blank

1

Getting Started with the WSH and VBScript

In this chapter, you'll be introduced to a number of topics. These include a high-level overview of the Windows Script Host (WSH) and VBScript. You will learn how the WSH and VBScript work together to provide a comprehensive scripting environment. You will also be introduced to HTML Applications (HTAs) and learn how an HTA can be used to provide your scripts with a graphical user interface (GUI). In addition, you'll learn a little bit about VBScript's history and its relationship to other languages in the Visual Basic programming family. As a wrap-up, you'll learn how to create and execute your very first VBScript.

Specifically, you will learn the following:

- The basic mechanics of the WSH
- How to write and execute VBScripts using the WSH
- Background information about VBScript and its capabilities
- How you can use HTAs to add GUIs to your scripts
- How to create your first VBScript game

Project Preview: The Knock Knock Game

In this chapter, as in all the chapters to follow, you will learn how to create a computer game using VBScript. This chapter's game is called the Knock Knock game. Actually, it's more of a riddle than a game, but it provides a great starting point for demonstrating how VBScript works and how it can be used to develop games and other useful scripts.

The Knock Knock game begins by displaying a pop-up dialog box that reads “Knock Knock.” It then waits for the user to respond with “Who’s there?” The dialog between the game and the player continues until the computer finally displays the game’s punch line. Figures 1.1 through 1.3 demonstrate operations of the script on Windows 7 and show the flow of the conversation between the game and the player. Figure 1.4 shows the message that appears if the player does not play the game correctly.

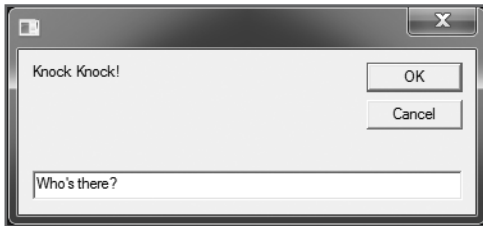


Figure 1.1 The game begins by knocking on the door and waiting for the player to respond. © 2014 Cengage Learning.

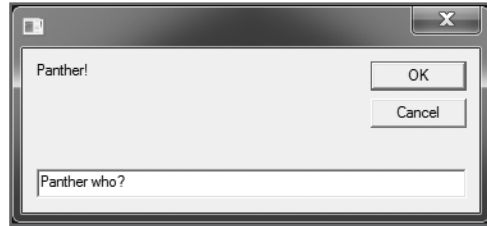


Figure 1.2 The first clue is provided. © 2014 Cengage Learning.

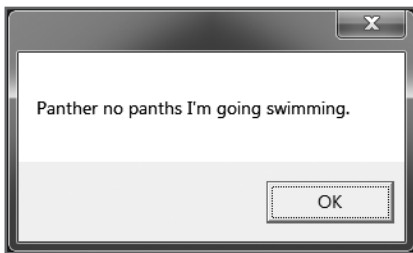


Figure 1.3 The joke’s punch line is delivered. © 2014 Cengage Learning.

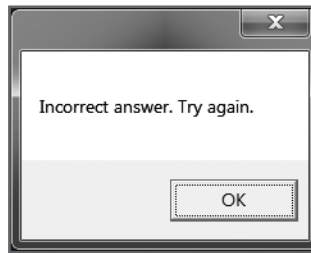


Figure 1.4 If the user makes a mistake when playing the game, an error message providing another invitation to play the game appears. © 2014 Cengage Learning.

By the time you have created and run this game, you’ll have learned the fundamental steps involved in writing and executing VBScripts. At the same time, you will have prepared yourself for the more advanced programming concepts developed in later chapters, including how to use the WSH and VBScript to develop some really cool games.

What Is the WSH?

The Windows Script Host (WSH) is a programming environment that allows you to write and execute scripts that run on Windows operating systems. You can use the WSH to create and execute *scripts*—small text-based files written in an English-like programming language—from the Windows command prompt or directly from the Windows desktop. Scripts provide quick and easy ways to automate lengthy or mundane tasks that take too much time or effort using the Windows graphical user interface (GUI). Scripts are also better suited for automating tasks that are not complex enough to justify the development of an entire application using a language such as C++ or Visual Basic.

The WSH is made up of a number of different components. These components include the following:

- Script engines
- Script execution hosts
- The WSH core object model

The relationship of each of the components to one another is shown in Figure 1.5.

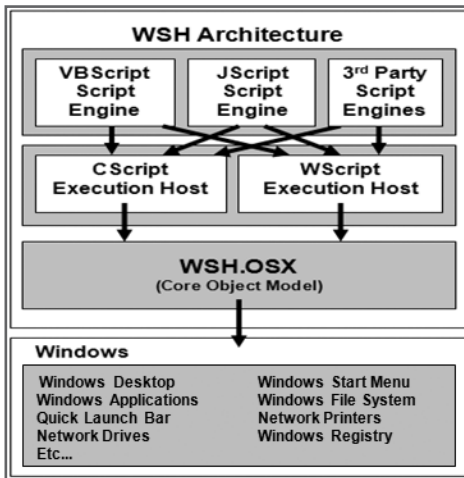


Figure 1.5 The components that comprise the WSH.

© 2014 Cengage Learning.

WSH Scripting Engines

A *script execution engine* is a program that processes (interprets) the statements that make up scripts and translates them into machine-readable code that the computer can understand and execute. By creating an environment in which scripts can execute, the WSH makes script development a straightforward task.

The WSH provides each script with a number of resources. The WSH provides script engines for processing scripts. By default, Microsoft provides two script engines for the WSH:

- **VBScript.** A scripting language based on Microsoft's Visual Basic programming language.
- **JScript.** A scripting language based on Netscape's JavaScript Web-scripting language.

Therefore, by default, the WSH can process scripts written in either VBScript or JScript. The WSH is designed in a modular fashion, allowing Microsoft and third-party software developers to add support for additional scripting engines. For example, script execution engines have been developed for Perl, Python, and Rexx.

Selecting a WSH Script Execution Host

To actually run a script, the WSH uses a script execution host to process a script after a script engine has interpreted that script. The WSH supplies two different script execution hosts:

- **CScript.exe.** An execution host that enables scripts to execute from the Windows command prompt and display text-based messages.
- **WScript.exe.** An execution host that enables scripts to execute from the Windows desktop, display messages, and collect user input using graphical pop-up dialog boxes.

With the exception of the WScript.exe execution host's capability to display graphical pop-up dialog boxes, the functionality provided by the WSH's two execution hosts is identical. In fact, if you run a script using the CScript.exe execution host, the script can, depending on how it is written, still display messages using pop-up dialog boxes.

Definition

Within the context of this discussion, the term *host* describes an environment that provides all the resources required for a script to execute.

As both execution hosts provide the same basic functionality, you're probably wondering which one you should use. There's no right or wrong answer here. Often, the selection of an execution host is simply a matter of personal preference. However, there are some circumstances in which you may want to choose one over the other. For example, if you plan to run your scripts in the background, or if you want to schedule the execution of your scripts using the Windows Task Scheduler service and have no requirement for interacting with the user, you might want to use CScript.exe. However, if your scripts need to interact with the user—which will be the case with the games you'll create with this book—you'll want to use the WScript.exe execution host. Another factor that may affect your selection of a script execution host is your personal comfort level in working with the Windows command prompt.

Introducing the WSH Core Object Model

The WSH provides one final component, called the *core object model*, which is critically important to the development and execution of scripts. The WSH core object model provides VBScript with direct access to Windows resources.

Examples of the types of Windows resources to which the WSH core object model provides access include the following:

- Windows desktop
- Windows Start menu
- Windows applications
- Windows file system
- Network printers
- Network drives
- Windows Registry

The Windows operating system can be viewed as a collection of objects. For example, a file is an object. So is a folder, disk drive, printer, or any other resource that is part of the computer. What the core object model does is expose these objects in a format that allows scripts to view, access, and manipulate them. Each exposed object has associated properties and methods that scripts can then use to interact with an object, as well as affect its behavior or status. For example, a file is an object, and a file has a number of associated properties, such as its name and file extension. By exposing the Windows file system, the WSH enables scripts to access files and their properties and to perform actions, such as renaming a particular file or its file extension. Files also have methods associated with them. Examples of these methods are those that perform the copy and move operations. Using these methods, you can write scripts that can move or copy files from one folder to another or, if you are working on a network, from one computer to another.

Definition

In this book, the term *property* refers to an object-specific attribute, such as a file's name, that can be used to affect the status of the object.

Don't worry if the WSH core object model seems a little confusing right now. You will learn more about it in Chapter 2, "An Introduction to the Windows Script Host." In addition, you can jump to Appendix C, "The WSH Core Object Model," at any time for additional insight. The important thing to understand for now is that the WSH enables scripts to access Windows resources (objects) and to change their attributes (properties) or perform actions that affect them (using object methods).

Definition

In this book, the term *method* is used to refer to a built-in function that your scripts can execute to perform an action on an object, such as to copy or move a file to another location.

How Does the WSH Compare to Windows Shell Scripting?

Windows shell scripts are plain text files that have a .bat or .cmd file extension. Unlike scripts written to work with the WSH, which are written using specific scripting languages like VBScript and JScript, Windows shell scripts are developed using regular Windows commands and a collection of shell-scripting statements. The WSH provides a more complete scripting environment due in large part to its core object model. However, Windows shell scripts still offer a powerful scripting solution. This is partly because you can execute any Windows command or command-line utility from within a shell script. Windows shell scripting also provides a complete collection of programming statements that include support for variables, looping, conditional logic, and procedures. For non-programmers, shell scripts may be easier to read, understand, and modify.

Another difference between scripts written using the WSH and Windows shell scripts is that Windows shell scripts only support text-based communications with the user. In other words, shell scripts cannot display messages or prompt the user for information using graphical pop-up dialog boxes. Windows shell scripting does not provide support for any type of object model like the WSH does. Therefore, Windows shell scripts are not capable of directly interacting with many Windows resources. For example, Windows shell scripts cannot directly edit the Windows Registry or create desktop shortcuts. However, Windows Resource Kits

provide Windows shell scripts with access to a number of command-line utilities that provide indirect access to many Windows resources.

To write shell scripts, you must have a good understanding of Windows commands and their syntax. You must also be comfortable working with the Windows command prompt. Conversely, to effectively use the WSH, you must be well versed in one of its supported scripting languages. There are many cases in which you can accomplish the same task using either Windows shell scripting or the WSH. As a general rule, however, the more complex the task, the more likely you'll want, or need, to use the WSH.

Definition

A *Windows Resource Kit* is a combination of additional utilities and documentation designed for a particular Windows operating system but provided as a separate downloadable package. You can obtain Windows Resource Kits via the Microsoft Download Center (www.microsoft.com/en-us/download).

Hint

If you're really interested in learning more about Windows shell scripting, read *Microsoft Windows Shell Script Programming for the Absolute Beginner* (ISBN 1-59200-085-1).

WSH Versus Windows PowerShell

PowerShell is fully integrated into Microsoft's .NET framework, providing system administrators with access to system resources. Like VBScript and the WSH, Windows PowerShell is object oriented. PowerShell lets you execute PowerShell commands, referred to as cmdlets, and develop and execute small scripts that use those cmdlets.

Like WSH and VBScript, Windows PowerShell provides access to system resources and can be used to programmatically interact with the files system, Windows Registry, .NET, and WMI. WSH, VBScript, and Windows PowerShell support a robust collection of language constructions like variables, conditional logic, loops, and functions.

Unlike WSH and VBScript, PowerShell does not support the use of pop-up dialog boxes and is restricted to the command line. Unlike VBScript, which is based on the widely popular and easy-to-use BASIC programming language, Windows PowerShell represents a completely new scripting language, which is arguably more difficult for new programmers to learn and understand.

Microsoft has put a lot of time and resources into the development of Windows PowerShell and is promoting it as the future of Windows scripting. However, Microsoft is continuing to support the WSH as a Windows scripting environment, as evidenced by the recent release of WSH 5.8. Microsoft will continue to support the WSH—and for good reason. Companies all over the world have invested significant time and resources in it and have developed hundreds of millions of lines of code that are used to run mission-critical applications and administer servers and workstations.

Companies continue to rely on the WSH and VBScript and extend their use. As such, WSH and VBScript programming will remain essential for application developers and systems administrators for the foreseeable future.

Hint

If you're really interested in learning more about Windows PowerShell, read *Microsoft Windows PowerShell 2.0 Programming for the Absolute Beginner, Second Edition* (ISBN 1-59863-899-8).

Understanding How the Windows Shell Works

Even if you have used Windows operating systems for many years, chances are that you have only limited experience working with the Windows shell. To become a really efficient and proficient script programmer, you'll need a solid understanding of what the Windows shell is and how to work with it.

An understanding of how to work with the Windows shell is also important when learning how to work with the Cscript.exe execution host, because scripts run by this execution host are generally started from the Windows command prompt. Finally, it's important to understand the Windows shell when working with the WScript.exe execution host because it provides support for command-line script execution.

Definition

The Windows *command prompt* enables you to submit commands to the Windows shell for processing. By default, the command prompt appears in the form of a drive letter followed by a colon, the backslash character, and then the greater-than character (for example, C:\>).

You cannot touch the Windows operating system itself. This would be far too complex and difficult. Instead, you must go through an interface. Windows operating systems support two such interfaces:

- **The Windows GUI.** The Windows GUI is provided in the form of the Windows desktop, Start menu, and other graphical elements with which you normally interact when using your computer. The purpose of the GUI is to make the operating system easier to work with.
- **The Windows shell.** The Windows shell is a text-based interface between you or your scripts and the operating system. You communicate with the Windows shell by typing commands in the Windows command prompt; the Windows shell translates these commands into a format that the operating system can process. The operating system then returns any results to the Windows shell, which displays them in the Windows Console.

Accessing the Windows Console in Normal Mode

To access the Windows shell and begin working with it using the command prompt, you must first open a Windows Console. To open a Windows Console on a computer running Windows Vista or Windows 7 (see Figure 1.6), open the Start menu, choose All Programs, choose Accessories, and then choose Command Prompt.

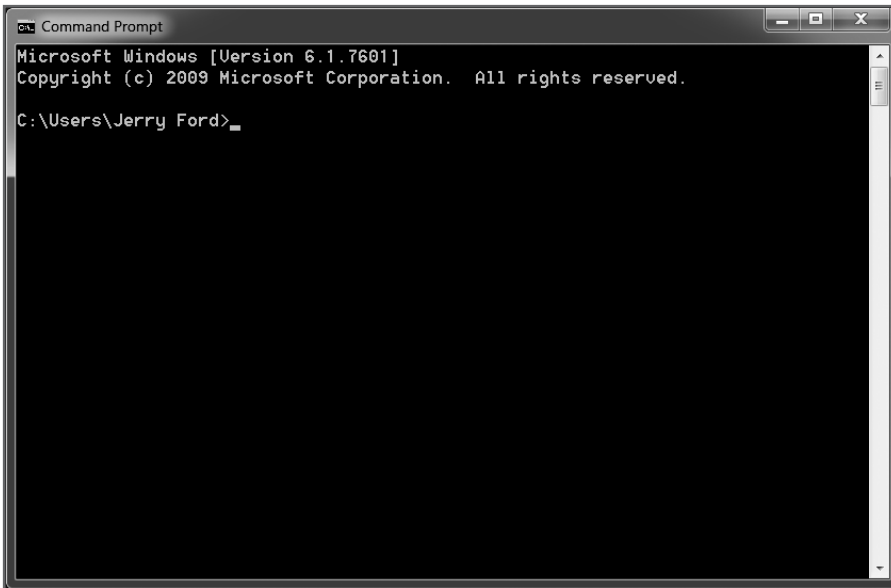


Figure 1.6 The Windows 7 Console provides access to the Windows command prompt.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

To open a Windows Console on Windows 8.1, press the Windows key on the keyboard, type `cmd`, and then press the Enter key.

Accessing the Windows Console in Elevated Mode

In previous editions of this book, all interaction with the Windows shell was done through the Windows Console in what now can be referred to as normal mode. Starting with Windows Vista, however, Microsoft introduced the concept of elevated command line access, such that there are now two different modes in which the Windows Console can operate. In normal mode, you can execute any command or script as long as it does not require administrative level privileges to run.

Hint

For most of the examples and work done in this book, accessing the command prompt through a Windows Console operating in normal mode will be sufficient.

Starting an Elevated Windows Console

To execute commands and scripts requiring elevated access privileges, you need to open an elevated instance of the Windows Console. Doing so is easy. On Windows Vista and Windows 7, all you have to do is open the Start menu, type `cmd` in the Search field, right-click the `cmd` utility, and then choose Run as Administrator from the pop-up menu that appears (see Figure 1.7).

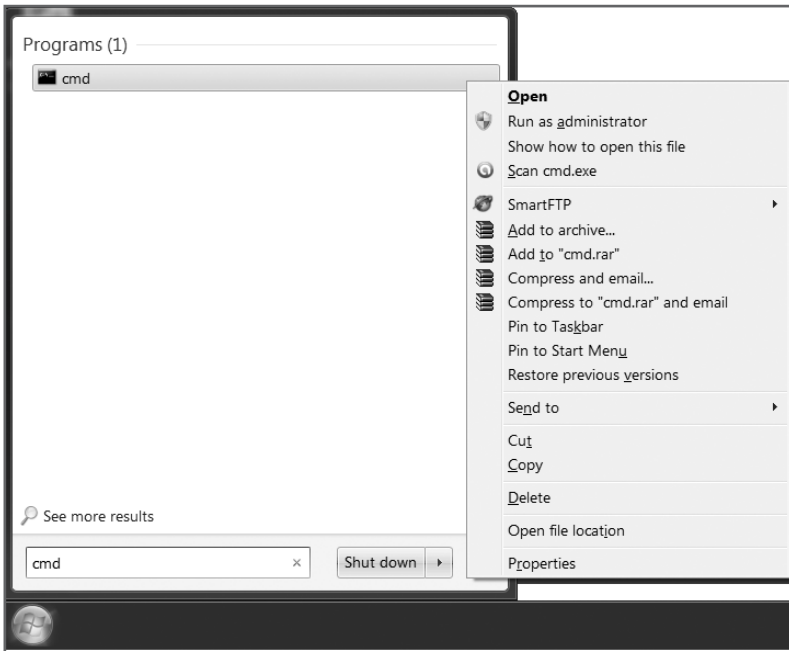


Figure 1.7 Accessing the Windows Console in elevated mode on a computer running Windows 7.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

In response, Windows Vista or Windows 7 will display the User Account Control dialog box shown in Figure 1.8. This dialog box requires you to confirm your command to run the Windows Console in elevated mode. Click Yes; Windows will display the window shown in Figure 1.9. You can tell the Windows Console is running in elevated mode by the appearance of the word “Administrator” in the text string displayed in the Windows Console’s title bar (see Figure 1.9).



Figure 1.8 The User Account Control dialog box on a computer running Windows 7.

© 2014 Microsoft Corporation. Used with permission from Microsoft.



Figure 1.9 The Windows Console in elevated mode on a computer running Windows 7.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

To open a Windows Console in elevated mode on Windows 8.1, move the cursor to the bottom-right corner of the screen. This will display a list of icons, the top-most of which is the Search icon. Click this icon and type `cmd`. When the Windows Console option appears, right-click it and select Run as Administrator from the menu that appears.

Adding a Menu Command for Starting an Elevated Windows Console

If you find yourself frequently needing to work with Windows Console in elevated mode, you can configure a Run as Administrator menu option for the Windows Console to make things easier on yourself. The following procedure outlines how to set this up on a computer running Windows 7.

1. Click the Start button, type Regedit in the Search field, and press the Enter key to open the Regedit utility.
2. Click Yes when prompted by the User Account Control dialog box.
3. Navigate to the following key: `HKEY_CLASSES_ROOT\VBScript\Shell`.
4. Right-click the key, select New > Key, and type RunAs as its name.
5. Right-click the RunAs key and select New > Key. Then type Command as its name.
6. Select the Command key and then double-click its (Default) value to open the Edit String dialog box.
7. Type `"C:\Windows\System32\WScript.exe" "%1" %"` as its Value data and click OK.

8. Right-click on the Command key and select New > String Value. Then type IsolatedCommand and press the Enter key.
9. Double-click the IsolatedCommand key to display the Edit String dialog box.
10. Type "C:\Windows\System32\WScript.exe" "%1" %" as its Value data and click OK.
11. Close the Regedit utility.

Now the next time you need to run a VBScript using the WScript host with elevated privileges, all you have to do is right-click on the script and select Run as Administrator from the menu that appears, as shown in Figure 1.10.

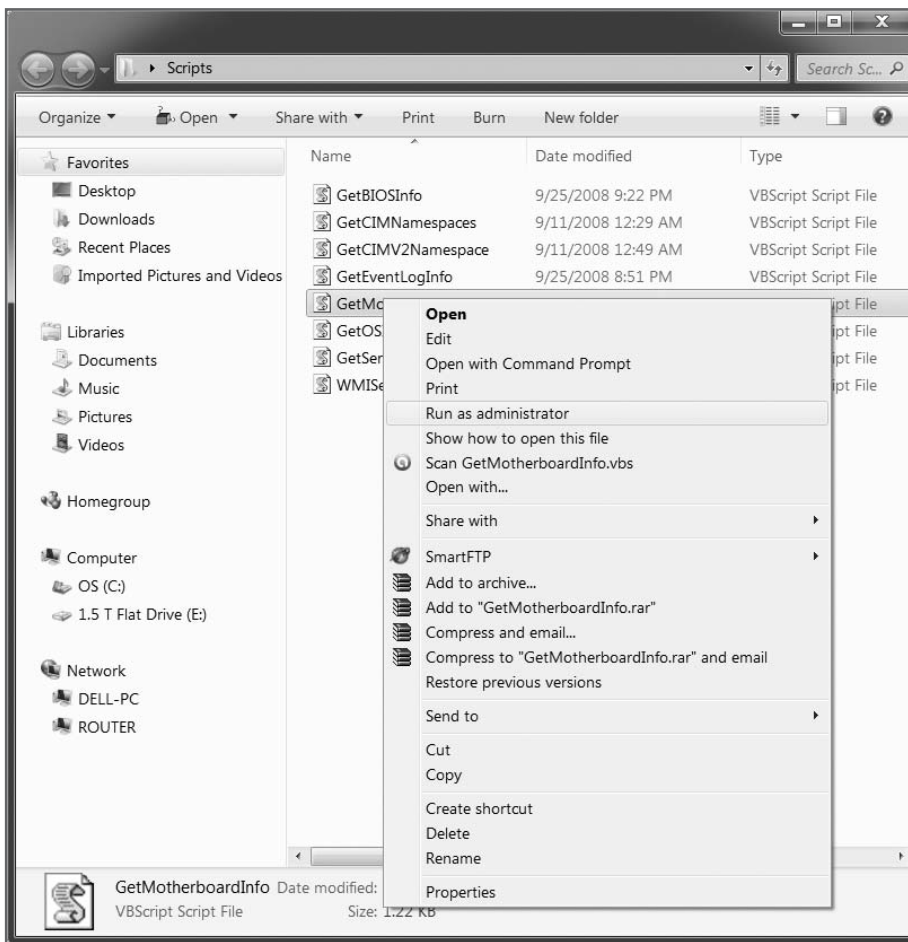


Figure 1.10 Adding Run as Administrator as a menu option for your VBScripts.

Interacting with the Windows Shell through the Command Prompt

As you can see, when the Windows Console first opens, it displays information about the version of Windows in use and Microsoft's copyright information. Then the command prompt appears. Just to the right of the command prompt, you'll see a blinking cursor or underscore character. This character indicates that the command prompt is ready to accept input. For example, type the command `DIR` and then press the Enter key. The `DIR`, or *directory*, command instructs Windows to display a list of all the files and folders in the current working directory. The following output shows the results that were returned when I executed this command on my computer:

```
C:\>dir
Volume in drive C is OS
Volume Serial Number is 2A2C-3CC5

Directory of C:\

03/29/2012  05:37 PM    <DIR>          Apps
03/29/2012  06:27 PM    <DIR>          Drivers
10/08/2013  09:44 AM    <DIR>          history
11/10/2013  11:14 PM    <DIR>          Program Files
11/10/2013  11:19 PM    <DIR>          Program Files (x86)
10/04/2013  04:23 PM    <DIR>          Scripts
01/23/2013  09:46 PM    <DIR>          Temp
03/03/2013  11:25 AM    <DIR>          Users
11/02/2013  10:39 AM    <DIR>          Windows
             5 File(s)          40,180 bytes
            17 Dir(s) 283,386,662,912 bytes free

C:\>
```

As you can see, the last line in the output is the Windows command prompt. The Windows shell redisplay the command prompt as soon as the output of the `DIR` command is complete, allowing for the entry of another command. For example, if I have a VBScript named `Hello.vbs` in a folder named `Scripts` on the computer `C:` drive, I could execute it by typing `CScript C:\Scripts\Hello.vbs` and pressing the Enter key. After the script finishes its execution, I could type additional commands, run more scripts, or end my Windows shell session by closing the Windows Console. The Windows Console is closed just like any other Windows application: by clicking the Close (\times) button in the upper-right corner of the Windows screen or by right-clicking on the icon in the upper-left corner of the screen and selecting `Close`.

Hint

You also can close the Windows Console by typing `Exit` and pressing the Enter key.

How Does It All Work?

To execute a script using the WSH, you must first create the script using one of the WSH's supported scripting languages. In this book, that language is VBScript. Windows operating systems recognize the type of data stored in files based on the file extension assigned to the file. For example, a file with a .txt file extension is a text file. Windows automatically associates files with this file extension with its Notepad application. Therefore, when you double-click on a file with a .txt extension to open it, Windows automatically loads the file into Notepad.

When you create your VBScripts, you need to save them as plain-text files and assign them a .vbs file extension. That way, Windows will know that the file contains VBScripts. In a similar fashion, to write a script using JavaScript, you must save the file with a .js file extension so that Windows can properly identify it as well.

As long as the WSH has been installed on your computer, all you have to do to execute a script that has been saved with the appropriate file extension is to run it. There are several ways to run a script. One way is to simply double-click on the file. Windows will recognize the file as a script and then automatically process it using the appropriate WSH script engine (based on the script's file extension). What happens next depends on how you have configured the WSH. By default, the WSH is configured to run all scripts using the WScript.exe execution host, although you can modify this default behavior to make the CScript.exe execution host the default if you want. However, the WScript.exe execution host allows scripts to display messages and to collect text input using graphical pop-up dialog boxes, but the CScript.exe execution host does not. As the script runs in the execution host, it can access and manipulate Windows resources, thanks to the core object model.

Trap

Windows runs a script based on the authority of the person who starts it. Therefore, your scripts have no more access to Windows and its resources than you do. If you try to create a script to perform a task that you cannot perform manually via the GUI, your script will not work. If this is the case, you might want to talk with your system administrator to see if you can be assigned additional access permissions and user rights.

Operating System Compatibility

The current version of the WSH is 5.8. This is the fifth version of the WSH released by Microsoft. The four previous versions were versions 5.7, 5.6, 2.0, and 1.0. Depending on which operating system your computer runs, you may already have access to one of these versions. For example, if you are using Windows Server 2008 R2, Windows 7, Windows 8, or Windows 8.1, then you already have WSH 5.8. However, if you work with other Windows operating systems, you likely have an older version of the WSH installed. Table 1.1 provides a list of Windows operating systems and the version of the WSH that is supplied with them.

This book covers WSH 5.8 and VBScript 5.8. However, because WSH 5.6 and WSH 5.7 and VBScript 5.6 and VBScript 5.7 are nearly identical, everything that you learn in this book should apply to these earlier versions.

**TABLE 1.1 VERSIONS OF THE WSH FOUND
ON MICROSOFT OPERATING SYSTEMS**

Operating System	WSH Version	Operating System	WSH Version
Windows 95	None	Windows 2008	5.7
Windows NT 4.0	None	Windows Vista	5.7
Windows 98	1.0	Windows 7	5.8
Windows 2000	2.0	Windows 8	5.8
Windows Me	2.0	Windows 8.1	5.8
Windows XP	5.6	Windows 2012	5.8
Windows 2003	5.6		

© Jerry Lee Ford, Jr. All Rights Reserved.

Hint

Microsoft provides free downloads for WSH 5.7 for Windows 2000, Windows XP, and Windows 2003 from the Microsoft Download Center (www.microsoft.com/downloads/). As of the writing of this book, no downloads were available for WSH 5.8. Windows XP users can also upgrade to WSH 5.7 by installing Service Pack 3. WSH upgrades are not longer available for Windows 95, NT, 98, or Me.

How Do You Install It?

You can download and upgrade to WSH 5.7 on Windows 2000, XP, or 2003. Microsoft provides separate downloads for each of these three operating systems at www.microsoft.com/downloads/. The steps involved in upgrading to version 5.7 once you've downloaded it are outlined here:

1. Double-click on the Windows script host file that you downloaded to begin the installation process.
2. Click Next when the Software Update Installation wizard appears.
3. Click I Agree when prompted by the wizard to accept the license agreement.
4. The wizard will complete the installation process. Click Finish when prompted to close the wizard.

When the installation process is complete, the following components will have been installed. At this point, your computer is ready to support the development and execution of VBScripts using the latest and most reliable version of WSH and VBScript.

- Visual Basic Script Edition (VBScript) 5.7
- JScript 5.7
- Windows Script Host 5.7

Hint

WSH 5.7 downloads as a Windows Package Installer file, allowing you to manage it using the Add/Remove Programs option in the Windows Control Panel.

How Does It Work with VBScript?

Microsoft originally designed VBScript to operate as a Web-scripting language. That means it could run only when embedded within HTML pages that were executed by Internet Explorer. VBScript's success as a Web-scripting language has always been limited. One reason for this is that Netscape never provided support for it in its Internet browser. In addition, from the beginning, Netscape provided JavaScript free of charge. There was hesitation on the part of many programmers to abandon JavaScript for VBScript, which Microsoft maintained as a proprietary technology, meaning that Microsoft and Microsoft alone owned and controlled VBScript.

Microsoft has since created a modified version of VBScript that is designed to work with the WSH. This version of VBScript lacks many of the features found in browser-based versions of VBScript. For example, it does not work with forms and frames. Then again, as a WSH scripting language, VBScript doesn't need this functionality because these types of resources are beyond the scope of its environment.

Hello World: Creating and Executing Your First VBScript

Instead of being embedded within HTML pages, VBScripts run by the WSH are saved as standalone files with a .vbs file extension. For example, take a look at the following VBScript:

```
MsgBox "Hello World!"
```

As you can see, the script consists of just one line of code. To create this script, open your editor and type the line of code exactly as I've shown it here and then save the script as Hello.vbs. That's it. Now run it: First locate the folder in which you saved the script and then double-click on it. You should see a graphical pop-up dialog box similar to the one shown in Figure 1.11.

Let's talk about the script that you just wrote and executed. First of all, because you executed it by double-clicking it, you ran it using the default execution host. The default execution host is WScript.exe unless you've changed it. (I'll go over how to change the execution host in the next chapter.) The script itself executes a VBScript function called MsgBox().



Figure 1.11 Viewing the pop-up dialog box created by your first VBScript.

© 2014 Cengage Learning.

The `MsgBox()` function is a built-in VBScript function that you can call within your scripts to display messages in pop-up dialog boxes. As you can see, the text “Hello World!” was displayed when you ran the script. This VBScript was run using a WSH execution engine (for example VBScript) and one of the WSH’s two execution hosts (either `WScript.exe` or `CScript.exe`). However, the code itself was all VBScript.

Let’s modify the script just a little bit to demonstrate how to incorporate the `WScript` object. The `WScript` object is one of a small number of objects that make up the WSH core object model. (I’ll go over this object and the rest of the WSH core object model in greater detail in Chapter 2 and Appendix C.) Using your editor, open the `Hello.vbs` script and modify it so that it looks exactly like the following example:

```
Set WshShl = WScript.CreateObject("WScript.Shell")
```

```
WshShl.Popup "Hello World!"
```

Now save the script and run it again. This time, unless you entered a typo, you should see a pop-up dialog box similar to the one shown in Figure 1.12.

Definition

A *function* is a collection of statements that is called and executed as a unit.



Figure 1.12 The pop-up dialog box created by your modified VBScript.

© 2014 Cengage Learning.

As you can see, things look pretty much the same. The same message is displayed and the words “Windows Script H...” are now displayed in the pop-up dialog box’s title bar. Let’s break it down and examine exactly how the script is now written. Don’t worry if you don’t fully understand everything that is covered here—it’s fairly complex and you’ll be better prepared to understand it soon. For now, I’d like you to just read along with the steps I’ll present so that you’ll understand the process involved in creating and executing scripts using VBScript and the WSH.

First, the script uses the `Set` command to define a variable named `WshShl`. This variable is then assigned a value using the following expression:

```
WScript.CreateObject("WScript.Shell")
```

This statement executes the `WScript` object’s `CreateObject()` method. This method is used to instantiate (that is, create a new instance of) the `WshShell` object, which is another WSH core object. The second line of code in the example uses the `WshShell` object’s `Popup()` method to display a pop-up dialog box.

Hint

The `WScript` object is one of the WSH's core objects. Do not confuse it with the WSH `WScript.exe` execution host. It is unfortunate that they share the same name because they are very different.

As the two versions of the previous script show, you can often perform the same task using either a VBScript function or a WSH method. This script also demonstrates how easy script creation and execution can be, and how even a one- or two-line script can perform some pretty neat tricks, such as displaying a pop-up dialog box.

In the Real World

In the previous example, you created your first VBScript by following the steps that I set down. Often, depending on the size and complexity of the script that you're going to develop, you can get away with simply writing the script as you go. More often than not, however, you'll want to take a more methodical approach to script development. First, make sure you know exactly what you want to achieve. Then break the task down into specific steps that, when combined, complete the task. Spend a little time sketching out the design of your script and try to break the script into different sections. Then develop a section at a time, making sure that one section works before moving on to the next. I'll try to point out ways to do this throughout the book.

Executing Your Script from the Command Prompt

In the previous example, you executed your script by double-clicking on it, and everything worked fine. That's because the scripts were written so that they could run from the Windows desktop. Sometimes, however, the execution host you use to run your script can have a big impact on how the script operates. Let's take a look at an example.

1. Open the `Hello.vbs` script again and replace the contents of the script with the following statement:

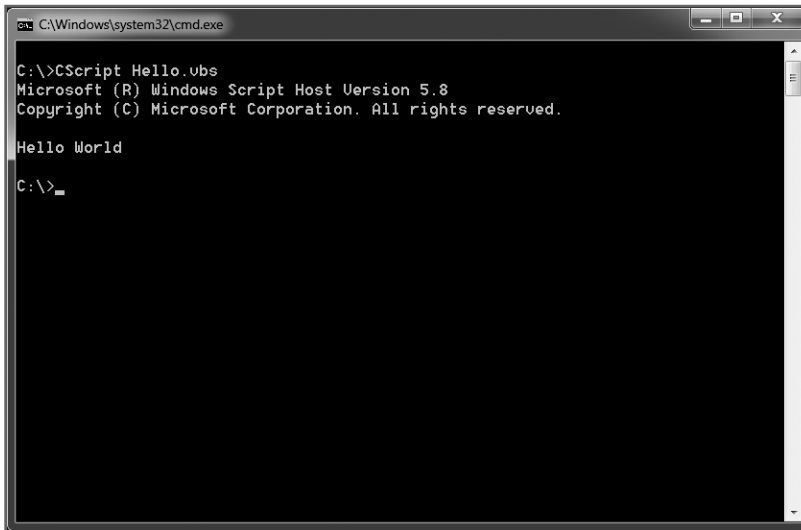
```
WScript.Echo "Hello World"
```

This statement uses the `WScript` object's `Echo()` method to display a text message.

2. Save the script and execute it by double-clicking it. Unless you have modified the default WSH configuration, the script will run using the `WScript.exe` execution host. The result is that the message is displayed in a pop-up dialog box.
3. Copy the file to the C: drive on your computer and open a Windows Console.
4. At the command prompt, type `CD \` and press the Enter key. This command changes the current working directory to the root of the C: drive, where `Hello.vbs` script now resides.
5. Type the following command and press the Enter key:

```
CScript Hello.vbs
```

What you see this time is quite different. Instead of a pop-up dialog box, the script's output is written to the Windows console, as shown in Figure 1.13.

A screenshot of a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. The text displayed is: "C:\>CScript Hello.vbs", "Microsoft (R) Windows Script Host Version 5.8", "Copyright (C) Microsoft Corporation. All rights reserved.", "Hello World", and "C:\>".

```
C:\Windows\system32\cmd.exe
C:\>CScript Hello.vbs
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.
Hello World
C:\>
```

Figure 1.13 Scripts executed by the CScript.exe execution host display their output in the Windows Console.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

- As a final experiment, type the following command at the Windows command prompt:
WScript Hello.vbs

As you see, the message produced by the script is once again displayed in a pop-up dialog box because even though the script was run from the Windows command prompt, the WScript.exe execution host displays its output graphically.

What Other Scripting Languages Does the WSH Support?

As I have already alluded to, the WSH supports other languages besides VBScript. Microsoft ships the WSH with both JScript and VBScript. JScript is Microsoft's implementation of the ECMAScript language, originally developed by Netscape as LiveScript and later renamed JavaScript. Like VBScript, the version of JScript that is shipped with the WSH is a modified version of the browser-based scripting language. Also like VBScript, JScript is a complete programming language replete with support for variables, conditional logic, looping, arrays, and procedures.

JScript's overall syntax structure is a little more difficult to master than VBScript's unless you are already familiar with JavaScript. VBScript provides better support for arrays, whereas JScript provides a stronger collection of mathematical functions. JScripts are created as plain-text files and saved with a .js file extension.

Hint

To learn more about JScript, check out the JScript Documentation link on [http://msdn.microsoft.com/en-us/library/vstudio/72bd815a\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/vstudio/72bd815a(v=vs.100).aspx).

In addition to VBScript and JScript, a number of third-party scripting languages are also designed to work with the WSH, as outlined in Table 1.2.

TABLE 1.2 THIRD-PARTY WSH-COMPATIBLE SCRIPT ENGINES

Language	Website
PerlScript	www.activestate.com
PythonScript	www.sourceforge.com
ooRexx	www.oorexx.org
RubyScript	www.ruby-lang.org
ActiveScript	www.php.net
TclScript	www.sourceforge.com

© Jerry Lee Ford, Jr. All Rights Reserved.

Introducing VBScript

As you now know, VBScript is a scripting language that enables you to develop scripts that automate tasks that would otherwise have to be manually performed in the environment in which they execute. VBScripts are stored as plain-text files with a .vbs file extension and can be created using any text editor. This makes them easy and quick to develop.

Unlike the standalone implementations of many scripting languages, such as Perl or Python, VBScripts cannot execute without an execution host. VBScript was originally designed to execute as text embedded within HTML pages inside the Internet Explorer browser. Over the years, however, Microsoft has extended VBScript's capabilities to enable it to function in numerous different settings. VBScript is now supported in a number of different environments, including the following:

- **Windows Script Host.** VBScript provides a host automation language for performing system and network tasks.
- **Internet Explorer.** VBScript supplies a client-side Web-scripting language.
- **Microsoft Windows Script Console.** This allows VBScript to be added to third-party applications to incorporate its scripting capabilities.
- **Internet Information Server (IIS) and Active Server Pages (ASP).** VBScript can be embedded into ASP to access local databases and help deliver dynamic Web content.
- **Outlook Express.** VBScript provides the ability to automate a number of Outlook's functions.

As you can see, after you master VBScript within the context of WSH script development, you'll have a number of other avenues in which you can begin using your new VBScript programming skills.

VBScript Capabilities

VBScripts cannot execute without an execution host. Therefore, the language's capabilities vary greatly based on where they are run. For example, when embedded in HTML pages, VBScript can access and manipulate forms, frames, links, images, and other objects that are based on Web pages. When placed inside ASP pages, VBScripts have access to server-based resources such as databases. However, because the purpose of this book is to teach you how to program using VBScript within the context of the WSH, I think it's best that we focus on the capabilities that VBScript has when executed in this environment.

As I'll show you throughout this book, you can create games using VBScript and the WSH. While game development is a great way to have fun while learning a new language, it's important to understand the reason Microsoft enabled VBScript to operate in the WSH and to be familiar with the capabilities that Microsoft has given to VBScript within the context of WSH script development.

VBScript provides programmers with a quick development tool for creating small applications and utilities and for prototyping new applications. System and network administrators use these tools to automate system administrative tasks, such as the following:

- Creating user and group accounts
- Configuring the desktop
- Creating ad hoc reports
- Automating network file, folder, and drive administration
- Managing Windows services
- Administering local and network printers

Some tasks simply take a long time to perform manually or must be done so frequently that they become bothersome. By providing the ability to automate these tasks, VBScript provides a powerful yet easy way to use programming tools. Once developed, script execution can be automated using the Windows scheduling service. This allows you to run your scripts at the times that are most convenient for you.

For example, suppose you wrote a script that reorganizes files on your computer by moving them from various folders into a centralized location. That way, at the end of each month, you can run the script and reorganize a month's worth of messy file placement. The number of files to be moved may be such that it takes the script a while to complete its work, during which time the computer runs slowly and is no fun to use. Fortunately for you, however, VBScripts can be scheduled. You can set up the execution of this script to run at night, over the weekend, or any time you don't plan on using your computer.

VBScript's Roots

Microsoft first released VBScript in 1996 as a Web-based client-side scripting language for Internet Explorer 3.0. At the time, another Web-based client-side scripting language, JavaScript, was already making big waves in the Internet community. Despite the similarity in name, JavaScript had very little in common with Java, which was also fast becoming popular in the mid to late 1990s.

As mentioned, JavaScript's popularity as a client-side Web-scripting language has continued over the years, while VBScript's stalled. Even today, the only way to perform client-side Web scripting and to be sure that everyone with an Internet browser has access is to use JavaScript.

Still, Microsoft has remained committed to the development of VBScript over the years. It released VBScript 2.0, along with IIS 3.0, turning VBScript into a server-side Web-development language. Now Web developers could embed VBScripts into their ASP pages, giving them the ability to access local databases and create dynamic HTML pages.

VBScript's big break came with VBScript 3.0. This version was packaged with multiple Microsoft products, including the following:

- Internet Explorer 4.0
- IIS 4.0
- Outlook 98
- Windows Scripting Host

VBScript 3.0 now could be used as a scripting language for Microsoft's email client. However, VBScript really took off when it was included as a scripting language for the WSH. Visual Basic programmers, computer administrators, and technology enthusiasts with a background in Visual Basic found VBScript easy to learn. It quickly proved to be a great language for developing small scripts to perform tasks that did not merit the development of a complete standalone application.

Microsoft later released VBScript 4.0 as part of its Microsoft Visual Studio application development suite. Microsoft gave VBScript 4.0 the capability to access the Windows file system; otherwise, VBScript 4.0 remained pretty much unchanged from the previous version.

In 2000, Microsoft released VBScript 5.0 as a component of Windows 2000, which included Internet Explorer 5 and WSH 2.0. In 2001, Microsoft released Windows XP Professional, Windows XP Home Edition, and Internet Explorer 6.0. Along with these goodies came WSH 5.6 and VBScript 5.6. VBScript 5.7 was made available in 2007 as part of WSH 5.7. It was also made available as part of the install of Internet Explorer 7. Likewise, VBScript 5.8 was made available in 2009 as part of WSH 5.8 and as part of the install of Internet Explorer 8.

VBScript's Cousins: Visual Basic and VBA

VBScript is the third member in a family of three closely related programming languages:

- Visual Basic
- Visual Basic for Applications (VBA)
- VBScript

Visual Basic is the original member of this family. Microsoft first introduced it in 1991. Microsoft has steadily improved Visual Basic, releasing a number of versions along the way. The most current version of Visual Basic is Visual Basic 2012. As a .NET-compliant language, Visual Basic supports Microsoft's .NET framework.

Hint

If you want to learn more about .NET, visit www.microsoft.com/net.

Visual Basic is generally used to create standalone programs. This means that once written and compiled into executable code, a Visual Basic application does not need anything other than a Windows operating system to execute. Visual Basic earned a reputation very early on for being easy to learn. As a result, it did not take Visual Basic long to become one of the most popular programming languages ever developed. Today Visual Basic is taught in colleges around the world and is used to build applications in companies of all sizes and types.

Visual Basic applications are created using Visual Basic's built-in integrated development environment (IDE). Visual Basic's IDE includes a built-in compiler, debugger, help system, and tools for managing Visual Basic projects. Although Visual Basic's IDE provides a rich and powerful programming development environment, it takes a substantial amount of time and effort to learn. Because of the complexities of its IDE, Visual Basic is not well suited to the development of small scripts. Visual Basic's strength lies in aiding the development of larger and more complex programs that justify the time and effort required to develop them.

Hint

To learn more about Microsoft Visual Basic .NET, check out *Microsoft Visual Basic 2008 Express Programming for the Absolute Beginner* (ISBN 1-59863-900-5).

The next language in the Visual Basic family is Visual Basic for Applications (VBA), which Microsoft first released in 1993. VBA represents a subset of Visual Basic and is designed to provide applications with a Visual Basic-like programming language. For example, using VBA for Microsoft Excel, programmers can develop entire applications using features provided by Excel. Similarly, VBA for Microsoft Access provides a powerful programming language for creating applications that require a Microsoft Access database.

Definition

.NET is a Microsoft framework that has been designed by Microsoft from the ground up to support integrated desktop, local area network, and Internet-based applications. Microsoft's .NET framework assists in developing applications by facilitating data exchange over a network—including the Internet.

Definition

An *IDE* is an application development program that gives programmers the tools required to create applications using a particular programming language. An IDE provides tools such as a compiler, which translates application code into a finished executable program; a debugger, which assists in tracking down and fixing programs; and tools for managing projects, which may consist of multiple applications.

Like Visual Basic applications, VBA applications are created using a sophisticated IDE program. Unlike Visual Basic applications, which can be compiled into fully executable programs, VBA can only be compiled into a format known as p-code, which you can think of as *partial compilation*. Using p-code, VBA code can load and run more quickly than VBScript, which is an interpreted language, but will still run more slowly than a Visual Basic application. VBA also requires a host application such as Microsoft Excel or Microsoft Access.

VBA 7.1 was released as part of Microsoft Office 2010 and is still the current version. Using VBA, you can develop programs for any of the following Microsoft applications:

- Word
- PowerPoint
- Excel
- Outlook
- Access
- FrontPage

Hint

To learn more about VBA and Microsoft Excel, check out *Microsoft Excel VBA Programming for the Absolute Beginner*, by Duane Birnbaum. To learn more about VBA and Microsoft Access, check out *Microsoft Access VBA Programming for the Absolute Beginner*, by Michael Vine.

Back to the Knock Knock Game

Let's turn the focus of this chapter back to the development of the Knock Knock game. This project will demonstrate the steps involved in creating and running your first VBScript game. Along the way, you'll learn how to use VBScript to create a script that can communicate with the user via pop-up dialog boxes. You will also learn a little about conditional programming logic.

Designing the Game

The Knock Knock game's design is very straightforward, involving basic programming techniques. The game begins by displaying the message "Knock Knock" in a pop-up dialog box. It then waits for the player to reply by typing "Who's there?" The game then replies "Panther" and waits for the player to respond by typing "Panther who?" at which time the punch line, "Panther no panths I'm going swimming" is displayed. If the player fails to exactly type the proper responses at any point of the game, an error message will be displayed inviting the player to try again.

This project will be completed in five steps, as follows:

1. Present the player with the Knock Knock pop-up dialog box and collect the player's response.
2. Validate the player's reply and continue the game if appropriate. Otherwise, display an error message.
3. Present the player with the name of the person at the door and collect his or her reply.
4. Validate the player's reply and continue the game if appropriate. Otherwise, display an error message.
5. Display the game's punch line.

Starting the Script Development Process

The first step in creating the Knock Knock game is to start your script editor and use it to create an empty VBScript file. For example, to create the script using the Notepad text editor on a computer running Windows XP, you would execute the following steps:

1. Click the Start button, choose All Programs, select Accessories, and choose Notepad. The Notepad application opens.
2. Open the File menu and choose Save. The Save As dialog box appears.
3. Specify the location where you want the script to be stored. Then type `KnockKnock.vbs` in the File Name field at the bottom of the dialog box and click Save.

The Notepad editor should now display the name of the Knock Knock script in its title bar.

Starting the Game and Collecting Initial User Input

Now let's begin the script by writing its first VBScript statement. The first thing the game is supposed to do is display a pop-up dialog box with the "Knock Knock" message and then wait for the user response. This task is performed surprisingly easily using VBScript, and can be done with a single statement:

```
Reply1 = InputBox("Knock Knock!")
```

In plain English, this VBScript statement displays a pop-up dialog box with a "Knock Knock" message and then waits for the player to type something into the dialog box's text field and click the OK button.

Let's break this statement down into pieces and see how it works. First, the statement executes a built-in VBScript function called `InputBox()`. This function displays a pop-up dialog box with a text entry field that allows the script to collect text input from the player.

Definition

A *statement* generally consists of a single line of code but can be spread over two or more lines depending on the size of the statement.

Hint

The VBScript `InputBox()` function is just one of a number of options for collecting input. The `InputBox()` function facilitates direct interaction with users. When direct user interaction is not required, you can also develop VBScripts that can read input from text files or the Windows Registry. I'll show you how to read data from text files in Chapter 8, "Storing and Retrieving Data," and how to interact with the Windows Registry in Chapter 10, "Using the Windows Registry to Configure Script Settings." You can also create VBScripts that process data passed to them at run-time. I'll show you how this works in Chapter 4, "Constants, Variables, Arrays, and Dictionaries."

To communicate with the player, the `InputBox()` function allows you to display a message. In this example, the message is simply "Knock Knock," but could just as easily be "Hello, what is your name?" or any other question that helps the player understand the type of information the script is trying to collect.

Finally, the text typed by the player in the pop-up dialog box's text field is temporarily assigned to a variable called `Reply1`. Variables provide scripts with the capability to store and later reference data used by the script.

Functions and variables are fundamental components of VBScript. Unfortunately, it is difficult to write even the simplest scripts without using them. For now, don't worry too much about them and keep your focus on the overall steps used to create and run the Knock Knock game. I'll go over the use of variables in great detail in Chapter 4 and the use of functions in Chapter 7, "Using Procedures to Organize Scripts."

Validating User Input

The player's role in this game is to first type the phrase "Who's there?" Any variation in spelling or case will result in an error. After the player has typed this message and clicked the OK button, the script needs to perform a test that validates whether the player is playing the game properly.

The following three lines of code accomplish this task:

```
If Reply1 = "Who's there?" Then
.
.
.
End If
If Reply1 <> "Who's there?" Then MsgBox "Incorrect answer. Try again."
```

The first two lines of actual code—`If Reply1 = "Who's there?" Then` and `End If`—go together. The three dots between these lines of code are placeholders for more statements that will be inserted in the next section. The first of these two lines tests the value of `Reply1`. Remember that `Reply1` is a variable that contains the response typed by the player. This statement checks to see if the values stored in `Reply1` match the phrase "Who's there?" If there is an exact match, then the lines of code that you will soon place within the first two statements are executed. Otherwise, these statements are not processed. The third line of code

inverts the test performed by the first two lines of code by checking to see if the player's reply is not equal to (that is, <>) the expected phrase. If this is the case, then the rest of the third statement executes the display of an error message. The text performed by the third statement may prove true for a number of reasons, including the following:

- The player clicked the Cancel button.
- The player clicked the OK button without typing a response.
- The player typed an incorrect response.

Finishing Input Collection

If you are creating the script as you read along, then your script should now contain the following statements:

```
Reply1 = InputBox("Knock Knock!")
If Reply1 = "Who's there?" Then
.
.
.
End If
If Reply1 <> "Who's there?" Then MsgBox "Incorrect answer. Try again."
```

It's now time to add three lines of code that will reside in the lines currently marked with periods. The first of these three lines of code is as follows:

```
Reply2 = InputBox("Panther!")
```

This statement is very similar to the first statement in the script, except that instead of displaying the message "Knock Knock," it displays the message "Panther" and then waits for the player to type a response (that is, "Panther who?"). The text typed by the player is then stored in a variable named Reply2.

Validating the User's Last Response

The following two lines of code need to be inserted just after the previous statement:

```
If Reply2 = "Panther who?" Then _
    MsgBox "Panther no panths I'm going swimming."
If Reply2 <> "Panther who?" Then MsgBox "Incorrect answer. Try again."
```

The first line checks to see if the value stored in Reply2 is equal to the phrase "Panther who?" If it is, then the rest of the statement displays the joke's punch line. If the player typed something other than "Panther who?" then the second of these two statements executes, displaying a message that informs the player that he did not provide the correct response.

The Final Result

Now let's take a look at the fully assembled script.

```
Reply1 = InputBox("Knock Knock!")
If Reply1 = "Who's there?" Then
    Reply2 = InputBox("Panther!")
    If Reply2 = "Panther who?" Then _
        MsgBox "Panther no panths I'm going swimming."
    If Reply2 <> "Panther who?" Then MsgBox "Incorrect answer. Try again."
End If
If Reply1 <> "Who's there?" Then MsgBox "Incorrect answer. Try again."
```

As you can see, the script only has seven lines of code, and yet it displays multiple graphical pop-up dialog boxes that collect player text input and display any of three additional messages in pop-up dialog boxes. In addition, this script demonstrates one way of testing player input and then altering the execution of the script based on that input.

Save and then run the script, and make sure everything works as expected. If not, open the script and double-check each statement to make sure you typed it correctly.

Summary

This chapter has covered a lot of ground for an introductory chapter. Not only did you create your first VBScript, but you also learned how to use the WSH to execute it and to incorporate WSH elements within your scripts. In addition, you learned a lot about VBScript and how it relates to other languages that make up the Visual Basic family of programming languages. Finally, you created your first computer game, learning how to collect and validate user input and to display output. All in all, I'd say that this has been a very good start.

Challenges

1. The Knock Knock game is a very simple game. Its main purpose was to introduce you to the basics of script and game development. Try to improve the game by adding additional jokes so that the game does not end after the first joke.
2. Try running the Knock Knock game using both the CScript.exe and WScript.exe WSH execution hosts. How does the execution of the script change and why?
3. See if you can create a new script that prompts you for your name and then displays a personalized greeting message that includes your name. Hint: When displaying the customized greeting message, you will need to concatenate (glue together) the name of the user with a greeting message as follows:

```
MsgBox "Greetings " & UserName
```

This page intentionally left blank

2

An Introduction to the Windows Script Host

Because VBScripts cannot execute without an execution host of some type, the WSH is at the heart of any VBScript that you run from the Windows desktop or command line. The WSH not only provides an environment in which VBScripts can execute, but it also provides scripts with direct access to Windows resources such as the Windows desktop, Start menu, Registry, event logs, and network resources. To effectively create and execute VBScripts in this environment, it's helpful to know a little something about the WSH core object model. It is also important that you know how to configure the WSH to best suit your needs. In this chapter, you will learn the following:

- About the objects that make up the WSH core object model
- How to configure scripts for command-line execution
- How to configure scripts for desktop execution
- How to override host execution settings
- How to enable and disable the WSH

Project Preview: The Rock, Paper, and Scissors Game

In this chapter, you will learn how to create a computer version of the Rock, Paper, and Scissors game that you played as a child. The game begins by displaying its rules and then asks the player to choose between one of the three possible choices. After the player makes a selection, the game randomly makes its own selection and displays the results. Figures 2.1 through 2.3 demonstrate the flow of the game.

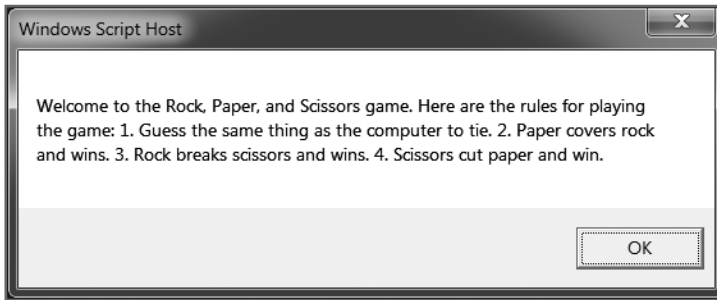


Figure 2.1 The script begins by displaying the rules of the game.

© 2014 Cengage Learning.



Figure 2.2 The player then types in a selection.

© 2014 Cengage Learning.

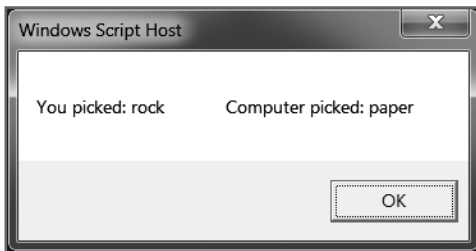


Figure 2.3 The script randomly picks a selection and displays the results of the game. © 2014 Cengage Learning.

Through the development of this game, you'll get a chance to practice incorporating WSH objects and their methods into VBScripts. You'll also learn how to perform a little simple conditional logic, as well as take a peek at using a number of built-in VBScript functions.

Examining Scripting Environments

As discussed in Chapter 1, "Getting Started with the WSH and VBScript," this book's primary focus is on teaching you how to program through the development of VBScripts executed using the WSH. VBScript, like JScript, is an ActiveX scripting engine. A scripting engine provides the operating system with access to a dynamic link library (DLL), which supports a given scripting language. In the case of VBScript, the DLL is VBScript.dll. (For JScript, it is JScript.dll.) These DLL files are stored in C:\Windows\System32. The DLL itself never executes. Instead, it is called by an execution host like those provided by IE and the WSH.

Before being adapted to work with the WSH, VBScript was used as a Web-scripting language designed to support script execution within Internet Explorer and Internet Explorer-compatible Web browsers. Internet Explorer provides VBScript with an environment that gives it access to things like forms, frames, and various browser controls via the browser's Document Object Model (DOM).

Like Internet Explorer, the WSH is also a programming environment that supports VBScript execution. Rather than supporting the browser DOM, the WSH provides scripts with access to local computer resources like the Windows Registry and file system through its own core object model. Whereas Internet Explorer runs as a full-blown Windows application, consuming all resources that such an application requires (including the memory required to formulate and maintain a graphical user interface), the WSH has been streamlined to do just one thing: host and execute scripts. The WSH, therefore, requires fewer computer resources to execute your scripts. There is no GUI to load. Less memory and fewer CPU cycles are required to execute your scripts. As a result, your scripts execute more quickly.

In Chapter 14, "Adding a GUI to Your Scripts," you will learn how to put your scripting skills to use in the development of HTML Applications. This will enable you to develop scripts that look and operate like desktop applications. This third scripting environment provides your scripts with the ability to access both the Internet Explorer DOM as well as the WSH core object model. The end result is the ability to use the DOM to build graphical user interfaces for your scripts while still allowing them to access and manipulate Windows resources via the WSH.

An Examination of WSH Components

Think of a computer, its operating system, and its hardware and software as being a collection of objects such as files, disk drives, printers, and so on. To automate tasks on Windows operating systems, VBScript needs a way of interacting with these objects. This is provided by the WSH's core object model.

An understanding of the WSH core object model is essential to your success as a VBScript programmer. Not only will it provide the technical insights you'll need to develop scripts that will run on Windows operating systems, but by introducing you to working with objects, it will also prepare you to work with other object models. For example, many Windows applications, including Microsoft Office applications, expose their own object models, allowing VBScript to programmatically manipulate them. In addition, other VBScript execution hosts, such as Internet Explorer, provide VBScript with access to other object models. The WSH core object model is complex and may at first seem rather daunting. But don't worry—you'll continue to develop your understanding of this complex topic as you go through the rest of this book. By the time you are finished, you should have a solid grasp of the WSH core object model.

Definition

An *object model* is a representation of a number of related objects that provide a script or program with the capability to view and interact with each of the objects (files, disks, printers, and so on) represented in the object model.

A Quick Introduction to the WSH Core Object Model

The WSH core object model provides access to 14 core objects through which you can programmatically access different types of Windows resources. The objects that make up the WSH core object model are depicted in Figure 2.4. Each of these objects provides you with access to a specific category of Windows resources.

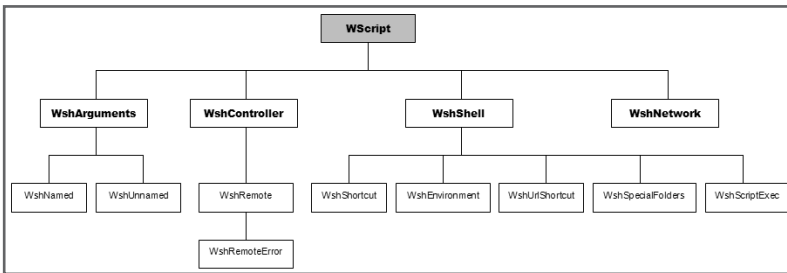


Figure 2.4 Each of these 14 objects has properties and methods that expose different Windows operating system resources.

© 2014 Cengage Learning.

The WScript object lies at the top, or *root*, of the WSH core object model. All other WSH core object model objects are created, or instantiated, from this object. The WScript object is automatically created for you when the execution host starts and can therefore be referenced without first being instantiated within your scripts. Once instantiated, you can work with the properties and methods belonging to any of the objects that comprise the WSH core object model. For example, the following statement demonstrates how to create a one-line script called Greeting.vbs that uses the WScript object's Echo method to display a text string.

Definition

Instantiation describes the process of creating a new instance of an object.

```
WScript.Echo "Example: Using the WScript object's Echo() method"
```

To test this script, open your script editor and type this statement. Then save the script (with a .vbs file extension), and run it by double-clicking it. The pop-up dialog box, shown in Figure 2.5, should appear. As this script demonstrates, you can automatically access any of the properties and methods belonging to the WScript object directly from within your scripts.

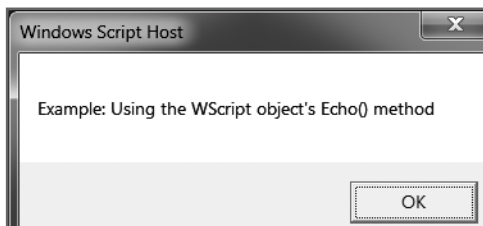


Figure 2.5 A pop-up dialog box created using the WScript object's Echo() method.

© 2014 Cengage Learning.

Working with the WScript Object

The `WScript` object is an extremely important object that you will find yourself working with all the time in your VBScripts. It provides access to a number of very useful methods that you'll see used throughout this book. These methods include the following:

- `CreateObject()`. This establishes an instance of the specified object.
- `DisconnectObject()`. This prevents a script from accessing a previously instantiated object.
- `Echo()`. This displays a text message in the Windows Console or as a pop-up dialog box, depending on which execution host runs the script.
- `Quit()`. This terminates a script's execution.
- `Sleep()`. This pauses the execution of a script for a specified number of seconds.

The `WScript` object is referred to as a public or exposed object. The WSH core object model has three other public objects: `WshController`, `WshShell`, and `WshNetwork`. Each of these three objects must be instantiated within your scripts using the `WScript` object's `CreateObject()` method. All the other objects in the WSH core object model can be instantiated only by using properties or methods associated with the `WScript`, `WshController`, `WshShell`, and `WshNetwork` objects.

Hint

You will see plenty more examples of how to work with the `WScript` object and most of the other objects that comprise the WSH core object model as you make your way through this book. In addition, you will find a detailed technical overview of all 14 of these objects, including their methods and properties, in Appendix C, "The WSH Core Object Model." As a sneak peek of what lies ahead, you may want to jump to Appendix C now and spend a few minutes looking it over to get a better feeling of the steps involved in instantiating objects and then working with their properties and methods.

Configuring WSH Execution Hosts

So far you've used the `CScript.exe` and `WScript.exe` execution hosts' default settings for each script that you've created and run. If you want to, you can modify these default settings to better suit your personal preferences. The WSH provides separate configuration settings for the `WScript.exe` and `CScript.exe` execution hosts. Because the `WScript.exe` execution host can process scripts run from either the Windows GUI or the Windows command line, there are two different ways to configure it. The WSH also allows you to override execution host settings on the fly by passing configuration arguments to the execution host when starting a script's execution. Finally, the `WScript.exe` execution host allows you to set execution host settings unique to a particular script using a WSH file. Each of these execution host configuration options is examined in detail in the sections that follow.

Configuring WScript.exe and CScript.exe Command-Line Execution

You can use either the WScript.exe or CScript.exe execution host to run any VBScript. Generally speaking, you'll use the WScript.exe execution host to run scripts that need to use pop-up dialog boxes and the CScript.exe execution host to run scripts silently in the background.

Even though they have their own separate configuration settings, both the WScript.exe and CScript.exe execution hosts are configured in the same way, using the exact same set of options. The syntax used to configure these two execution hosts is as follows:

```
wscript [//options]
cscript [//options]
```

Begin by opening a Windows Console. At the Windows command prompt, type the name of the execution host that you want to configure followed by one or more options, each of which is preceded by the // characters.

Trap

Any changes you make to the default execution host will affect your scripts only. If you share a computer with another user, that person's WSH execution host settings will not be affected. If you want WSH settings to be standardized for all users of the computer, make sure each user sets them accordingly.

Table 2.1 lists the configuration options supported by the WScript.exe and CScript.exe execution hosts.

TABLE 2.1 COMMAND-LINE OPTIONS FOR THE WSCRIPT.EXE AND CSCRIPT.EXE EXECUTION HOSTS

Configuration Option	Purpose
//?	Displays the command syntax for the CScript.exe and WScript.exe execution hosts.
//b	Runs a script in batch mode, where all errors and message output are suppressed.
//d	Turns on script debugging.
//e:jscript e:vbscript	Sets the script engine that is to be used to run the script.
//h:wscript h:cscript	Sets the execution host that is to be used to run the script.
//i	Runs the script interactively, displaying all errors and message output.
//job:id	Identifies a specific job within a Windows script file to be run.
//logo	Displays the CScript or WScript logo at the start of script execution.
//nologo	Suppresses the display of the CScript or WScript logo at the start of script execution.
//s	Saves the currently specified options and sets them as the default settings.
//t:nn	Establishes a timeout value that limits how long a script can execute. By default, these are not execution time limits imposed on script execution.
//x	Turns off script debugging.

Now let's look at some examples of how to modify the configuration of the execution hosts. By default, the WSH sets WScript.exe as the default execution host. However, you can change this by typing the following command and pressing the Enter key:

```
cscript //H:cscript //s
```

The //H:cscript option makes CScript.exe the default execution host and the //s option makes the change permanent. If you left the //s option off the command, the change would be in effect for your current working session only.

To change the default command-line execution host back to WScript.exe, type the following command and press the Enter key:

```
wscript //H:wscript //s
```

Now let's try an example that sets more than one configuration option:

```
wscript //H:cscript //nologo //t:60 //s
```

In this example, the CScript.exe execution host is set as the default. In addition, the //nologo option prevents the display of the WScript logo during script execution. It also prevents the display of the following text just before any output text when scripts are executed from the command line:

```
Microsoft (R) Windows Script Host Version 5.8  
Copyright (C) Microsoft Corporation. All rights reserved.
```

The //t:60 option prevents any script from executing for more than 60 seconds. Finally, the //s option saves all specified settings.

Trick

Even the best programmers can make mistakes. Sometimes these mistakes cause scripts to behave in unexpected ways, such as being stuck in a loop that executes forever. By setting the //T:nn option for both the WScript.exe and CScript.exe execution hosts, you can set up a sort of safety net that prevents any script that you run from executing for more than a specified amount of time.

Configuring WScript.exe Desktop Execution

The WScript.exe execution host's desktop configuration settings are different from its command-line configuration settings. For one thing, there are only two configuration settings. The first specifies an optional time limit for script execution, and the second specifies whether the WScript logo is displayed when scripts are run from the Windows Console.

The steps involved in configuring the WScript.exe execution host from the Windows desktop are as follows:

1. Open a Windows Console.
2. Type WScript and then click OK. The Windows Script Host Settings dialog box appears, as shown in Figure 2.6.

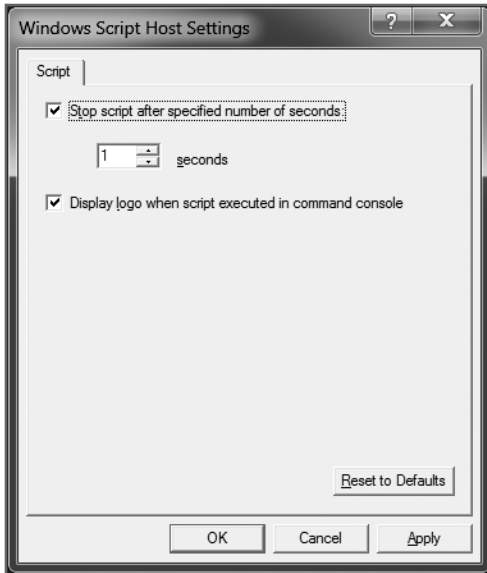


Figure 2.6 Modifying the WScript.exe execution host's desktop configuration. © 2014 Microsoft Corporation. Used with permission from Microsoft.

3. By default, the WScript.exe execution host does not have an execution time setting. To configure one, select the Stop Script After Specified Number of Seconds checkbox and then specify a time limit (in seconds).
4. By default, the WScript.exe execution host displays its logo when scripts are executed from the Windows Console. To prevent this behavior, deselect the Display Logo When Script Executed in Command Console checkbox.
5. Click OK.

The configuration settings established through this procedure are stored in the following registry key: `HKLM\Software\Microsoft Script Host\Settings`.

Overriding Command-Line Host Execution Settings

So far, you've learned how to configure the default execution of the WScript.exe and CScript.exe execution hosts from the Windows command line and desktop. Now let's see how to override the default command-line settings, without permanently changing them, to temporarily alter them for the execution of a specific script.

The syntax for temporarily overriding WScript.exe and CScript.exe execution is as follows:

```
wscript scriptname [//options] [arguments]
cscript scriptname [//options] [arguments]
```

First, open the Windows Console and type the name of the execution host you want to use to run the script. Next, type the name and path of the script to be executed. Then type as many configuration settings as you want, preceding each with a pair of // characters. If the script you're executing expects any input to be passed to it at execution time, specify the required arguments. Finally, press the Enter key.

Now let's look at a few examples of how to override host script execution settings. Let's assume you're working with a script called Test.vbs and you want to prevent it from executing for more than 30 seconds. Open the Windows Command Console and type the following command to run the script using the WScript.exe execution host:

```
wscript Test.vbs //T:30
```

To execute the same script using the CScript.exe execution host for a maximum of 30 seconds, type the following command:

```
cscript Test.vbs //T:30
```

Now let's look at a slightly more complicated example, in which multiple configuration settings are overridden:

```
wscript Test.vbs //T:30 //nologo
```

In this example, the script is prevented from executing for more than 30 seconds using the WScript.exe execution host. In addition, the WScript.exe execution host's logo is suppressed to prevent it from being displayed at the beginning of the script's execution.

Customizing WScript.exe Settings for Individual Desktop Scripts

The WSH also provides a way, using the WScript.exe execution host, to permanently override configuration settings for specific scripts run from the Windows desktop. This is done by creating a text file with the same name as the script and giving the file a .wsh extension. Then, within the WSH file, you can specify WSH configuration settings. For example, to set up a WSH file for a script named Test.vbs, you would create a file called Test.wsh and save it in the same folder in which the Test.vbs script resides. You could then run the script by double-clicking the WSH file or by double-clicking the script itself. If you double-click the WSH file, the WSH automatically finds the script that is associated with it and, after processing the configuration settings stored in the WSH file, runs the script. Conversely, whenever you double-click a script, the WSH first looks to see if it has an associated WSH file before running it. If it does not, then the WSH processes it using the execution host's default configuration settings.

Definition

In the context of this discussion, an *argument* is a piece of data passed to a script for processing. For example, if you wrote a VBScript to create new user accounts, your script might expect you to pass it one or more usernames to process.

The following statements show the contents of a typical WSH file:

```
[ScriptFile]
Path=C:\Test.vbs
[Options]
Timeout=30
DisplayLogo=0
```

The first line contains the section label called `[ScriptFile]`. The next statement provides the name and path of the script associated with this WSH file. Next comes an `[Options]` section label. The last two lines contain configuration settings specific to the execution of this script. `Timeout=30` specifies that this script will not be allowed to process for more than 30 seconds, and `DisplayLogo=0` specifies that the WScript logo is to be suppressed. An alternative setting for this option would be `DisplayLogo=1`, which would enable the display of the WScript logo.

There are two ways to create a WSH file. One is to use a text editor, such as Windows Notepad, to manually create the file. The other is to let Windows create the WSH file for you using the following procedure:

1. Locate the folder in which the VBScript is stored.
2. Right-click the script and select Properties from the menu that appears. The script's Properties dialog box opens, as shown in Figure 2.7.

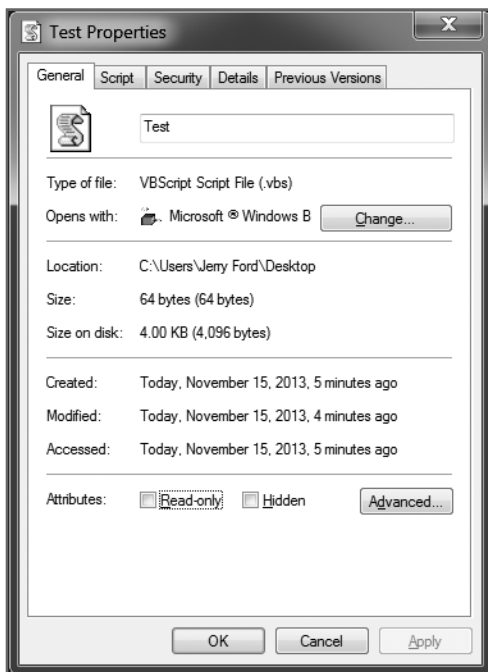


Figure 2.7 Examining the properties on the General tab of the script's Properties dialog box.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

3. Click the Script tab, as shown in Figure 2.8.

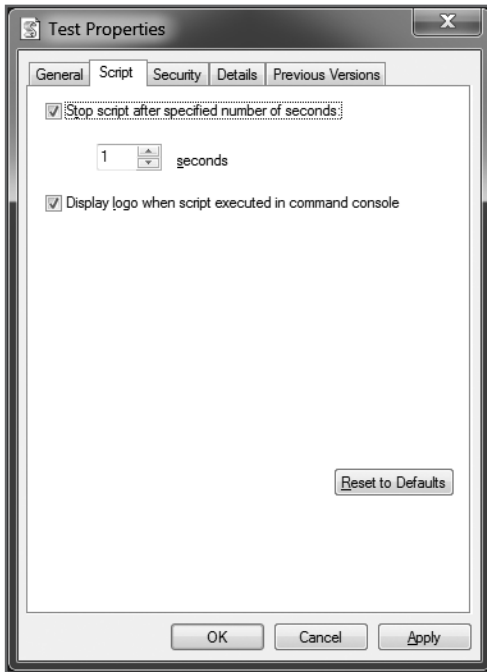


Figure 2.8 Configuring a WSH file via a script's Properties dialog box. © 2014 Microsoft Corporation. Used with permission from Microsoft.

4. Specify a script execution time limit, as required.
5. Enable or disable the display of the WScript logo as desired.
6. Click OK.

A new WSH file is created for you and stored in the same folder as the script.

Enabling and Disabling the Windows Script Host

It is no secret that today's computers exist under the threat of viruses, worms, and other pesky computer programs designed to spy on, disrupt, or otherwise harm or even take over your computer. In the past, clever people exploited the power and convenience of the WSH to spread and infect computers using scripts. Many defenses have been developed to combat these threats. For example, several email providers scan emails and email attachments for threats and block them if a threat is discovered. Likewise, anti-virus programs can be installed to help protect computers from these threats. Firewalls can also be used to protect computers or entire computer networks.

Despite these protections and precautions, threats still persist. One way people have dealt with them in the past has been to try to block the unauthorized execution of programs and scripts. In the case of the WSH, some people have disabled it.

If you have inherited a computer with the WSH disabled, you will receive the following notification the first time you attempt to run a VBScript using the WSH:

```
CScript Error: Windows Script Host access is disabled on this machine. Contact your administrator for details.
```

Re-enabling the WSH

If the WSH has been disabled on your computer, you can re-enable it using one of the following procedures. However, doing so requires administrative-level privileges. If you lack these privileges, you will need to seek out your computer administrator and request that this procedure be performed. The first procedure will re-enable WSH only for yourself when you are logged on to the computer. The second procedure will re-enable WSH execution for any user logged on to the computer.

To re-enable the WSH for an individual user, do the following:

1. Click the Start button, type `Regedit` in the Search field, and press `Enter` to open the Regedit utility.
2. Click `Yes` when prompted by the User Account Control dialog box.
3. Navigate to the following key: `HKEY_CURRENT_USER\Software\Microsoft\Windows Script Host\Settings`.
4. Under `DWORD`, look for an `Enabled` entry. If it is present, double-click it and change its assigned value to `1`. If it is not present, right-click the `Settings` key and select `New > DWORD`. Then type `Enabled` as its name. Finally, double-click the `Enabled` `DWORD` entry and assign it a value of `1`.
5. Close Regedit.

To re-enable the WSH for all users of a computer, do the following:

1. Click the Start button, type `Regedit` in the Search field, and press `Enter` to open the Regedit utility.
2. Click `Yes` when prompted by the User Account Control dialog box.
3. Navigate to the following key: `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Script Host\Settings`.
4. Under `DWORD`, look for an `Enabled` entry. If it is present, double-click it and change its assigned value to `1`. If it is not present, right-click the `Settings` key and select `New > DWORD`. Then type `Enabled` as its name. Finally, double-click the `Enabled` `DWORD` entry and assign it a value of `1`.
5. Close Regedit.

Disabling the WSH

If, after enabling the WSH on your computer, you decide that you need to return it to a disabled state, you can do so using one of the following procedures. The first procedure blocks WSH execution only when you are logged on to the computer. The second procedure affects any logged in user of the computer.

To disable the WSH for an individual user, follow these steps:

1. Click the Start button, type `Regedit` in the Search field, and press the Enter key to open the Regedit utility.
2. Click Yes when prompted by the User Account Control dialog box.
3. Navigate to the following key: `HKEY_CURRENT_USER\Software\Microsoft\Windows Script Host\Settings`.
4. Right-click the key and select `New > DWORD`. Then type `Enabled` as its name. By default, a value of 0 (representing a value of False) is assigned to this entry.
5. Close Regedit.

To disable the WSH for all users of a computer, follow these steps:

1. Click the Start button, type `Regedit` in the Search field, and press the Enter key to open the Regedit utility.
2. Click Yes when prompted by the User Account Control dialog box.
3. Navigate to the following key: `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Script Host\Settings`.
4. Right-click the key and select `New > DWORD`. Then type `Enabled` as its name. By default, a value of 0 (representing a value of False) is assigned to this entry.
5. Close Regedit.

Back to the Rock, Paper, and Scissors Game

Now it's time to go back to where this chapter started—talking about the Rock, Paper, and Scissors game. In this project, you will create a scripted version of this classic game. This version is a bit limited, given that you've not yet had the chance to learn everything you'll need to create a more sophisticated version. However, you know enough to build the game's foundation and get a working model going. In Chapter 5, "Conditional Logic," you'll return to this project and spice things up a bit.

Designing the Game

The basic design of this game is simple. First, display the rules of the game, and then ask the player to type rock, paper, or scissors. Next, have the script randomly pick a choice of its own and display the results.

This project is completed in six steps:

1. Define the resources used by this script.
2. Display the game's instructions.
3. Provide a way for the user to select a choice.

4. Devise a way for the script to generate a random number.
5. Assign the computer's choice based on the script's randomly selected number.
6. Display the final results of the game.

Defining the Resources Used by the Script

Begin by opening your editor and saving a blank file with a name of `RockPaperScissors.vbs`. Next, add the first few lines of the script as follows:

```
'Formally declare variables used by the script before trying to use them  
  
Dim WshShl, Answer, CardImage  
  
'Create an instance of the WScript object in order to later use the  
'Popup method  
Set WshShl = WScript.CreateObject("WScript.Shell")
```

Notice that the first line begins with a `'` character. This character identifies a VBScript comment. Comments can be used to document the contents of scripts. Comments have no effect on the execution of a script.

The next line begins with the VBScript keyword `Dim`. This statement defines three variables that will be used by the script. A *variable* is simply a portion of the computer memory where your scripts can store and retrieve data. I'll provide more information about variables and how they work in Chapter 4, "Constants, Variables, Arrays, and Dictionaries."

The third statement is another comment, and the fourth statement uses the `WScript` object's `CreateObject()` method to set up an instance of the `WshShell` object. This statement allows the script to access `WshShell` properties and methods.

Displaying the Rules of the Game

Next, let's take advantage of the `WshShell` object that you just defined by using its `Popup()` method to display a message in a graphical pop-up dialog box:

```
'Display the rules of the game  
WshShl.Popup "Welcome to the Rock, Paper, and Scissors game. Here are the " & _  
"rules of the game: 1. Guess the same thing as the computer " & _  
"to tie. 2. Paper covers rock and wins. 3. Rock breaks " & _  
"scissors and wins. 4. Scissors cut paper and win."
```

This is really just two lines of code, although it looks like five. The first line is a comment. The second line was so big that I chose to break it down into multiple pieces for display purposes. To do so, I broke the message that I wanted to display into multiple segments of similar lengths, placing each segment within a pair of quotation marks. To tie the different segments into one logical statement, I added the VBScript `&` character to the end of each line, followed by the `_` character.

Collecting the Player's Selection

When the player clicks OK, the pop-up dialog box displaying the game's rules disappears and is replaced with a new pop-up dialog box that is generated by the following code:

```
'Prompt the user to select a choice
Answer = InputBox("Type Paper, Rock, or Scissors.", _
    "Let's play a game!")
```

The first statement is a comment and can be ignored. The second statement uses the VBScript `InputBox()` function to display a pop-up dialog box into which the user can type “rock,” “paper,” or “scissors.” The value typed by the user is then assigned to a variable called `Answer`.

Setting Up the Script's Random Selection

Now that the player has made a choice, it's the script's turn to make a random selection on behalf of the computer. This can be done in two statements, as shown by the following statements:

```
'Time for the computer to randomly pick a choice
Randomize
GetRandomNumber = Int((3 * Rnd()) + 1)
```

The first line is a comment and can be ignored. The second line executes the `Randomize` statement, which ensures that the computer generates a random number. If you leave this line out and run the script several times, you'll notice that after making an initial random choice, the script always makes the exact same choice time and time again. The `Randomize` statement prevents this behavior by ensuring that a random number is generated each time the script executes.

The next statement generates a random number between 1 and 3. I'll break down the activity that occurs in this statement. First, the `Rnd()` function generates a random number between 0 and 1. Next, the `Int()` function, which returns the integer portion of a number, executes, multiplying 3 times the randomly generated number and then adding 1 to it. The final result is a randomly generated number with a value between 1 and 3.

Assigning a Choice to the Script's Selection

Next, you'll need to assign a choice to each of the three possible numeric values randomly generated by the script:

```
'Assign a value to the randomly selected number
If GetRandomNumber = 3 then CardImage = "rock"
If GetRandomNumber = 2 then CardImage = "scissors"
If GetRandomNumber = 1 then CardImage = "paper"
```

If the number 3 is generated, then a value of `rock` is assigned as the computer's selection. If the number 2 is generated, then a value of `scissors` is assigned as the computer's selection. Finally, if the number 1 is generated, then a value of `paper` is assigned as the computer's selection.

Displaying the Results of the Game

After the script comes up with the computer's selection, it's time to display the results of the game so that the user can see who won:

```
'Display the game's results so that the user can see if he or she won
WshShl.Popup "You picked: " & Answer & Space(12) & "Computer picked: " & _
    CardImage
```

The WshShell object's Popup() method is used to display the results of the game. Using the & concatenation character, I pieced together the various parts of the message. These parts included text phrases enclosed within quotation marks; the Answer variable; the CardImage variable, which represents the user's and computer's choices; the Space() method, which adds 12 blank spaces to the text messages; and the _ character, which allows me to spread the message out over two separate lines.

The Final Result

Now let's put all the pieces of the script together and then save and run the script:

```
'Formally declare variables used by the script before trying to use them
Dim WshShl, Answer, CardImage

'Create an instance of the WScript object in order to later use the
'Popup method
Set WshShl = WScript.CreateObject("WScript.Shell")

'Display the rules of the game
WshShl.Popup "Welcome to the Rock, Paper, and Scissors game. Here are the " & _
    "rules for playing the game: 1. Guess the same thing as the " & _
    "computer to tie. 2. Paper covers rock and wins. 3. Rock breaks " & _
    "scissors and wins. 4. Scissors cut paper and win."

'Prompt the user to select a choice
Answer = InputBox("Type Paper, Rock, or Scissors.", _
    "Let's play a game!")

'Time for the computer to randomly pick a choice
Randomize
GetRandomNumber = Round(FormatNumber(Int((3 * Rnd()) + 1)))

'Assign a value to the randomly selected number
If GetRandomNumber = 3 then CardImage = "rock"
```

```
If GetRandomNumber = 2 then CardImage = "scissor"  
If GetRandomNumber = 1 then CardImage = "paper"
```

```
'Display the game's results so that the user can see if he or she won  
WshShl.Popup "You picked: " & Answer & Space(12) & "Computer picked: " & _  
CardImage
```

Summary

In this chapter, you were introduced to the WSH core object model and its 14 objects. You were introduced to the WScript object and learned how to use it to display text messages in pop-up dialog boxes. You also learned how to configure both the WScript.exe and CScript.exe execution hosts to best suit your personal requirements and preferences. This included learning how to configure scripts for command-line and desktop execution and to customize settings for individual desktop scripts. You also learned how to enable and disable the Windows script host.

Challenges

1. See whether you can expand RockPaperScissors.vbs by adding logic that compares the player's selection to the script's random selection to determine the winner.
2. Based on the preceding comparison, display a little additional text that explicitly states the result of the game—for example, "You Win!," "You Lose!," or "Tie!"

This page intentionally left blank

II

Learning VBScript and WSH Scripting

Chapter 3: VBScript Basics

**Chapter 4: Constants, Variables, Arrays,
and Dictionaries**

Chapter 5: Conditional Logic

Chapter 6: Processing Collections of Data

**Chapter 7: Using Procedures to Organize
Scripts**

This page intentionally left blank

3

VBScript Basics

This chapter begins your VBScript education by teaching you a number of important concepts. You'll learn about the objects that make up the VBScript core and run-time object models. In addition, you'll learn about basic VBScript syntax, functions, reserved words, and special characters. You'll also learn about VBScript and WSH output functions and methods. Along the way, you'll create a math game while learning more about how VBScript works with the WSH. You also will learn the following:

- The basic rules that you must follow when writing VBScripts
- The objects that make up the VBScript core and run-time object models
- How to enhance your scripts using built-in VBScript functions
- Different ways of displaying script output

Project Preview: The Math Game

This chapter's game project shows you a programming technique that enables you to write VBScripts that can open and interact with other Windows applications. It's called `MathGame.vbs`, and it tests the player's understanding of the principle of *precedence* in solving a numeric expression. If the user gets the answer correct, he or she is congratulated for possessing superior math skills. If the player provides an incorrect answer, then the game offers to teach the player how to solve the expression.

To teach the player how to solve the equation, the program opens the Microsoft WordPad application and types out instructions that explain the steps required to solve the problem. To further demonstrate how the equation is solved, the program starts the Windows Calculator application and uses it to solve the equation.

The WordPad and Calculator demonstrations play out almost like a movie or slide show, starting automatically, pausing as input and text are automatically keyed in, and finally automatically closing when the demonstrations end. Figures 3.1 through 3.4 show some of the screens that users will see when they play the Math Game on a computer running Windows 8.1.

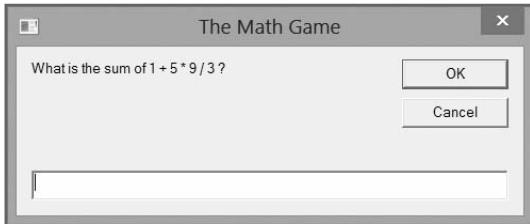


Figure 3.1 The game begins by asking the player to solve a mathematical equation. © 2014 Cengage Learning.

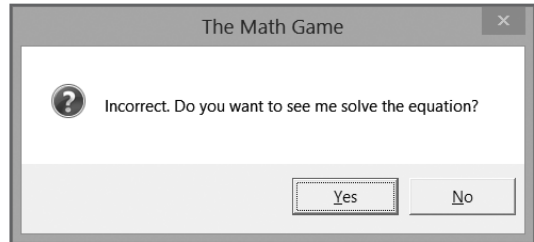


Figure 3.2 If the player provides an incorrect answer, the game offers to demonstrate how the equation is solved. © 2014 Cengage Learning.

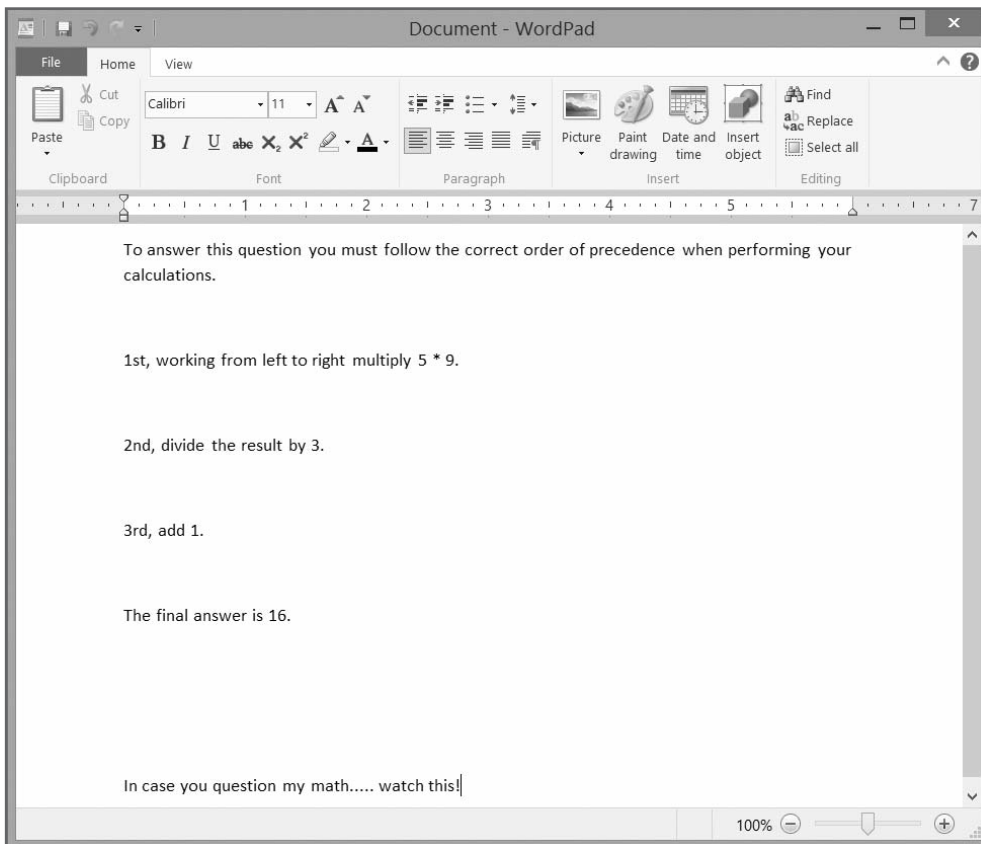


Figure 3.3 Using WordPad, the game types out detailed instructions for solving the problem.



Figure 3.4 The game then starts the Calculator application and solves the equation again, just for fun.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

As you go through the steps involved in creating this game, you'll learn how to use a number of `WScript` and `WshShell` object methods. You'll also get a brief introduction to VBScript's support for conditional programming logic.

VBScript Statements

Like any programming language, VBScript is composed of programming statements. As you go through the chapters in this book, you'll be introduced to the statements that make up the VBScript's scripting language, learning a few different statements in every chapter, until, by the end of the book, you've seen and worked with most of them.

Table 3.1 lists the statements that make up the VBScript scripting language.

TABLE 3.1 VBSCRIPT STATEMENTS

Statement	Description
Call	Executes a procedure
Class	Defines a class name
Const	Defines a constant
Dim	Defines a variable
Do...Loop	Repeatedly executes a collection of one or more statements as long as a condition remains True or until the condition becomes True
Erase	Reinitializes the elements stored in an array
Execute	Executes a specified statement

TABLE 3.1 VBSCRIPT STATEMENTS (CONTINUED)

Statement	Description
ExecuteGlobal	Executes a specified statement in a script's global namespace
Exit	Ends a loop, subroutine, or function
For Each...Next	Processes all the elements stored in an array or collection
For...Next	Repeats a collection of one or more statements a specified number of times
Function	Defines a function and its associated arguments
If...Then...Else	Executes one or more statements depending on the value of a tested condition
On Error	Enables error handling
Option Explicit	Forces explicit variable declaration
Private	Defines a private variable
Property Get	Defines a property name and its arguments and then returns its value
Property Let	Defines a property procedure's name, its arguments, and code that allows the assignment of a property's value
Property Set	Defines a property procedure's name, its arguments, and code that allows the assignment of a property as a reference to an object
Public	Defines a public variable
Randomize	Initializes VBScript's random-number generator
ReDim	Defines or redefines the dimension of an array
Rem	A comment statement
Select Case	Defines a group of tests, of which only one will execute if a matching condition is found
Set	Sets up a variable reference to an object
Sub	Defines a subroutine and its arguments
While...Wend	Executes one or more statements as long as the specified condition is True
With	Associates one or more statements that are to be executed for a specified object

© Jerry Lee Ford, Jr. All Rights Reserved.

VBScript Syntax Rules

To properly apply the programming statements that make up the VBScript programming language, you must have an understanding of the syntax rules that govern these statements. Each VBScript statement has its own particular syntax.

The following is a list of rules that you should keep in mind as you write your VBScripts:

- By default, all VBScript statements must fit on one line.
- You can spread a single statement out over multiple lines by ending each line with the `_` (continuation) character.
- You can place more than one VBScript statement on a single line by ending each statement with the `:` (colon) character.
- By default, VBScript is not case sensitive, meaning that VBScript regards different case spelling of words used by variables, constants, procedures, and subroutines as the same.
- You can enforce case sensitivity by adding the `Option Explicit` statement to the beginning of your VBScripts.
- By default, an error will halt the execution of any VBScript.
- You can prevent an error from terminating a VBScript's execution by adding the `On Error Resume Next` statement to your VBScripts.
- Extra blank spaces are ignored within scripts and can be used to improve scripts' format and presentation.

Every VBScript statement has its own specific syntax that must be exactly followed. Failure to properly follow a statement's syntax will result in an error. Let's look at an example. The following statement tries to use the VBScript's `MsgBox()` function to display a text message:

```
MsgBox "Thanks for playing!
```

Unfortunately, the statement's syntax requirements have not been followed. The `MsgBox()` function requires that all text messages be enclosed within a pair of quotation marks. If you look closely, you will see that the closing quotation mark is omitted. Figure 3.5 shows the error produced on a computer running Windows 8.1 by this statement at run-time.

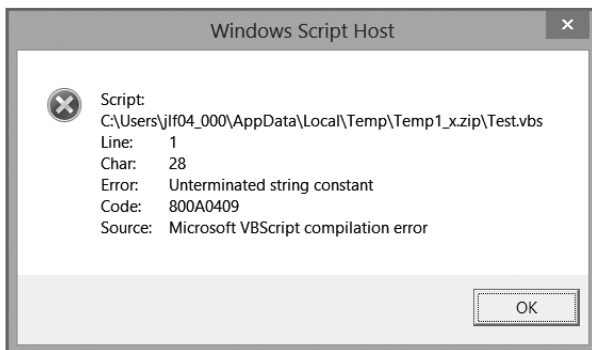


Figure 3.5 The error message caused by an unmatched quotation mark in a `MsgBox()` statement.

Reserved Characters

Like any programming language, VBScript has a collection of reserved words. *Reserved words* are words that you cannot use within your scripts because VBScript also assigns a special meaning to them. Some of these words are reserved because they are part of the language itself. Others are reserved for future use. Table 3.2 lists VBScript's reserved words. The important thing to remember when it comes to VBScript reserved words is that you can only use them as intended (that is, you cannot use them as variables, constants, or procedure names).

TABLE 3.2 VBSCRIPT'S COLLECTION OF RESERVED WORDS

And	EndIf	LSet	RSet
As	Enum	Me	Select
Boolean	Eqv	Mod	Set
ByRef	Event	New	Shared
Byte	Exit	Next	Single
ByVal	False	Not	Static
Call	For	Nothing	Stop
Case	Function	Null	Sub
Class	Get	On	Then
Const	GoTo	Option	To
Currency	If	Optional	True
Debug	Imp	Or	Type
Dim	Implements	ParamArray	TypeOf
Do	In	Preserve	Until
Double	Integer	Private	Variant
Each	Is	Public	Wend
Else	Let	RaiseEvent	While
ElseIf	Like	ReDim	With
Empty	Long	Rem	Xor
End	Loop	Resume	

Adding Comments

One of the easiest VBScript statements to understand is the comment statement. The comment statement gives you the ability to add to your VBScripts descriptive text that documents why you wrote the script the way you did. Documenting your scripts with comments makes them easier to support and helps others who may come after you to pick up where you left off. Comments do not have any effect on the execution of your scripts and you should use them liberally.

Comments can be added to scripts using the VBScript Rem (short for “remark”) statement, as follows:

```
Rem Use the VBScript MsgBox() function to display a message
MsgBox "Thanks for playing!"
```

Comments also can be created using the ' character:

```
'Use the VBScript MsgBox() function to display a message
MsgBox "Thanks for playing!"
```

The ' character is my preferred style. I find it less visually intrusive and just as effective.

Also, you can add a comment to the end of any statement:

```
MsgBox "Thank you for playing" 'Display a thank you message
```

Hint

One sign of an experienced programmer is the number and usefulness of comments added to his scripts. Consider adding comments that describe the function of variables, constants, and arrays. Also use them to explain complicated pieces of coding.

Comments can also be used to create a script template, which will provide additional structure to your VBScripts. For example, consider the following template:

```
'*****
'Script Name: ScriptName.vbs
'Author: Author Name
'Created: MM/DD/YY
'Description: XXXXXXXXXXXXXXXXXXXXXXXX.
'*****

'Initialization Section
Option Explicit
On Error Resume Next
Dim...
Const...
```

```
Set...
```

```
'Main Processing Section
```

```
'Procedure Section
```

```
'This function...  
Function Xxxxx(Zzzz)  
    XXXXXXXXXXXX  
End Function
```

This template begins with a documentation section that provides a place to record the script's name, author, and creation date, as well as a brief description. Other information that you might want to add here includes the following:

- Instructions for running the script
- Documentation for any arguments the script expects to receive at execution time
- Documentation of the recent updates to the script, including when, by whom, and why
- Copyright information
- Contact or support information

The rest of the template is divided into three sections:

- **The initialization section.** This contains statements that globally affect the scripts, including `Option Explicit` and `On Error`, as well as the declaration of any variables, constants, arrays, and objects used by the script.
- **The main processing section.** This section contains the statements that control the main processing logic of the script. The statements in this section access the resources defined in the initialization section as necessary, and call on the procedures and functions located in the procedure section.
- **The procedure section.** This section contains all the script's procedures. *Procedures* are groups of statements that can be called and executed as a unit. You'll learn how to work with procedures in Chapter 7, "Using Procedures to Organize Scripts."

Mastering the VBScript Object Model

In Chapter 2, "An Introduction to the Windows Script Host," you learned about the WSH core object model and its properties and methods. You also learned how to instantiate WSH objects to access and manipulate their properties and methods. VBScript also provides two collections of objects that you can use in your scripts. Table 3.3 provides an overview of VBScript's built-in or core objects.

TABLE 3.3 VBSCRIPT BUILT-IN OBJECTS

Object Name	Description
Class	Provides access to class events
Err	Provides access to information about run-time errors
Match	Provides access to the read-only properties of a regular expression match
Matches Collection	A collection of regular expression Match objects
RegExp	Supports regular expressions
SubMatches Collection	Provides access to read-only values of regular expression submatch strings

© Jerry Lee Ford, Jr. All Rights Reserved.

Check out Chapter 11, “Working with Built-in VBScript Objects,” to learn more about VBScript’s built-in objects.

Working with VBScript Run-Time Objects

In addition to its core object model, VBScript’s `FileSystemObject` object also provides a number of run-time objects. As Table 3.4 shows, your scripts can use these objects and their properties and methods to interface with the Windows file system.

TABLE 3.4 VBSCRIPT RUN-TIME OBJECTS

Object Name	Description	Properties	Methods
Dictionary	Stores data keys, item pairs	Count, Item, Key	Add, Exists, Items, Keys, Remove, RemoveAll
Drive	Provides access to disk properties	AvailableSpace, DriveLetter, DriveType, FileSystem, FreeSpace, IsReady, Path, RootFolder, SerialNumber, ShareName, TotalSize, VolumeName	None
Drives Collection	Provides access to information regarding a drive’s location	Count, Item	None

TABLE 3.4 VBSCRIPT RUN-TIME OBJECTS (CONTINUED)

Object Name	Description	Properties	Methods
File	Provides access to file properties	Attributes, DateCreated, DateLastAccessed, DateLastModified, Drive, Name, ParentFolder, Path, ShortName, ShortPath, Size, Type	Copy, Delete, Move, OpenAsTextStream
Files Collection	Provides access to files stored in a specified folder	Count, Item	None
FileSystemObject	Provides access to the file system	Drives	BuildPath, CopyFile, CopyFolder, CreateFolder, CreateTextFile, DeleteFile, DeleteFolder, DriveExists, FileExists, FolderExists, GetAbsolutePathName, GetBaseName, GetDrive, GetDriveName, GetExtensionName, GetFile, GetFileName, GetFolder, GetParentFolderName, GetSpecialFolder, GetTempName, MoveFile, MoveFolder, OpenTextFile.
Folder	Provides access to folder properties	Attributes, DateCreated, DateLastAccessed, DateLastModified, Drive, Files, IsRootFolder, Name, ParentFolder, Path, ShortName, ShortPath, Size, SubFolders, Type	Copy, Delete, Move, OpenAsTextStream
Folders Collection	Provides access to folders located within another folder	Count, Item	Add

© Jerry Lee Ford, Jr. All Rights Reserved.

The WSH core object model provides access to a number of Windows resources. Absent from this model is a file system object. Therefore, to access system files from your VBScripts, you'll need to learn how to work with VBScript's `FileSystemObject` object.

With this object, your scripts will be able to do the following

- Check for the existence of files and folders before attempting to work with them.
- Create and delete files and folders.
- Open and read files.
- Write or append to files.
- Close files.
- Copy and move files and folders.

Properties

Like WSH objects, VBScript run-time objects support a large number of properties. Table 3.5 provides a complete list of VBScript run-time properties.

TABLE 3.5 VBSCRIPT RUN-TIME PROPERTIES

Property Name	Description
AtEndOfLine	Returns a value of either true or false based on whether the file pointer has reached the TextStream file object's end-of-line marker (TextStream is a built-in class that provides facilities for accessing files.)
AtEndOfStream	Returns a value of either true or false based on whether the end of a TextStream file object has been reached
Attributes	Modifies or retrieves file and folder attributes
AvailableSpace	Retrieves the amount of free space available on the specified drive
Column	Retrieves the current column position in a TextStream file object
CompareMode	Sets or returns the comparison mode used to compare a Dictionary object's string keys
Count	Returns a value representing the number of the items in a collection or Dictionary object
DateCreated	Retrieves a file or folder's creation date and time
DateLastAccessed	Retrieves the date and time that a file or folder was last accessed
DateLastModified	Retrieves the date and time that a file or folder was last modified
Drive	Retrieves the drive letter where a file or folder is stored
DriveLetter	Retrieves the specified drive's drive letter
Drives	Establishes a Drives collection representing all the drives found on the computer
DriveType	Returns a value identifying a drive's type

TABLE 3.5 VBSCRIPT RUN-TIME PROPERTIES (CONTINUED)

Property Name	Description
Files	Establishes a Files collection to represent all the File objects located within a specified folder
FileSystem	Retrieves the name of the file system used on the specified drive
FreeSpace	Retrieves the amount of free space available on the specified drive
IsReady	Returns a value of either true or false based on the availability of the specified drive
IsRootFolder	Returns a value of either true or false based on whether the specified folder is the root folder
Item	Retrieves or sets an item based on the specified Dictionary object key
Key	Sets a Dictionary object key
Line	Retrieves the current line number in the TextStream file object
Name	Gets or modifies a file or folder's name
ParentFolder	Returns a reference to the specified file or folder's parent folder object
Path	Retrieves the path associated with the specified file, folder, or drive
RootFolder	Retrieves the Folder object associated with the root folder on the specified drive
SerialNumber	Retrieves the specified disk volume's serial number
ShareName	Retrieves the specified network drive's share name
ShortName	Retrieves the specified file or folder's 8.3-character short name
ShortPath	Retrieves a file or folder's short path name associated with a file or folder's 8.3-character name
Size	Returns the number of bytes that make up a file or folder
SubFolders	Establishes a Folders collection made up of the folders located within a specified folder
TotalSize	Retrieves a value representing the total number of bytes available on a drive
Type	Retrieves information about the specified file or folder's type
VolumeName	Gets or modifies a drive's volume name

© Jerry Lee Ford, Jr. All Rights Reserved.

Methods

VBScript run-time objects also support a larger number of methods, which you will find essential when working with the Windows file system. These methods are outlined in Table 3.6.

TABLE 3.6 VBSCRIPT RUN-TIME METHODS

Method Name	Description
Add (Dictionary)	Adds a key and item pair to a Dictionary object
Add (Folders)	Adds a Folder to a collection
BuildPath	Appends a name to the path
Close	Closes an open TextStream file object
Copy	Copies a file or folder
CopyFile	Copies one or more files
CopyFolder	Recursively copies a folder
CreateFolder	Creates a new folder
CreateTextFile	Creates a file and a TextStream object so that it can be read from and written to
Delete	Deletes a file or folder
DeleteFile	Deletes a file
DeleteFolder	Deletes a folder's contents
DriveExists	Returns a value of true or false based on whether a drive exists
Exists	Returns a value of true or false based on whether a key exists in a Dictionary object
FileExists	Returns a value of true or false based on whether the specified file can be found
FolderExists	Returns a value of true or false based on whether the specified folder can be found
GetAbsolutePathName	Retrieves a complete path name
GetBaseName	Retrieves a file name without its file extension
GetDrive	Returns the Drive object associated with the drive in the specified path
GetDriveName	Returns the name of a drive
GetExtensionName	Returns a file's extension
GetFile	Returns a File object
GetFileName	Returns the last file name or folder of the specified path
GetFileVersion	Returns a file's version number
GetFolder	Returns the Folder object associated with the folder in the specified path
GetParentFolderName	Returns the name of the parent folder
GetSpecialFolder	Returns a special folder's name

TABLE 3.6 VBSCRIPT RUN-TIME METHODS (CONTINUED)

Method Name	Description
GetTempName	Returns the name of a temporary file or folder
Items	Returns an array where items in a Dictionary object are stored
Keys	Returns an array containing the keys in a Dictionary object
Move	Moves a file or folder
MoveFile	Moves one or more files
MoveFolder	Moves one or more folders
OpenAsTextStream	Opens a file and retrieves a TextStream object to provide a reference to the file
OpenTextFile	Opens a file and retrieves a TextStream object to provide a reference to the file
Read	Returns a string containing a specified number of characters from a TextStream file object
ReadAll	Reads the entire TextStream file object and its contents
ReadLine	Reads an entire line from the TextStream file object
Remove	Deletes a Dictionary object's key, item pair
RemoveAll	Deletes all of a Dictionary object's key, item pairs
Skip	Skips a specified number of character positions when processing a TextStream file object
SkipLine	Skips an entire line when processing a TextStream file object
Write	Places a specified string in the TextStream file object
WriteBlankLines	Writes a specified number of new-line characters to the TextStream file object
WriteLine	Writes the specified string to the TextStream file object

© Jerry Lee Ford, Jr. All Rights Reserved.

Using VBScript Run-Time Objects in Your Scripts

Now seems like a good time to look at an example of how to incorporate the VBScript FileSystemObject object into your scripts and use its properties and methods to work with the Windows file system. Take a look at the following script:

```

*****
'Script Name: FreeSpace.vbs
'Author: Jerry Ford
'Created: 01/22/14
'Description: This script demonstrates how to use VBScript run-time
'objects and their properties and methods.
*****

'Initialization Section
Option Explicit

Dim FsoObject, DiskDrive, AvailSpace

'Instantiate the VBScript FileSystemObject
Set FsoObject = WScript.CreateObject("Scripting.FileSystemObject")

'Use the FileSystemObject object's GetDrive method to set up a reference
'to the computer's C: drive
Set DiskDrive = FsoObject.GetDrive(FsoObject.GetDriveName("c:"))

'Main Processing Section

'Use the FileSystemObject FreeSpace property to determine the amount of
'free space (in MB) on the C: drive
AvailSpace = (DiskDrive.FreeSpace / 1024) / 1024

'Use the VBScript FormatNumber function to format the results as a
'whole number
AvailSpace = FormatNumber(AvailSpace, 0)

'Display the amount of free space on the C: drive
WScript.Echo "You need 100 MB of free space to play this game. " & _
vbCrLf & "Total amount of free space is currently: " & AvailSpace & " MB"

The script begins by instantiating the FileSystemObject object, as shown here:
Set FsoObject = WScript.CreateObject("Scripting.FileSystemObject")

The script then uses this instance of the FileSystemObject object to execute its GetDrive() method and
set up a reference to the computer's C: drive:
Set DiskDrive = FsoObject.GetDrive(FsoObject.GetDriveName("c:"))

```

The next statement uses the `FileSystemObject` object's `FreeSpace` property to retrieve the amount of free space on the C: drive:

```
AvailSpace = (DiskDrive.FreeSpace / 1024) / 1024
```

This statement divides this value by 1,024, and then again by 1,024, to present the amount of free space in megabytes.

The next statement formats this value further by eliminating any numbers to the right of the decimal point. The last statement displays the final result. Figure 3.6 shows how this result looks on a computer running Windows 8.1.

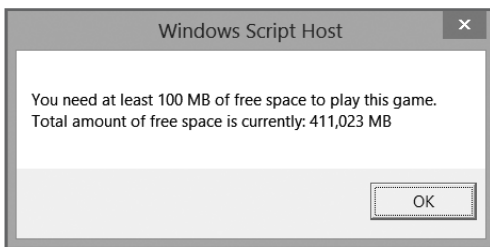


Figure 3.6 Using the `FileSystemObject` object to access information about disk drives. © 2014 Cengage Learning.

For more information on how to use the VBScript `FileSystemObject` object, see Chapter 5, “Conditional Logic,” in which I’ll show you how to create and write to Windows files to produce reports and log files. In Chapter 8, “Storing and Retrieving Data,” I’ll cover how to open and read from Windows files.

Examining Built-in VBScript Functions

One of the real advantages of working with VBScript is having access to its large number of built-in functions. In the previous example, you saw how to use the `FormatNumber()` function. There are too many built-in VBScript functions to try to list them all here. For a complete list, see Appendix D, “Built-in VBScript Functions.”

Demo: The Square-Root Calculator

By using functions, you can really streamline your scripts. VBScript’s built-in functions provide built-in code that you don’t have to write. The best way to illustrate this is to use two examples. In the first example, I’ve written a small VBScript that prompts the user to type a number so that the script can calculate its square root. The second script, discussed in the following section, is a rewrite of the first script, using the VBScript `Sqr()` function in place of the original programming logic.

Here's the first example:

```
'*****
'Script Name: SquareRoot-1.vbs
'Author: Jerry Ford
'Created: 01/22/14
'Description: This script demonstrates how to solve square-root
'calculations using a mathematic solution devised by Sir Isaac Newton
'*****
```

```
'Initialization Section
```

```
Option Explicit
```

```
Dim UserInput, Counter, X
```

```
UserInput = InputBox ("Type a number", "Square Root Calculator")
```

```
X = 1
```

```
For Counter = 1 To 15
```

```
    X = X - ((X^2 - UserInput) / (2 * X))
```

```
Next
```

```
MsgBox "The square root of " & UserInput & " is " & X
```

As you can see, the first part of the script displays a pop-up dialog box to collect the number, and the last part displays the script's final results. The middle is where the real work results:

```
X = 1
```

```
For Counter = 1 To 15
```

```
    X = X - ((X^2 - UserInput) / (2 * X))
```

```
Next
```

I won't go into the mathematical logic behind these statements. Unless you're a math major, it's a bit of a challenge to understand. This solution is based on Sir Isaac Newton's solution for solving square-root equations. Granted, it took only four lines of code to reproduce the formula, but would you like to have tried to write these four statements from scratch? I don't think so.

Demo: A New and Improved Square-Root Calculator

Now let's look at a rewrite of the square-root calculator script in which I use VBScript's built-in `Sqr()` function to perform square-root calculations.


```

'*****
'Script Name: SquareRoot-2.vbs
'Author: Jerry Ford
'Created: 01/22/14
'Description: This script demonstrates how to solve square-root
'calculations using VBScript's Built-in Sqr() function
'*****

'Initialization Section
Option Explicit

Dim UserInput
UserInput = InputBox ("Type a number", "Square Root Calculator")

MsgBox "The square root of " & UserInput & " is " & Sqr(UserInput)

```

As you can see, this time you don't have to be a mathematician to write the script. All you have to know is the correct way to use the `Sqr()` function, which is simply to pass it a number. (In the case of this script, that number is represented by a variable named `UserInput`.) These two examples show clearly the advantage of using VBScript's built-in functions. These functions can save you a lot of time and effort and perhaps a few headaches.

Figure 3.7 and Figure 3.8 demonstrate the operation of either version of these two scripts on a computer running Windows 7.

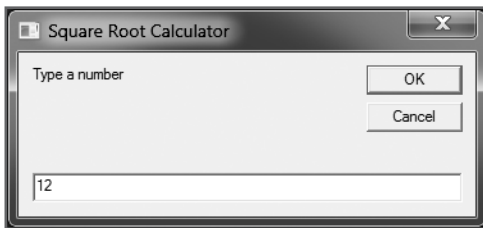


Figure 3.7 First, the script prompts the user to supply a number.

© 2014 Cengage Learning.

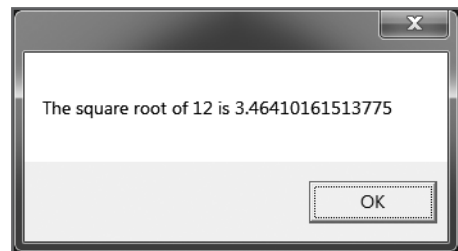


Figure 3.8 The script then determines the number's square root. © 2014 Cengage Learning.

Displaying Script Output

You've already seen many examples of how to display output messages in VBScripts. Output display is a critical tool in any programmer's toolbox. As a VBScript programmer working with the WSH, you have four different options for displaying script output.

Two of these options are provided by the WSH in the form of object methods:

- **Echo()**. This displays text messages in the Windows Console when processed by the CScript.exe execution host, and in pop-up dialog boxes when processed by the WScript.exe execution host.
- **Popup()**. This displays text messages in pop-up dialog boxes, giving you control over the icons and buttons that are displayed and, optionally, returning a value representing the button that is clicked.

In addition to these two WSH options, VBScript gives you two functions of its own:

- **InputBox()**. This displays a text-entry field in a pop-up dialog box to collect user input.
- **MsgBox()**. This displays text messages in pop-up dialog boxes, giving you control over the icons and buttons that are displayed and, optionally, returning a value representing the button that is clicked.

The WScript Object's Echo() Method

The WScript object's Echo() method can display text output in the Windows Console or in a pop-up dialog box, depending on the execution host that processes it. Table 3.7 outlines the Echo() method's behavior based on the execution host that processes it. Unlike other WSH output methods or VBScript functions, the Echo() method cannot collect user input.

TABLE 3.7 WSCRIPT ECHO() METHOD EXECUTION OPTIONS

WSH Execution Host	Output
WScript.exe	Displays text messages in graphical pop-up dialog boxes
CScript.exe	Displays text messages in the Windows Console

© Jerry Lee Ford, Jr. All Rights Reserved.

The syntax for the WScript object's Echo() method is as follows:

```
WScript.Echo [Arg1] [,Arg2]...
```

The Echo() method can display any number of arguments:

```
WScript.Echo "This message appears differently depending on the " & _  
"execution host that runs it."
```

The WshShell Object's Popup() Method

The WshShell object's Popup() method displays messages in pop-up dialog boxes. You can customize its appearance by selecting the buttons and the icon to be displayed. You can also determine which button the user clicked. To work with the WshShell object, you must first instantiate it within your script, which you can do by adding a statement like the one shown here.

```
Set WshShl = WScript.CreateObject("WScript.Shell")
```

Once instantiated, you can access `WshShell` object's methods and properties. The `WshShell` object's `Popup()` method can be used in either of two ways. The syntax of the first option is as follows:

```
Response = WScript.Popup(StrText,[Time],[TitleBarMsg],[DialogSettings])
```

The syntax of the second option is as follows:

```
WScript.Popup StrText,[Time],[TitleBarMsg],[DialogSettings]
```

`Response` is a variable that stores a number representing the button that was clicked by the user. `StrText` represents the message text to be displayed. `Time` is a value that determines how long, in seconds, the pop-up dialog box will be displayed; if omitted, the default is forever. `TitleBarMsg` is an optional message that is displayed in the pop-up dialog box's title bar. Finally, `DialogSettings` is a numeric value that specifies the buttons and the icon that are to appear on the pop-up dialog box. If omitted, the pop-up dialog box displays the OK button without an icon.

To determine what numeric value to specify as the `DialogSettings` value, see Table 3.8 and Table 3.9. Table 3.8 lists the different collections of buttons that can be displayed on pop-up dialog boxes created by the `Popup()` method, and Table 3.9 displays the different icons that you can display using the `Popup()` method.

TABLE 3.8 POPUP() METHOD BUTTON TYPES

Value	Button(s)
0	Displays the OK button
1	Displays the OK and Cancel buttons
2	Displays the Abort, Retry, and Ignore buttons
3	Displays the Yes, No, and Cancel buttons
4	Displays the Yes and No buttons
5	Displays the Retry and Cancel buttons

© Jerry Lee Ford, Jr. All Rights Reserved.

TABLE 3.9 POPUP() METHOD ICON TYPES

Value	Icon
16	Displays the stop icon
32	Displays the question mark icon
48	Displays the exclamation mark icon
64	Displays the information icon

© Jerry Lee Ford, Jr. All Rights Reserved.

For example, to display a pop-up dialog box that contains the OK button without any icon, you would specify a value of 0 for *DialogSettings*. As this is the default option for the `Popup()` method, you do not have to specify this value at all. To display a pop-up dialog box with the Yes, No, and Cancel button and no icon, you specify a value of 3 for *DialogSettings*. To display a pop-up dialog box with OK and Cancel buttons and the information icon, you specify a value of 65 (that is, the collective sum of 1 and 64).

If you use the first form of the `Popup()` method (to be able to determine which button the user clicked), you'll need to examine the value of `Response` as demonstrated here:

```
Set WshShl = WScript.CreateObject("WScript.Shell")
Response = WshShl.Popup("This is a text message", , "Test Script", 5)
If Response = 4 Then
    WshShl.Popup "You clicked on Retry"
End If
```

Table 3.10 lists the possible range of values that can be returned by the `Popup()` method.

TABLE 3.10 POPUP() METHOD RETURN VALUES

Value	Results
1	OK button
2	Cancel button
3	Abort button
4	Retry button
5	Ignore button
6	Yes button
7	No button

© Jerry Lee Ford, Jr. All Rights Reserved.

The VBScript InputBox() Function

VBScript provides two built-in functions that you can use to display text messages and interact with users. The `InputBox()` function displays your text message in a pop-up dialog box that also includes an entry field. You have already seen the `InputBox()` function in action in both the Knock Knock game and the Rock, Paper, and Scissors game.

The syntax for this function is as follows:

```
Response = InputBox(StrText[, TitleBarMsg][, default][, xpos][, ypos]
[, helpfile, context])
```

Response is a variable that stores a number representing the input typed by the user. *StrText* is the message that you want to display. *TitleBarMsg* is an optional message that will be displayed in the pop-up dialog box's title bar. *Default* is an optional default answer that you can display in the pop-up dialog box. *xpos* and *ypos* are optional arguments that specify, in twips, the horizontal and vertical location of the pop-up dialog box on the screen. *helpfile* and *context* are also optional. They specify the location of an optional context-sensitive help file.

The following statement provides another example of how to use the VBScript `InputBox()` function:

```
PlayerName = InputBox("Please type your name")
MsgBox "You typed: " & PlayerName
```

Definition

Twip stands for "twentieth of a point" and represents a value of 1/1440 inch.

The VBScript MsgBox() Function

The VBScript `MsgBox()` function displays a pop-up dialog box that is very similar to the pop-up dialog box produced by the WSH `Popup()` method. It gives you the ability to customize the appearance of the dialog box by selecting the buttons and the icon to be displayed. You also can use it to determine which button the user clicked.

The syntax for the `MsgBox()` function is as follows:

```
MsgBox(TextMsg[, buttons][, TitleBarMsg][, helpfile, context])
```

TextMsg is the message to be displayed in the dialog box. *buttons* is a representation of the buttons and icons to appear in the pop-up dialog box. *TitleBarMsg* is an optional message that will be displayed in the pop-up dialog box's title bar. *helpfile* and *context* are optional; when used, they specify the location of an optional context-sensitive help file.

Table 3.11 defines the different collections of buttons that can be appear on pop-up dialog boxes displayed using the `MsgBox()` function.

TABLE 3.11 VBSCRIPT MSGBOX() FUNCTION BUTTONS

Constant	Value	Description
<code>vbOKOnly</code>	0	Displays the OK button
<code>vbOKCancel</code>	1	Displays the OK and Cancel buttons
<code>vbAbortRetryIgnore</code>	2	Displays the Abort, Retry, and Ignore buttons
<code>vbYesNoCancel</code>	3	Displays the Yes, No, and Cancel buttons
<code>vbYesNo</code>	4	Displays the Yes and No buttons
<code>vbRetryCancel</code>	5	Displays the Retry and Cancel buttons

Table 3.12 defines the list of icons that you can add to the MsgBox() pop-up dialog box.

TABLE 3.12 VBSCRIPT MSGBOX() FUNCTION ICONS

Constant	Value	Description
vbCritical	16	Displays the critical icon
vbQuestion	32	Displays the question mark icon
vbExclamation	48	Displays the exclamation mark icon
vbInformation	64	Displays the information icon

© Jerry Lee Ford, Jr. All Rights Reserved.

You can use the MsgBox() function in your scripts like this:

```
MsgBox "Thanks for playing!"
```

You also can use the MsgBox() like this:

```
UserSelected = MsgBox("Would you like to play a game?")
```

The advantage to this last option is that you can interrogate the button that the user clicks and use it to drive the execution flow of your script like this:

```
UserSelected = MsgBox("Would you like to play a game?", 4, "Text Script")
If UserSelected = 6 Then
    MsgBox "OK, The rules of this game are as follows:!"
End If
```

Alternatively, you could rewrite the previous statements as follows:

```
UserSelected = MsgBox("Would you like to play a game?", 4, "Text Script")
If UserSelected = vbYes Then
    MsgBox "OK, let's play!"
End If
```

Table 3.13 defines the list of return values associated with the various MsgBox() buttons.

TABLE 3.13 VBSCRIPT MSGBOX() FUNCTION RETURN VALUES

Constant	Value	Description
vbOK	1	User clicked OK
vbCancel	2	User clicked Cancel
vbAbort	3	User clicked Abort
vbRetry	4	User clicked Retry
vbIgnore	5	User clicked Ignore
vbYes	6	User clicked Yes
vbNo	7	User clicked No

© Jerry Lee Ford, Jr. All Rights Reserved.

Back to the Math Game

The Math Game is played by displaying a math equation and asking the player to provide the solution. If the player provides the correct answer, the game ends; however, if the player gets the answer wrong, then the script offers to show the player how to arrive at the correct answer. This is achieved in a slide show or movie-like fashion, in which the script first starts WordPad, then starts the Calculator application, and finally uses these applications to solve the equation while the player sits back and watches. When the script is finished with its presentation, it ends by closing both the WordPad and the Calculator applications.

A Quick Overview of the WshShell Object's SendKeys() Method

Before I jump completely into the design of the Math Game, I need to give you one more piece of information. The Math Game's capability to interact with the WordPad and Calculator applications depends on the use of the `WshShell` object's `SendKeys()` method. This method is used to send keystrokes to the currently active Windows application.

Because opening another window while the `SendKeys()` method is executing will divert the keystrokes to the new window, you will want to find another way of integrating your scripts with other applications whenever possible. Many applications, such as Excel and Word, provide their own built-in core object model. WSH scripts can interact directly with these applications by first instantiating references to the application's objects and then accessing their methods and properties. The only trick here is that you need to know the objects that make up the application's object model, as well as their associated methods and properties. You can often get this information from the application vendor's website or by searching the Internet. Of course, if the application you want to work with does not expose an object model for your scripts to work with, you can always try using the `SendKeys()` method.

Trap

Because it sends keystrokes to the currently active Windows application, it is very important that, when the script is running, the player does not open any new windows (applications). If he does, the script will begin sending keystrokes to whatever applications the player opened, causing any of a number of unpredictable problems.

The syntax of the `SendKeys()` method is as follows:

```
SendKeys(string)
```

string is a value representing the keystrokes that are to be sent to the target application. You can send more keystrokes by simply typing them out, like this:

```
SendKeys "I am "  
SendKeys "38"  
SendKeys " years old."
```

However, in many cases, you'll want to send other types of keystrokes. For example, to send an Enter key keystroke, you'll need to type the following:

```
SendKeys "[td]"
```

Table 3.14 provides a list of `SendKeys()` keystrokes that you're likely to want to use.

TABLE 3.14 SENDKEYS() KEYSTROKES

Key	Corresponding SendKeys() Codes
Backspace	{BACKSPACE}, {BS}, or {BKSP}
Break	{BREAK}
Caps Lock	{CAPSLOCK}
Del or Delete	{DELETE} or {DEL}
down arrow	{DOWN}
End	{END}
Enter	{ENTER} or [td]
Esc	{ESC}
Help	{HELP}
Home	{HOME}
Ins or Insert	{INSERT} or {INS}
left arrow	{LEFT}

TABLE 3.14 SENDKEYS() KEYSTROKES (CONTINUED)

Key	Corresponding SendKeys() Codes
Num Lock	{NUMLOCK}
Page Down	{PGDN}
Page Up	{PGUP}
Print Screen	{PRTSC}
right arrow	{RIGHT}
Scroll Lock	{SCROLLLOCK}
Tab	{TAB}
up arrow	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}
F8	{F8}
F9	{F9}
F10	{F10}
F11	{F11}
F12	{F12}
F13	{F13}
F14	{F14}
F15	{F15}
F16	{F16}

© Jerry Lee Ford, Jr. All Rights Reserved.

Besides the keystrokes outlined in Table 3.14, Table 3.15 lists three additional keystroke combinations that can be used to send keystrokes that require a special key to be pressed in conjunction with another key. For example, if you were working with an application that could be closed by holding down the Alt key and pressing the F4 key, you could perform this operation as follows:

```
SendKeys "%{F4}"
```

TABLE 3.15 SPECIAL SENDKEYS() KEYSTROKES

Key	Corresponding SendKeys() Codes
Shift	+
Ctrl	^
Alt	%

© Jerry Lee Ford, Jr. All Rights Reserved.

Designing the Game

Let's start building the Math Game. This game will be assembled in five steps. The first three steps create the logic that interacts with the user and plays the game. The last two steps perform the game's application demonstrations. These steps are as follows:

1. Add the standard documentation template and define any variables, constants, and objects used by the script.
2. Present the player with the equation and then test the player's response to determine whether he provided an answer and whether that answer was numeric.
3. Test to see whether the player provided the correct answer. If not, offer to show the user how to arrive at the correct answer.
4. Add the statements required to start and control the WordPad application.
5. Add the statements required to start and control the Calculator application.

Beginning the Math Game

Let's begin by adding the script template that I introduced earlier in this chapter and then modifying it as shown here. This includes initializing variables and constants and setting up object declaration statements.

```

'*****
'Script Name: Mathgame.vbs
'Author: Jerry Ford
'Created: 01/23/14
'Description: This script prompts the user to solve a mathematical
'expression and demonstrates how to solve it in the event that the user
'cannot
'*****

'Initialization Section
Option Explicit

```

```
Dim WshShl, QuestionOne, ProveIt

'Define the title bar message to be displayed in the script's
'pop-up dialog box
Const cTitlebarMsg = "The Math Game"

'Instantiate an instance of the WshShell object
Set WshShl = WScript.CreateObject("WScript.Shell")
```

Collect the Player's Answer and Test for Errors

Next, display the equation and store the player's answer in a variable called `QuestionOne`, like this:

```
'Present the player with the equation
QuestionOne = InputBox("What is the sum of 1 + 5 * 9 / 3 ?", cTitlebarMsg)
```

Now verify that the player actually typed an answer instead of just clicking OK or Cancel. If the player did not type an answer, display an error message and end the game.

```
'See if the player provided an answer
If Len(QuestionOne) = 0 Then
    MsgBox "Sorry. You must enter a number to play this game."
    WScript.Quit
End If
```

Another good test to perform is to make sure that the player is, in fact, typing a number as opposed to a letter or other special character:

```
'Make sure that the player typed a number
If IsNumeric(QuestionOne) <> True Then
    MsgBox "Sorry. You must enter a number to play this game."
    WScript.Quit
End If
```

Check for the Correct Answer

Okay, now add a test to see if the player provided the correct answer. If the answer provided is correct, then compliment the player's math skills. Otherwise, offer to teach the player how to solve the equation.

```
'Check to see if the player provided the correct answer
If QuestionOne = 16 Then
    MsgBox "Correct! You obviously know your math!"
Else
    ProveIt = MsgBox("Incorrect. Do you want to see me solve the " & _
    "equation?", 36, cTitlebarMsg)
```

```
If ProveIt = 6 Then 'Player wants to see the solution  
  
.  
.  
.  
End If
```

```
End If
```

As you can see, I left space in the previous statements. This space marks the spot where the rest of the script's statements will be written as you continue to develop the script.

Interacting with WordPad

For the script to work with the WordPad application, WordPad must first be started. This can be done using the `WshShell` object's `Run()` method.

```
WshShl.Run "WordPad"
```

It may take a moment or two for the application to finish starting, so pause the script's execution for two seconds and wait using the `WScript` object's `Sleep()` method, like this:

```
WScript.Sleep 2000
```

Next add a series of statements that use the `SendKeys()` method to write text to WordPad. To slow things down a bit and make the process run more like a slide show, add the `Sleep()` method after each write operation. Finally, pause for a couple seconds and then close WordPad.

```
WshShl.SendKeys "To answer this question you must follow the " & _  
    "correct order of precedence when performing your calculations."  
WScript.Sleep 2000  
WshShl.SendKeys "[td][td]"  
WshShl.SendKeys "1st, working from left to right multiply 5 * 9."  
WScript.Sleep 2000  
WshShl.SendKeys "[td][td]"  
WshShl.SendKeys "2nd, divide the result by 3."  
WScript.Sleep 2000  
WshShl.SendKeys "[td][td]"  
WshShl.SendKeys "3rd, add 1."  
WScript.Sleep 2000  
WshShl.SendKeys "[td][td]"  
WshShl.SendKeys "The final answer is 16."  
WScript.Sleep 2000
```

```
WshShl.SendKeys "[td][td]"
WshShl.SendKeys "[td][td]"
WshShl.SendKeys "In case you question my math..... watch this!"
WScript.Sleep 2000
WshShl.SendKeys "%{F4}"
WshShl.SendKeys "%{N}"
```

Take notice of the last two statements. The first statement closed WordPad by sending the Alt+F4 key-stroke. As a new document was just opened, WordPad displays a dialog box asking if you want to save it. The last statement responds by sending the Alt+N keystrokes indicating a “no” response.

Interacting with the Calculator Application

The final piece of the game opens the Windows Calculator application and resolves the equation, just in case the player has any doubts as to the answer you presented using WordPad. The statements required to write this portion of the script are as follows:

```
'Start the Calculator application
WshShl.Run "Calc"

'Use the Calculator application to solve the equation
WScript.Sleep 2000
WshShl.SendKeys 5 & "{*}"
WScript.Sleep 2000
WshShl.SendKeys 9
WScript.Sleep 2000
WshShl.SendKeys "[td]"
WScript.Sleep 2000
WshShl.SendKeys "{/}" & 3
WScript.Sleep 2000
WshShl.SendKeys "[td]"
WScript.Sleep 2000
WshShl.SendKeys "{+}" & 1
WScript.Sleep 2000
WshShl.SendKeys "[td]"
WScript.Sleep 2000
WshShl.SendKeys "%{F4}"
```

As you can see, the same techniques have been used here to work with the Windows Calculator as were used to control WordPad.

The Final Result

I suggest that you run and test this script to make sure it works as expected. For example, try typing a letter instead of a number for the answer. Then try typing nothing at all and just click OK or Cancel. Finally, try both a correct and then an incorrect answer and see what happens. This is a fairly lengthy script, so the odds of typing it correctly the first time are slim. If you get errors when you run the script, read them carefully and see if the error message tells you what's wrong and then go fix it. Otherwise, you may need to double-check your typing.

Trick

As your scripts grow more complex, you're going to run into more and more errors while developing them. I recommend that you learn to develop your scripts in a modular fashion, writing one section at a time and then testing it before moving on to the next section. In Chapter 9, "Handling Script Errors," I'll demonstrate how to do this.

Summary

In this chapter, you learned about the core and run-time VBScript objects and their associated properties and methods, and were shown how to use them within your VBScripts. You also learned about VBScript syntax, reserved words, and special characters. In addition, you learned about and saw the power and convenience of VBScript functions. Finally, you learned four different ways to display script output.

Challenges

1. Change the Math Game to use a different equation and modify the logic required to adapt the statements that work with the WordPad and Calculator applications.
2. Try using the `SendKeys()` method to work with other Windows applications, such as Notepad.
3. Spend some time reviewing VBScript's built-in math functions and see if you can create a new calculator similar to the square-root calculator.
4. Modify the VBScript template presented earlier in this chapter and adapt it to suit your personal preferences. Then use it as you begin developing new VBScripts.

This page intentionally left blank

4

Constants, Variables, Arrays, and Dictionaries

This is the second of five chapters in this book that teaches the fundamentals of VBScript. One of the key concepts that you need to understand when working with VBScript, or any programming language, is how to store, retrieve, and modify data. This chapter will teach you a number of different ways to perform these tasks. By the time you have completed this chapter, you will know how to write scripts that can collect and manipulate data. Specifically, you will learn the following:

- How to process data passed to the script at execution time
- How to store data that does not change
- How to work with data that can change during script execution
- How to process collections of related data as a unit

Project Preview: The Story of Captain Adventure

In this chapter, you will learn how to create a game that builds a comical adventure story based on user input. The game begins by collecting answers to a series of questions without telling the user how the answers will be used. After all the information that the script needs is collected, the story is displayed. Figures 4.1 through 4.3 show the execution of this script on a computer running Windows 8.1.

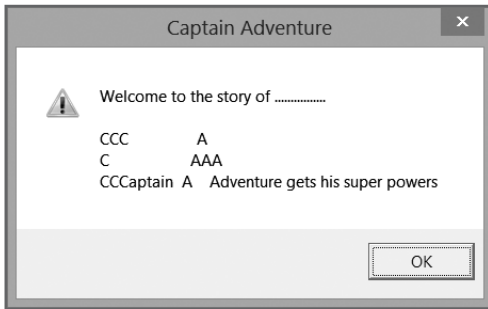


Figure 4.1 The story's initial splash screen.

© 2014 Cengage Learning.

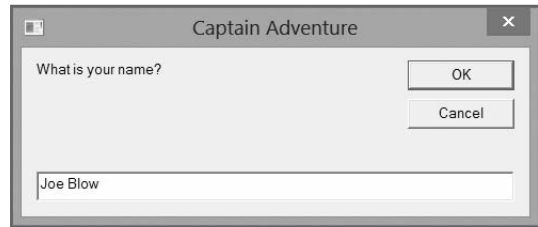


Figure 4.2 The user is the star of the story.

© 2014 Cengage Learning.

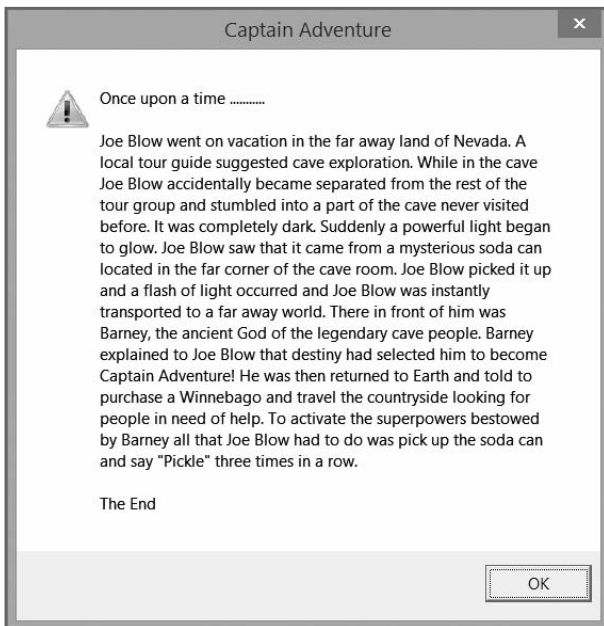


Figure 4.3 After the script has all the information it needs, the story is told. © 2014 Cengage Learning.

Through the development of this story-building game, you will learn a number of important programming techniques, including how to collect, store, and reference data. In addition, you will learn how to control the presentation of script output.

Understanding How Scripts View Data

VBScript, like other programming languages, needs a way of storing data so that it can be accessed throughout the execution of a script. In this book, you have seen a number of examples of how VBScript temporarily stores and references data. Now I'll explain how this works.

VBScript supplies a number of different statements that enable you to define several different types of data. These VBScript statements are outlined in Table 4.1.

Definition

Data is information that a computer program collects, modifies, stores, and retrieves during execution.

TABLE 4.1 VBSCRIPT STATEMENTS THAT DETERMINE HOW DATA IS DEFINED

Statement	Description
Const	Defines a VBScript constant
Dim	Defines a VBScript variable or array
ReDim	Defines a dynamic VBScript array

© Jerry Lee Ford, Jr. All Rights Reserved.

The `Const` statement is used to define data that never changes throughout the execution of a script. For example, in this book you will sometimes see constants used to define strings that define a standard greeting message in a pop-up dialog box. The `Dim` statement is used to define a variable. A *variable* stores an individual piece of data such as a name, number, or date. The `ReDim` statement is used to create an array. *Arrays* are used to store groups of related information. For example, instead of defining 20 different variables to store information about 20 different people, a single array could be defined, and then information about each person could be stored in it. Each of these statements will be examined in greater detail throughout the rest of this chapter.

Definition

A *variable* is an individual piece of data such as a name, number, or date that is stored in memory. An *array* is used to store groups of related information in memory.

Working with Data That Never Changes

Data should be defined within a script according to the manner in which it will be used. If the script only needs to reference a piece of data that has a value that is known during script development, then the data can be defined as a constant. An example of a constant is the mathematical value of π . Other examples of constants include specific dates in history, the name of places, and so on.

There are two sources of constants within scripts. First, you can define your own constants within your scripts. Another option is to reference a built-in collection of readily available constants provided by VBScript.

Definition

A *constant* is a VBScript construct that contains information that does not change during the execution of a script.

Assigning Data to Constants

If you're going to write a script and know for a fact that you need to reference one or more values that will not change during the execution of the script, then you can define each piece of data as a constant. One of the nice features of constants is that, once defined, their value cannot be changed. This prevents their values from being accidentally modified during the execution of the script.

Hint

If your script attempts to modify the value assigned to a constant after it has been initially assigned, you will see an "Illegal assignment: '*name of constant*'" error message when the script executes. Open your script, do a search for the name of the constant, and look for the statements that have attempted to modify its value to find the source of the error.

To define a constant within a VBScript, you must use the `Const` statement. This statement has the following syntax:

```
[Public | Private] Const ConstName = expression
```

`Public` and `Private` are optional keywords and are used to determine the availability of constants throughout a script. Defining a constant as `Public` makes it available to all procedures within the scripts. Defining a constant as `Private` makes it available only within the procedure that defines it. *ConstName* is the name of the constant being defined, and *expression* is the value that identifies the data being defined. To make sense of all this, let's look at an example.

Definition

A *procedure* is a collection of script statements that are processed as a unit. In Chapter 7, "Using Procedures to Organize Scripts," you will learn how to use procedures to improve the overall organization of your scripts and to create reusable units of code.

```
*****
'Script Name: LittlePigs.vbs
'Author: Jerry Ford
'Created: 01/28/14
'Description: This script demonstrates how to use a constant to create a
'standardized title bar message for pop-up dialog boxes displayed by the
'script
*****

'Specify the message to appear in each pop-up dialog box's title bar
Const cTitleBarMsg = "The Three Little Pigs"
```

```
'Display the story
MsgBox "Once upon a time.....", vbOkOnly, cTitleBarMsg
MsgBox "There were 3 little pigs", vbOkOnly, cTitleBarMsg
MsgBox "Who liked to build things.", vbOkOnly, cTitleBarMsg
```

In this example, I wrote a small VBScript that tells a very brief story about three little pigs. The script begins by defining a constant named `cTitleBar`. I then used three `MsgBox()` statements to display the text that makes up the story. The first argument in each `MsgBox()` statement is a text message, which is then followed by a VBScript `MsgBox()` constant `vbOkOnly`. This constant tells VBScript to only display the OK button on the pop-up dialog box. (A complete listing of `MsgBox()` constants is available in Chapter 3, “VBScript Basics.”) The last part of each `MsgBox()` statement is the `cTitleBarMsg` constant. VBScript automatically substitutes the value assigned to the `cTitleBarMsg` constant whenever the script executes. Figure 4.4 shows how the first pop-up dialog box appears when the script is executed on a computer running Windows 7.



Figure 4.4 By referencing the value assigned to a constant, you can create a standard title bar message for every pop-up dialog box displayed by your script.

© 2014 Cengage Learning.

Hint

I recommend you apply a naming convention for your constants. A good naming convention will make your constants easy to identify and will improve the overall readability of your scripts. For example, in this book I will use the following constant naming convention:

- Constant names begin with the lowercase letter C.
- Constant names describe their contents using English words or easily identifiable parts of words.

Other examples of tasks related to working with constants include assigning values such as numbers, strings, and dates. For example, the following statement assigns a value of 1000 to a constant called `cUpperLimit`:

```
Const cUpperLimit = 1000
```

To define a text string, you must place the value being assigned within a pair of quotes, like this:

```
Const cMyName = "Jerry Lee Ford, Jr."
```

In a similar fashion, you must use a pair of pound signs to store a date value within a constant, like this:

```
Const cMyBirthday = #11-20-64#
```

VBScript Run-Time Constants

VBScript supplies you with an abundance of built-in constants. In Chapter 3 you learned about the constants associated with the `MsgBox()` function. For example, the following VBScript statement executes the `MsgBox()` function using the `vbOkOnly` constant:

```
MsgBox "Welcome to my VBScript game!", vbOkOnly
```

This statement displays a pop-up dialog box that contains a single OK button. In addition to these constants, VBScript supplies constants that help when you're working with dates and times. VBScript also supplies a number of constants that can help you manipulate the display of text output and test the type of data stored within a variable.

Using Date and Time Constants

Table 4.2 lists VBScript date and time constants.

TABLE 4.2 VBSCRIPT DATE AND TIME CONSTANTS

Constant	Value	Description
<code>vbSunday</code>	1	Sunday
<code>vbMonday</code>	2	Monday
<code>vbTuesday</code>	3	Tuesday
<code>vbWednesday</code>	4	Wednesday
<code>vbThursday</code>	5	Thursday
<code>vbFriday</code>	6	Friday
<code>vbSaturday</code>	7	Saturday
<code>vbFirstFourDays</code>	2	First full week with a minimum of four days in the new year
<code>vbFirstFullWeek</code>	3	First full week of the year
<code>vbFirstJan1</code>	1	Week that includes January 1
<code>vbUseSystemDayOfWeek</code>	0	Day of week as specified by the operating system

© Jerry Lee Ford, Jr. All Rights Reserved.

The following script demonstrates how the `vbFriday` constant, listed in Table 4.2, can be used to determine whether the end of the workweek is here:

```
'*****
'Script Name: HappyHour.vbs
'Author: Jerry Ford
'Created: 01/28/14
'Description: This script tells the user if it's Friday
'*****
```

```
'Perform script initialization activities
Dim TodaysDate

' Weekday is a VBScript function that returns the day of the week
TodaysDate = Weekday(Date)

If TodaysDate = vbFriday then MsgBox "Hurray, it is Friday. Time " & _
    "to get ready for happy hour!"
```

Trick

You may have noticed the use of the & character in the previous example. The & character is a VBScript string concatenation operator. It allows you to combine two pieces of text into a single piece of text.

The first two lines of the script define a variable. (We'll discuss variables in detail in the next section.) The third line assigns a numeric value to the variable. In this case, the script used the VBScript Weekday() function to execute the VBScript Date() function. The Date() function retrieves the current date from the computer. The Weekday() function then provides a numeric value to represent the weekday for the date. Table 4.2 provides a list of the possible range of values in its Value column. If the current day of the week is Friday, then the value returned by the Weekday() function will be 6. Because the vbFriday constant has a value of 6, all that has to be done to determine if it is Friday is to compare the value returned by the Weekday() function to the vbFriday constant. If the two values are equal, a pop-up dialog box displays the message “Hurray, it is Friday. Time to get ready for happy hour!”

Using String Constants

Another group of constants that you may find useful is the VBScript string constants listed in Table 4.3.

TABLE 4.3 VBSCRIPT STRING CONSTANTS

Constant	Value	Description
vbCr	Chr(13)	Executes a carriage return
vbCrLf	Chr(13) and Chr(10)	Executes a carriage return and a line feed
vbFormFeed	Chr(12)	Executes a form feed
vbLf	Chr(10)	Executes a line feed
vbNewLine	Chr(13) and Chr(10)	Adds a newline character
vbNullChar	Chr(0)	Creates a 0 or null character
vbNullString	String with no value	Creates an empty string
vbTab	Chr(9)	Executes a horizontal tab
vbVerticalTab	Chr(11)	Executes a vertical tab

Using the constants shown in Table 4.3, you can control the manner in which output text is displayed. For example, take a look at the following script:

```
'*****
'Script Name: MsgFormatter.vbs
'Author: Jerry Ford
'Created: 01/28/14
'Description: This script demonstrates how to use VBScript string constants
'to control how text messages are displayed.
'*****

'Specify the message to appear in each pop-up dialog box title bar
Const cTitleBarMsg = "The three little pigs"

'Specify variables used by the script
Dim StoryMsg

'Specify the text of the message to be displayed
StoryMsg = "Once upon a time there were three little pigs" & vbCrLf & _
           "who liked to build things." & vbCrLf & vbCrLf & _
           vbCrLf & "The End"

'Display the story to the user
MsgBox StoryMsg, vbOkOnly + vbExclamation, cTitleBarMsg
```

The text message that is displayed by the script is as follows:

```
Once upon a time there were three little pigs who liked to build things. The
End
```

Notice how the `vbCrLf` and `vbTab` constants have been placed throughout the text to specify how VBScript should display the message text. Figure 4.5 shows the output that is displayed when this script is executed on a computer running Windows 8.1.

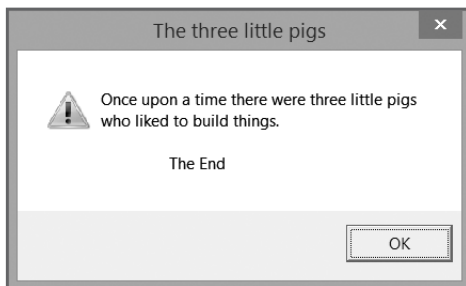


Figure 4.5 Using VBScript string constants to manipulate the display of text in pop-up dialog boxes.

If the `vbCrLf` and `vbTab` constants were removed from the formatted message, the text displayed in the pop-up dialog box would look completely different, as shown in Figure 4.6.

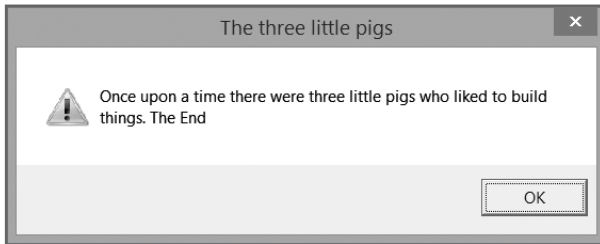


Figure 4.6 Displaying the same output as the previous example without the use of the `vbCrLf` and `vbTab` constants.

© 2014 Cengage Learning.

Storing Data That Changes During Script Execution

Chances are most programs that you write will have data in them that will need to be modified. For example, you may write a script that asks the user for input and then modifies the data while processing it. In this situation, you can define variables. Two categories of variables are available to your scripts: variables that you define within your scripts and environment variables that are maintained by Windows that your scripts can reference.

VBScript Data Types

Unlike many other programming languages, VBScript supports only one type of variable, called a *variant*. However, a variant is very flexible and can be used to store a number of different types of data. Table 4.4 lists variant data types supported by VBScript.

Definition

A *variant* is a type of variable that can contain any number of different types of data.

TABLE 4.4 VBSCRIPT SUPPORTED VARIANT SUBTYPES

Subtype	Description
Boolean	A variant with a value of <code>True</code> or <code>False</code>
Byte	An integer whose value is between 0 and 255
Currency	A currency value between -922,337,203,685,477.5808 and 922,337,203,685,477.5807
Date	A number representing a date between January 1, 100 and December 31, 9999
Double	A floating-point number with a range of -1.79769313486232E308 to -4.94065645841247E-324 or 4.94065645841247E-324 to 1.79769313486232E308
Empty	A variant that has not been initialized
Error	A VBScript error number
Integer	An integer with a value that is between -32,768 and 32,767

TABLE 4.4 VBSCRIPT SUPPORTED VARIANT SUBTYPES (CONTINUED)

Subtype	Description
Long	An integer whose value is between -2,147,483,648 and 2,147,483,647
Null	A variant set equal to a null value
Object	An object
Single	A floating-point number whose value is between -3.402823E38 and -1.401298E-45 or 1.401298E-45 and 3.402823E38
String	A string up to two billion characters long

© Jerry Lee Ford, Jr. All Rights Reserved.

Variants automatically recognize the types of data that are assigned to them and act accordingly. In other words, if a date value is assigned to a variant, then your script can use any of VBScript's built-in date functions to work with it. Likewise, if a text string is assigned to a variant, then your script can use any of VBScript's built-in string functions to work with it.

Like constants, VBScript provides you with some control over the way in which it identifies the types of values stored in a variant. For example, if you assign 100 as the variable value, then VBScript automatically interprets this value as a number. But if you enclose the value in quotation marks, VBScript treats it like a string.

VBScript also provides the capability to convert data from one type to another using built-in VBScript functions. You can use these functions within your scripts to change the way VBScript views and works with data. For example, the following VBScript statement defines a variable named `MyBirthday` and assigns it a text string of "November 20, 1964":

```
MyBirthday = "November 20, 1964"
```

VBScript views this variable's value as a text string.

However, using the `Cdate()` function, you can convert the string value to date format:

```
MyBirthday = CDate(MyBirthday)
```

Now, instead of seeing the variable's value as "November 20, 1964", VBScript sees it as 11/20/1964.

Definition

The term *string* and *text string* are used synonymously throughout this book to refer to text-based data or other types of data that have been enclosed within a pair of quotation marks.

Hint

VBScript provides a large number of conversion functions that you can use to convert from one data type to another. These functions include `Asc()`, `Cbool()`, `Cbyte()`, `Cbur()`, `Cdate()`, `Cdbl()`, `Chr()`, `Cint()`, `CLng()`, `CSng()`, and `CStr()`. For more information about how to use these functions, see Chapter 7.

Defining Variables

VBScript provides two ways of defining variables: dynamically, and formally, using the `Dim` statement. To dynamically create a variable within a script, you simply need to reference it like this:

```
MyBirthday = #November 20, 1964#
```

Using this method, you can define variables anywhere within your script. However, I strongly recommend against creating variables in this manner. It is much better to formally define any variables used in a script all at once at the beginning of the script. This helps keep things organized. In addition, I also strongly suggest that you use the `Dim` statement. The syntax of the `Dim` statement is as follows:

```
Dim VariableName
```

VariableName is the name of the variable being defined. For example, the following statement defines a variable named `MyBirthday`:

```
Dim MyBirthday
```

After a variable has been defined, you can assign a value to it, like this:

```
MyBirthday = #November 20, 1964#
```

Always use the `Dim` statement to make your code more readable and to explicitly show your intentions. You can define multiple variables within your scripts using multiple `Dim` statements:

```
Dim MyBirthday  
Dim MyName  
Dim MyAge
```

However, to reduce the number of lines of code in your scripts, you have the option of defining more than one variable at a time using a single `Dim` statement, by separating each variable with a comma and a space:

```
Dim MyBirthday, MyName, MyAge
```

As I already stated, it's better to formally define a variable before using it. One way to make sure that you follow this simple rule is to place the `Option Explicit` statement at the beginning of your scripts. The `Option Explicit` statement generates an error during script execution if you attempt to reference a variable without first defining it. Therefore, `Option Explicit` helps remind you to follow good programming practices when working with variables.

To test the use of the `Option Explicit` statement, let's take a look at another example:

```
'*****  
'Script Name: BigBadWolf.vbs  
'Author: Jerry Ford  
'Created: 01/29/14
```

```
'Description: This script demonstrates how to use the Option Explicit
'statement
'*****

'For the explicit declaration of all variables used in this script
Option Explicit

'Create a variable to store the name of the wolf
Dim WolfName

'Assign the wolf's name to the variable
WolfName = "Mr. Wiggles"

'Display the story
MsgBox "Once upon a time there was a big bad wolf named " & WolfName & _
      " who liked to eat little pigs", vbOkOnly
```

In this example, the `Option Explicit` statement is the first statement in the script. By making it the first statement in the script, `Option Explicit` will affect all variables that follow. The next statement defines a variable representing the name of the wolf. The statement after that assigns a name to the variable, which is then used by the script's final statement to tell the story. Run the script and you'll see that everything works fine.

Next, modify the script by placing a comment in front of the `Dim` statement and run the script again. This time, instead of executing properly, an error will appear, indicating that the script attempted to reference an undefined variable. Figure 4.7 shows the error message that is displayed when the script is executed on Windows 7.

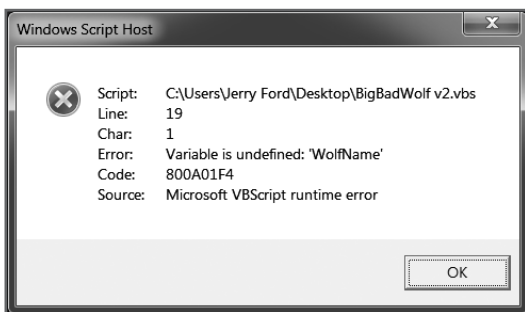


Figure 4.7 Use the `Option Explicit` statement in all your scripts to prevent the use of undefined variables. © 2014 Microsoft Corporation. Used with permission from Microsoft.

To summarize, the following list provides the guidelines that you should follow when working with variables:

- Define your variables at the beginning of your scripts in one location.
- Use the `Option Explicit` statement to enforce formal variable declaration.
- Use the `Dim` statement to define each variable.

Up to this point in the chapter, I have not been following these rules because I had not gotten to them yet. However, from this point on, you'll find them applied consistently in every script that uses variables.

Trick

There is one exception to the set of rules that I want to point out. In certain cases, you can limit the availability of a variable and its value to a specific portion of your scripts. This is called “creating a local variable” and can be useful when manipulating a sensitive variable to make sure that its value is not accidentally modified by other parts of the script. I'll provide more information about local variables a little later in this chapter.

Variable Naming Rules

Another important issue that merits attention is the proper naming of variables. VBScript has a number of rules that you must follow to avoid errors from inappropriately named variables. These rules include the following:

- Variable names must be fewer than 256 characters long.
- Variable names must begin with an alphabetic character.
- Only letters, numbers, and the underscore (`_`) character can be used when creating variable names.
- Reserved words cannot be used as variable names.
- Variable names cannot contain spaces.
- Variable names must be unique.

One more important thing to know about VBScript variables is that they are case insensitive. That means capitalization does not affect VBScript's capability to recognize a variable. Therefore, if a script defines a variable as `MyBirthday` and then later references it as `MYBIRTHDAY`, VBScript will recognize both spellings of the variable name as the same. However, mixing cases in this manner can be confusing to anyone looking at your code, so you should do your best to use a consistent case throughout your scripts.

You should assign descriptive variable names in your scripts and select a standard approach in the way that you use case in the spelling of your variables. Up to this point in the book, I have defined variables using complete words or parts of words, and I have begun each word or part of a word with a capital letter, as in `Dim MyBirthday`.

Another approach that many programmers take when naming variables is to add a three-character prefix, sometimes referred to as “Hungarian Notation,” to the beginning of each variable name. For example, instead of naming a variable `MyBirthday`, you would name it `dtmMyBirthday`. This way, when someone examines the first three characters of the variable name, they’ll be able to tell that it contains a date. The following list identifies typical prefixes associated with each of the variables subtypes supported by VBScript.

- Boolean: `bln`
- Byte: `byt`
- Currency: `cur`
- Date: `dtm`
- Double: `dbl`
- Error: `err`
- Integer: `int`
- Long: `lng`
- Object: `obj`
- Single: `sng`
- String: `str`
- Variant: `var`

Now that I have formally introduced you to Hungarian Notation, you’ll see it throughout the rest of this book.

Variable Scope

Another key concept that you need to understand when working with variables is *variable scope*. In this context, *scope* refers to the ability to reference a variable and its assigned value from various locations within a script.

VBScript supports two different variable scopes: global and local. A variable with a global scope can be accessed from any location within the script. However, a variable with a local scope can only be referenced from within the procedure that defines it. VBScript supports two types of procedures, subroutines and functions, both of which are discussed in detail in Chapter 7.

As you know, a procedure is a collection or group of statements that is executed as a unit. Without getting too far ahead of myself, for now just note that a variable defined outside a procedure is global in scope, meaning that it can be accessed from any location within the script, including from within the script’s procedures. A variable that is local in scope is defined within a procedure.

Modifying Variable Values with Expressions

Throughout this chapter, you have seen the equals sign (=) used to assign value to variables, like this:

```
strMyName = "Jerry Lee Ford, Jr."
```

To change the value assigned to a variable, all you have to do is use the equals sign again, along with a new value, like this:

```
strMyName = "Jerry L. Ford, Jr."
```

The two previous examples set and then modified the value assigned to a text variable. However, this same approach works just as well for other types of variables, such as those that contain numeric values:

```
Option Explicit
Dim intMyAge
intMyAge = 37
intMyAge = 38
```

In this example, the variable is defined and then assigned a numeric value. The value assigned to that variable is then modified to a different number. VBScript provides additional ways of modifying the value of numeric variables using the equals sign and VBScript arithmetic operators. Table 4.5 lists the VBScript arithmetic operators

TABLE 4.5 VBSCRIPT ARITHMETIC OPERATORS

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
-	Negation
^	Exponentiation
\	Integer division
Mod	Modulus

© Jerry Lee Ford, Jr. All Rights Reserved.

The best way to explain how these arithmetic operators are used is to show you an example. Take a look at the following script:

```
'*****
'Script Name: MathDemo.vbs
'Author: Jerry Ford
'Created: 01/29/14
'Description: This script demonstrates how to use various VBScript
'arithmetic operators
'*****

'Force the explicit declaration of all variables used in this script
Option Explicit
```

```
'Create a variable to store the name of the wolf
Dim intMyAge

'Assign my initial starting age
intMyAge = 37
WScript.Echo "I am " & intMyAge

'Next year I will be
intMyAge = intMyAge + 1
WScript.Echo "Next year I will be " & intMyAge

'But I am not that old yet
intMyAge = intMyAge - 1
WScript.Echo "But I still am " & intMyAge

'This is how old I'd be if I were twice as old as I am today
intMyAge = intMyAge * 2
WScript.Echo "This is twice my age " & intMyAge

'And if I took that value, divided it by 5, added 3, and multiplied it
'by 10
intMyAge = intMyAge / 5 + 3 * 10
WScript.Echo "This says that I will be " & intMyAge
```

If you run this script on a computer running Windows 8.1, you'll see the results shown in Figure 4.8.

The first four calculations should be fairly easy to understand. In the first calculation, I took the value of `intMyAge` and added 1 to it. Similarly, I subtracted 1 in the next calculation and then multiplied `intMyAge` by 2 in the third statement. The final calculation requires a little more explanation. You may have been surprised by this calculation's result. At first glance, it appears that VBScript will try to solve the equation as follows:

1. Divide `intMyAge` (which is 74) by 5 to get 14.8.
2. Add 14.8 to 3, getting 17.8.
3. Multiply 17.8 by 10, getting as the final result 178.

However, VBScript says that the answer is actually 44.8. How could this be? The answer lies in something called the "order of precedence," which tells VBScript the order in which to perform individual calculations within an equation or expression. Table 4.6 outlines the order in which VBScript order of precedence occurs.

```

C:\Scripts>CScript mathdemo.vbs
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

I am 37
Next year I will be 38
But I still am 37
This is twice my age 74
This says that I will be 44.8

C:\Scripts>

```

Figure 4.8 Using VBScript arithmetic operators to modify numeric variables.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

**TABLE 4.6 ORDER OF PRECEDENCE FOR
VBSCRIPT ARITHMETIC OPERATORS**

Operator	Description	Operator	Description
^	Exponentiation	\	Integer division
-	Negation	Mod	Modulus
*	Multiplication	+	Addition
/	Division	-	Subtraction

Note: Operators appearing at the beginning of the table have precedence over operators appearing later in the table. © Jerry Lee Ford, Jr. All Rights Reserved.

Exponentiation occurs before negation, negation occurs before multiplication, and so on. So when applied to the last calculation in the previous example, VBScript solves the equation as follows:

1. Multiply 3 by 10 to get 30.
2. Divide intMyAge (which is 74) by 5 to get 14.8.
3. Add 14.8 to 30 to get the final result of 44.8.

You can add parentheses to your VBScript expressions to exercise control over the order in which individual calculations are performed. For example, you could rewrite the last expression in the previous example as follows:

```
intMyAge = ((intMyAge / 5) + 3) * 10
```

VBScript now performs individual calculations located within parentheses first, like this:

1. Divide `intMyAge` (which is 74) by 5 to get 14.8.
2. Add 14.8 to 3 to get 17.8.
3. Multiply 17.8 by 10 to get a final result of 178.

Using the WSH to Work with Environment Variables

Thus far, the scripts that you have worked with in this chapter have used variables that are defined by the scripts themselves. A second type of variable, known as an *environment variable*, is also available to your VBScripts. Windows operating systems automatically create and maintain environment variables.

The two types of environment variables are as follows:

- **User.** User environment variables are created during user login and provide information specific to the currently logged on user.
- **System.** System environment variables are created based on what Windows learns about the computer. They are available at all times, as opposed to user variables, which are available only when you're logged in to the computer.

Hint

You may end up writing scripts that are designed to run when no one is logged on to the computer. This can be done using the Windows Scheduler Service to automate the execution scripts. In this case, user environment variables will not be available to your scripts. It will be up to you to make sure that your scripts do not depend on them.

User and system environment variables can be viewed from the Environment Variables dialog box. On Windows 7, you can access this dialog box using the following procedure:

1. Click the Start button.
2. Right-click Computer and select Properties. A window showing basic system information is displayed.
3. Click the Advanced system settings link. The System Properties window appears.
4. Click the Advanced tab and then click the Environment Variables button. A listing of environment variables is displayed.

You can use the following procedure to access system environment variables on Windows 8.1:

1. Move the mouse pointer to the bottom-right side of the screen and click the Search icon when it is displayed.
2. Type Control Panel in the Apps Search field and press Enter.
3. In the Control Panel, click the System and Security link, and then click the System link.
4. Click Advanced System Settings to display the System Properties window.
5. Click the Environment Variables button to display the Environment Variables window, shown in Figure 4.9.

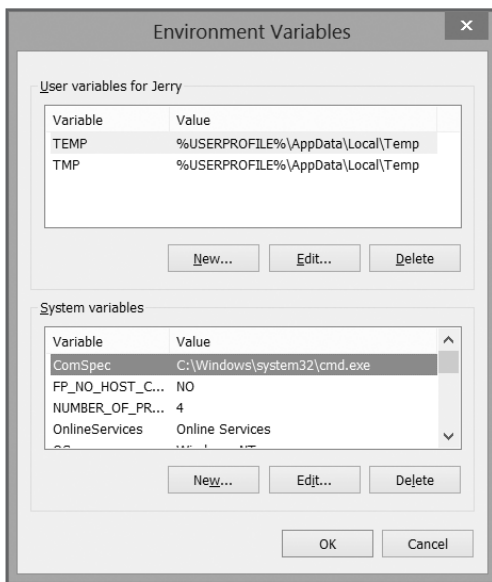


Figure 4.9 Examining Windows environment variables.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

Examples of user environment variables include the following:

- **TEMP.** A folder where temporary files can be stored
- **TMP.** Another folder where temporary files can be stored

Examples of system environment variables include the following:

- **NUMBER_OF_PROCESSORS.** Displays a value of 1 for single-processor computers
- **OS.** Displays the operating system's name
- **Path.** Specifies the current search path
- **PATHEXT.** Specifies a list of extensions that identify executable files

- **PROCESSOR_ARCHITECTURE.** Identifies the computer's processor type
- **PROCESSOR_IDENTIFIER.** Displays a detailed description of the computer's processor
- **PROCESSOR_LEVEL.** Displays the processor's stepping level
- **PROCESSOR_REVISION.** Displays the processor's revision number
- **TEMP.** A folder in which temporary files can be stored
- **TMP.** Another folder in which temporary files can be stored
- **Windir.** Identifies the location of the Windows folder

To access environment variables, you need to use the WSH. For example, take a look at the following script:

```
*****
'Script Name: ComputerAnalyzer.vbs
'Author: Jerry Ford
'Created: 01/29/14
'Description: This script demonstrates how to access environment
'variables using the WSH
*****

'Force the explicit declaration of all variables used in this script
Option Explicit

'Create a variable to store the name of the wolf
Dim objWshObject

'Set up an instance of the WScript.WshShell object
Set objWshObject = WScript.CreateObject("WScript.Shell")

'Use the WScript.Shell object's ExpandEnvironmentStrings() method to view
'environment variables
MsgBox "This computer is running a version of " & _
    objWshObject.ExpandEnvironmentStrings("%OS%") & vbCrLf & _
    "and has " & _
    objWshObject.ExpandEnvironmentStrings("%NUMBER_OF_PROCESSORS%") & _
    " processor(s)."
```

The first statement in this example creates an instance of the WSH `WScript.Shell` object using the `WScript` object's `CreateObject()` method. The next part of the script uses the `WScript.Shell` object's `ExpandEnvironmentStrings()` method to display the value of specific environment variables.

Although the script demonstrates how to access environment variables, it really isn't very useful. Another use for environment variables might be to validate the operating system on which the script has been started and to terminate script execution if it has been started on the wrong operating system like this:

```
Set objWshObject = WScript.CreateObject("WScript.Shell")
If objWshObject.ExpandEnvironmentStrings("%OS%") <> "Windows_NT" Then
    MsgBox "This script is designed to only run on " & _
        "Windows 2000, XP, 2003, Vista, 2008, 2012, 7 or 8."
    WScript.Quit
End If
```

In this example, the first line of code checks to see if the script is being run on Windows NT, 2000, XP, 2003, Vista, 2008, 2012, 7, or 8. If it isn't, then the second and third lines of code execute, informing the user of the situation and terminating the script's execution.

Working with Collections of Related Data

When using variables, you can store an incredible amount of information during the execution of your scripts. Indeed, you're limited only by the amount of memory available on your computer. However, keeping up with large numbers of variables can be difficult and may make your scripts difficult to maintain.

Often, data items processed by a script have a relationship to one another. For example, if you write a script that collects a list of names from the user to generate a list of personal contacts, it would be more convenient to store and manage the list of names as a unit instead of as a collection of individual names. VBScript provides support for arrays so that such a task can be performed.

For example, you can think of an array as being like a collection of numbered index cards, where each card contains the name of a person who has been sent an invitation to a party. If you assigned a name of `ItsMyParty` to the collection of cards, you could then programmatically refer to any card in the collection as `ItsMyParty(index#)`, where `index#` is the number written on the card.

The `ItsMyParty` collection is an example of a single-dimension array. VBScript is capable of supporting arrays with as many as 60 dimensions. In most cases, all you'll need to work with are single-dimension arrays, so that's where I'll focus most of my attention.

Definition

An *array* is an indexed list of related data. The first element, or piece of data, stored in the array is assigned an index position of 0. The second element is assigned an index position of 1, and so on. Thus, by referring to an element's index position, you can access its value.

Single-Dimension Arrays

To create a single-dimension array, you use the `Dim` statement. When used in the creation of arrays, the `Dim` statement has to have the following syntax:

```
Dim ArrayName(dimensions)
```

dimensions is a comma-separated list of numbers that specifies the number of dimensions that make up the array. For example, the following VBScript statement can be used to create a single-dimension array named `ItsMyParty` that can hold up to 10 names:

```
Dim ItsMyParty(9)
```

Notice that I used the number 9 to define an array that can hold up to 10 elements. This is because the first index number in the array is automatically set to 0, thus allowing the array to store 10 elements (that is, 0–9).

After an array is defined, it can be populated with data.

Hint

Because the first element stored in an array has an index of 0, its actual length is equal to the number supplied when the array is first defined plus one.

The following VBScript statements demonstrate how you can assign data to each element in the array:

```
ItsMyParty(0) = "Jerry"  
ItsMyParty(1) = "Molly"  
ItsMyParty(2) = "William"  
ItsMyParty(3) = "Alexander"  
.  
.  
.  
ItsMyParty(9) = "Mary"
```

As you can see, to assign a name to an element in the array, I had to specify the element's index number.

After you populate the array, you can access any array element by specifying its index number, like this:

```
MsgBox ItsMyParty(1)
```

In this example, `ItsMyParty(1)` equates to `Molly`.

Hint

If you like the suggestions I made earlier in this chapter about naming constants and about using Hungarian Notation when creating names for your variables, then you might want to combine these two approaches when naming your arrays. For example, instead of naming an array `ItsMyParty()`, you might want to name it `astrItsMyParty()`. The first character of the name identifies that it's an array and the next three characters identify the type of data that's stored in the array. This is the naming standard that I'll use when naming arrays throughout the rest of this book.

Multiple-Dimension Arrays

As stated, VBScript can support arrays with as many as 60 dimensions, although one or two dimensions are usually sufficient. Let's take a look at how to define a two-dimensional array, which you can think of as being like a two-column table. The first column contains the names of the guests invited to the party and the second column stores the guests' phone numbers.

```
Dim astrItsMyParty(3,1)

astrItsMyParty(0,0) = "Jerry"
astrItsMyParty(0,1) = "550-9933"
astrItsMyParty(1,0) = "Molly"
astrItsMyParty(1,1) = "550-8876"
astrItsMyParty(2,0) = "William"
astrItsMyParty(2,1) = "697-4443"
astrItsMyParty(3,0) = "Alexander"
astrItsMyParty(3,1) = "696-4344"
```

In this example, a two-dimensional array is created that is four rows deep and two columns wide, allowing it to store up to eight pieces of data. To refer to any particular element in this two-dimensional array, you must supply both its row and column coordinates, like this:

```
WScript.Echo astrItsMyParty(1,0)
WScript.Echo astrItsMyParty(1,1)
```

The first of the two previous statements refers to the `Molly` element. The second statement refers to the phone number associated with `Molly`.

Hint

Another way of thinking about a two-dimensional array is to consider it a one-dimensional array made up of a collection of one-dimensional arrays.

Processing Array Contents

So far, in all the examples, I have accessed each array element by specifically referencing its index position. This works fine as long as the array is small, but it's not a practical approach for processing the contents of large arrays, which may contain hundreds or thousands of entries. To handle arrays of this size, a different approach is needed. VBScript's solution to this issue is the `For...Each...Next` loop. The syntax of the `For...Each...Next` loop is as follows:

```
For Each element In group
    Statements . . .
Next [element]
```

element is a variable that the loop uses to iterate through each array element. *group* identifies the name of the array. *Statements* are the VBScript statements that you add to process the contents of each array element. The For...Each...Next loop continues processing until every element in the array has been examined.

The For...Each...Next loop lets your scripts process the entire contents of an array using just a few statements. The number of statements required does not increase based on array size. Therefore, you can use a For...Each...Next loop to process extremely large arrays with very little programming effort. For example, the next script defines an array named GameArray() and populates it with five elements. It then processes the entire array using just one line of code located within a For...Each...Next loop.

```
*****
'Script Name: ArrayDemo.vbs
'Author: Jerry Ford
'Created: 01/30/14
'Description: This script demonstrates how to store and retrieve data
'using a single-dimension VBScript array.
*****

'Perform script initialization activities
Option Explicit

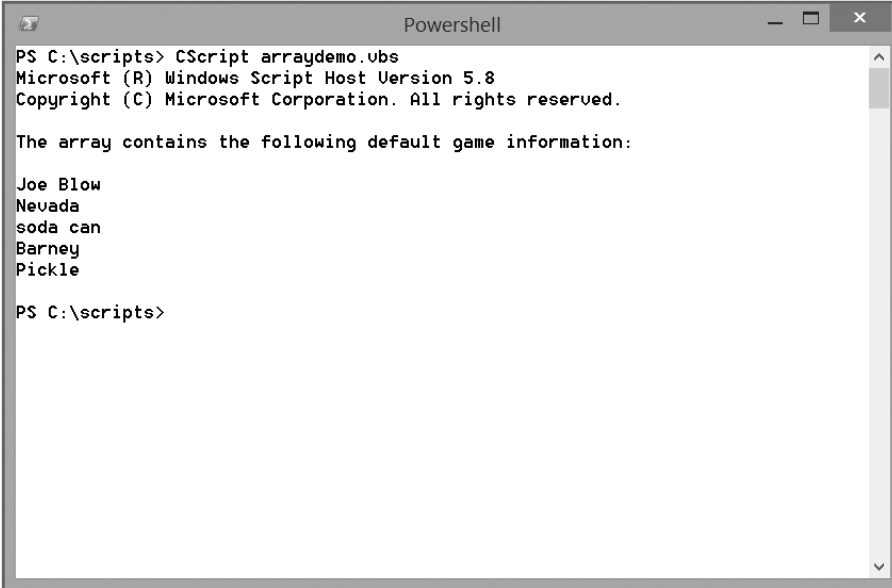
'Define variables used in the script
Dim intCounter    'Variable used to control a For...Each loop
Dim strMessage    'Message to be displayed in a pop-up dialog box

strMessage = "The array contains the following default game " & _
    "information: " & vbCrLf & vbCrLf

Dim astrGameArray(4) 'Define an array that can hold 5 index elements
astrGameArray(0) = "Joe Blow" 'The default username
astrGameArray(1) = "Nevada"    'A place worth visiting
astrGameArray(2) = "soda can"  'An interesting object
astrGameArray(3) = "Barney"    'A close friend
astrGameArray(4) = "Pickle"    'A favorite dessert

For Each intCounter In astrGameArray
    strMessage = strMessage & intCounter & vbCrLf
Next
WScript.Echo strMessage
```

If you run this script using the CScript.exe execution host on a computer running Windows 8.1, you receive the output shown in Figure 4.10.

A screenshot of a PowerShell window titled "Powershell". The window shows the execution of a script named "arraydemo.ubs" in the directory "C:\scripts". The output of the script is as follows:

```
PS C:\scripts> CScript arraydemo.ubs
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

The array contains the following default game information:

Joe Blow
Nevada
soda can
Barney
Pickle

PS C:\scripts>
```

Figure 4.10 Iteratively processing all the contents of an array.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

These same few lines of code can easily process the array, even if it has a hundred or a thousand elements. The alternative to the `For...Each...Next` loop is to write an individual statement to access each element stored within the array, like this:

```
'Display the contents of the array
WScript.Echo strMessage
WScript.Echo astrGameArray(0)
WScript.Echo astrGameArray(1)
WScript.Echo astrGameArray(2)
WScript.Echo astrGameArray(3)
WScript.Echo astrGameArray(4)
```

Writing statements for individual access array contents may not seem like a lot of work in a small program, but imagine trying to process an array with 1,000 elements!

Xref

For more information on how to work with the `For...Each...Next` loop, see Chapter 6, “Processing Collections of Data.”

Getting a Handle on the Size of Your Arrays

VBScript provides you with two built-in functions that make it easier to work with arrays:

- **Ubound()**. This returns a numeric value indicating the array's upper bound or its highest element.
- **Lbound()**. This returns a numeric value indicating the array's lower bound or its lowest element.

The `Lbound()` function isn't really that useful because the lower bound of all VBScript arrays is always equal to 0. On the other hand, the `Ubound()` function can be quite handy, especially when working with dynamic arrays. I'll cover dynamic arrays a little later in this chapter. The syntax of the `Ubound()` function is as follows:

```
Ubound(ArrayName, Dimension)
```

ArrayName is the name of the array whose upper bound is to be returned. *Dimension* is used to specify the array dimension whose upper bound is to be returned. For example, you could retrieve the upper bound of a single-dimension array called `astrItsMyParty` as shown here:

```
intSize = Ubound(astrItsMyParty)
```

In this case, the upper bound of the array is assigned to a variable named `intSize`. Let's look at a quick example of the `Ubound()` function in action:

```
Dim intCounter 'Define a variable to be used when processing array
                'contents
Dim strMessage 'Define a variable to be used to store display output

Dim astrGameArray(2) 'Define an array that can hold three index elements
astrGameArray(0) = "Joe Blow" 'The default username
astrGameArray(1) = "Nevada" 'A place worth visiting
astrGameArray(2) = "soda can" 'An interesting object

intSize = UBound(astrgameArray)

For intCounter = 0 to intSize
    strMessage = strMessage & astrGameArray(intCounter) & vbCrLf
Next

MsgBox strMessage
```

Run this example and you'll see that the script displays all three elements in the array in a pop-up dialog box.

Resizing Arrays

Sometimes it's impossible to know how many elements an array will need to store when developing your scripts. For example, you might develop a script that uses the `InputBox()` function to prompt the user to specify the data to be stored in the array. You might expect the user to specify only a few pieces of data, but the user may have an entirely different idea. To handle this type of situation, you need a way of resizing the array to allow it to store the additional data.

One way of dealing with this situation is to define the array without specifying its size, like this:

```
Dim astrGameArray()
```

This lets you define the array's size later in the script. For example, you might want to define the array and later ask the user how many pieces of data he intends to provide. This is accomplished by using the `ReDim` statement:

```
ReDim astrGameArray(9)
```

This `ReDim` statement has set up the array to store up to 10 elements. After its size has finally been defined, you can begin populating the array with data.

Trap

If you use the `ReDim` statement to set up a new array by accidentally specifying the name of an existing array, the data stored in the existing array will be lost.

Another, more flexible way of setting up an array so that it can be later resized is to replace the array's original `Dim` definition statement with the `ReDim` statement. For example, the following statement sets up a new array capable of holding up to 10 elements:

```
ReDim astrTestArray(9)
```

However, if you populate this array with data and then later attempt to resize it, as shown next, you'll lose all the data originally stored in the array.

```
ReDim astrTestArray(19)
```

To prevent this from happening, you can add the `Preserve` keyword to the `ReDim` statement, like this:

```
ReDim Preserve astrTestArray(19)
```

This statement instructs VBScript to expand the size of the array while preserving its current contents. After the array is expanded, you can then add more elements to it. For example, take a look at the next script. It defines an array with the capability to store five elements and then resizes the array to increase its storage capacity to eight elements. The script then uses a `For...Each...Next` loop to display the contents of the expanded array.

```

'*****
'Script Name: ResizeArray.vbs
'Author: Jerry Ford
'Created: 01/30/14
'Description: This script demonstrates how to resize an array during
'execution
'*****

'Perform script initialization activities
Option Explicit

'Define variables used in the script
Dim intCounter      'Variable used to control a For...Each loop
Dim strMessage     'Message to be displayed in a pop-up dialog box

strMessage = "The array contains the following default game " & _
    "information: " & vbCrLf & vbCrLf

ReDim astrGameArray(4) 'Define an array that can hold five index elements
astrGameArray(0) = "Joe Blow" 'The default username
astrGameArray(1) = "Nevada"   'A place worth visiting
astrGameArray(2) = "soda can" 'An interesting object
astrGameArray(3) = "Barney"   'A close friend
astrGameArray(4) = "Pickle"   'A favorite dessert

ReDim Preserve astrGameArray(7) 'Change the array to hold eight entries
astrGameArray(5) = "Lard Tart" 'Default villain name
astrGameArray(6) = "water gun" 'Default villain weapon
astrGameArray(7) = "Earth"     'Planet the villain wants to conquer

'Display the contents of the array
For Each intCounter In astrGameArray
    strMessage = strMessage & intCounter & vbCrLf
Next
WScript.Echo strMessage

```

Trap

Be careful not to accidentally lose any data if you decide to resize an array to a smaller size. For example, if you defined an array that can hold 100 elements and then later resize it to hold 50 elements using the Preserve keyword, only the first 50 elements in the array will actually be preserved.

Building Dynamic Arrays

Up to this point, all the arrays shown in this book have been static, meaning that their size was predetermined at execution time. But in the real world, you won't always know how many elements your arrays will need to store. For example, you might write a script that enables the user to supply a list of names of people to be invited to a party. Depending on the number of friends the user has, the amount of data to be stored in the array can vary significantly. VBScript's solution to this type of situation is dynamic arrays. A *dynamic array* is an array that can be resized during execution as many times as necessary.

You can use the `Dim` statement to define a dynamic array as shown here:

```
Dim astrItsMyParty()
```

Note that an index number for the array was not supplied inside the parentheses. This allows you to come back later on in the script and resize the array using the `ReDim` statement as demonstrated here:

```
ReDim astrItsMyParty(2)
```

Once resized, you can add new entries:

```
astritsmyParty(0) = "Molly"  
astrItsMyParty(1) = "William"  
astrItsMyParty(2) = "Alexander"
```

If the script later needs to add more elements to the array, you can resize it again:

```
ReDim Preserve astrItsMyParty(4)
```

This statement has increased the size of the array so that it can now hold an additional two elements. Note the use of the `Preserve` keyword on the `ReDim` statement. This parameter was required to prevent the array from losing any data stored in it before the array's size was increased.

Trap

Dynamic arrays can be increased or decreased in size. If you decrease the size of a dynamic array, all elements stored in the array are lost even if the `Preserve` keyword is added to the `ReDim` statement.

Now let's look at one more example of how to work with dynamic arrays. In this example, an array named `astrItsMyParty` is initially set up with the capability to store one element. The user is then prompted to provide a list of names to be added to the array. Each time a new name is supplied, the script dynamically increases the size of the array by 1, allowing it to hold additional information.

```
Dim astrItsMyParty()  
ReDim astrItsMyParty(0)  
Dim intCounter, strListOfNames
```

Definition

A *dynamic array* is an indexed list of related data stored in memory that can be resized during execution.

```
intCounter = 0

Do While UCase(strListOfNames) <> "QUIT"
    strListOfNames = InputBox("Enter the name of someone to be invited: ")

    If UCase(strListOfNames) <> "QUIT" Then
        astrItsMyParty(intCounter) = strListOfNames
    Else
        Exit Do
    End If

    intCounter = intCounter + 1
    ReDim Preserve astrItsMyParty(intCounter)
Loop
```

In this example, the array is named `astrItsMyParty`. A `Dim` statement is used to define it and then a `ReDim` statement is used to set its initial size, thus allowing it to store a single element. After setting up a couple variables used by the script, I added a `Do...While` loop to collect user input. The loop runs until the user types `Quit`.

Note

`UCase()` is a VBScript function that converts string characters to uppercase. Using `UCase()`, you can develop scripts that process user input regardless of how the user employs capitalization when entering data.

Assuming that the user does not type `Quit`, the script adds the names entered by the user to a string, which is stored in a variable named `strListOfNames`. Otherwise, the `Do...While` loop terminates. Each time a new name is entered, the `ReDim` statement is executed to redimension the array by increasing its size by one.

Erasing Arrays

When your script is finished working with the data stored in an array, you can erase or delete it, thus freeing up a small portion of the computer's memory for other work. This is accomplished using the `Erase` statement, which has the following syntax:

```
Erase ArrayName
```

For example, the following statement could be used to erase an array named `astrGameArray`:

```
Erase astrGameArray
```

Storing Data in Dictionaries

In addition to variables and arrays, VBScript also lets you store and manage data using Dictionary objects. With dictionaries, data is stored in an associative array and retrieved using key/value pairs (as opposed to using the data index position, as is the case with arrays). Data is stored in a dictionary using keys. Keys can be numbers, strings, or any other type of data supported by VBScript. Dictionaries provide the same benefits as arrays while also providing greater flexibility in the way data is stored and retrieved. In addition, Dictionary objects provide you with access to a number of different properties and methods that provide access and manage data.

The Dictionary object supports three properties, as listed here.

- **Count.** This returns the total number of items in a Dictionary object.
- **Item.** This retrieves or adds an item using a specified key in a Dictionary object.
- **Key.** This adds a key to a Dictionary object.

The Dictionary object also provides access to a number of methods:

- **Add.** This adds a key/item pair to a Dictionary object.
- **Exists.** This returns a Boolean value of true if a specified key exists in the Dictionary object and false if the value does not exist.
- **Items.** This returns an array of all items from a Dictionary object.
- **Keys.** This returns an array of keys from a Dictionary object.
- **Remove.** This deletes a key/item pair from a Dictionary object.
- **Remove All.** This removes all key/item pairs from a Dictionary object.

Keys and Values

To work the Dictionary object into a VBScript, you must first define a variable through which it can be referenced. Once this is done, you can create an instance of the object as demonstrated here:

```
Dim dictObject
Set dictObject = CreateObject("Scripting.Dictionary")
```

Adding Dictionary Items

After you have instantiated a Dictionary object, you can begin storing items in it as key/value pairs, as demonstrated here:

```
dictObject.Add "Model555", "Red Bike"
dictObject.Add "Model888", "Blue Bike"
dictObject.Item("Model999") = "Green Bike"
```

The first statement uses the Dictionary object's `Add()` method to add a key named `Model555` to the `dictObject` dictionary. A value of `Red Bike` is then assigned to this key. The second statement adds a second key named `Model888` to the dictionary, assigning it a value of `Blue Bike`. The last statement does things a little differently, using the `Item` property to make a key/value assignment.

Retrieving Dictionary Items

After you have created and populated a dictionary, you can begin working with the data stored in it. As an example of how to do so, look at the following statements:

```
If dictObject.Exists("Model555") = True Then
    MsgBox "Model555 is a " & dictObject.Item("Model555")
End If
```

In this example, the Dictionary object's `Exists()` method is used to ascertain whether the specified key exists. If it exists, the `Item()` method is used to retrieve the value associated with the key.

Deleting Dictionary Items

If necessary, you can delete key/value pairs from a dictionary just as easily as you added them. To do so, you use the Dictionary object's `Remove()` method. For example, the following statement shows how to use the `Remove()` method to delete a data element using its associated key:

```
DictObject.Remove "Model555"
```

As you can see, this statement removes the value associated with the `Model555` key from the dictionary. Alternatively, you can use the Dictionary object's `RemoveAll()` method to delete all of the key/value pairs stored in a dictionary, as demonstrated here:

```
DictObject.RemoveAll
```

Processing Data Passed to a Script at Run-Time

Up to this point, every script you have seen in this chapter expects to have its data hard-coded as constants, variables, and arrays. Another way for a script to access data for processing is to set it up so that the user can pass it arguments for processing at execution time.

Passing Arguments to Scripts

To pass arguments to a script, you must start the script from the command line, as follows:

```
CScript DisplayArgs.vbs tic tac toe
```

Definition

An *argument* is a piece of data passed to the script at the beginning of its execution. For example, a script that is designed to copy a file from one location to another might accept the name of the file to be copied as an argument.

In the previous statement, the CScript.exe execution host is used to start a VBScript named DisplayArgs.vbs. Three arguments have been passed to the script for processing. Each argument is separated by a blank space.

The next example shows a slight variation of the statement. In this case, the script still passes three arguments, but because the second argument contains a blank space, it must be enclosed in quotation marks:

```
CScript DisplayArgs.vbs tic "super tac" toe
```

Of course, for a script to accept and process arguments at execution time, it must be set up to do so, as demonstrated in the next section.

Designing Scripts That Accept Argument Input

To set a script up to accept arguments, as demonstrated in the previous section, you can use the WSH's WshArguments object as shown in the following script:

```
*****
'Script Name: ArgumentProcessor.vbs
'Author: Jerry Ford
'Created: 01/30/14
'Description: This script demonstrates how to work with arguments passed
'to the script by the user at execution time
*****

'For the explicit declaration of all variables used in this script
Option Explicit

'Define variables used during script execution
Dim objWshArgs, strFirstArg, strSecondArg, strThirdArg

'Set up an instance of the WshArguments object
Set objWshArgs = WScript.Arguments

'Use the WshArguments object's Count property to verify that three arguments
'were received. If three arguments are not received then display an error
'message and terminate script execution.
If objWshArgs.Count <> 3 then
    WScript.Echo "Error: Invalid number of arguments."
    WScript.Quit
End IF
```



```
'Assign each argument to a variable for processing
strFirstArg = objWshArgs.Item(0)
strSecondArg = objWshArgs.Item(1)
strThirdArg = objWshArgs.Item(2)

'Display the value assigned to each variable
WScript.Echo "The first argument is " & strFirstArg & vbCrLf & _
            "The second argument is " & strSecondArg & vbCrLf & _
            "The third argument is " & strThirdArg & vbCrLf
```

To use the `WshArguments` object, the script must first create an instance of it, like this:

```
Set objWshArgs = WScript.Arguments
```

Next, the script uses the `WshArguments` object's `Count` property to make sure that three arguments have been passed to the script. If more than or fewer than three arguments have been received, an error message is displayed and the script terminates its execution. Otherwise, the script continues and assigns each of the arguments to a variable. Each argument is stored in an indexed list by the `WshArguments` object and is referenced using the object's `Item()` method. `Item(0)` refers to the first arguments passed to the script. `Item(1)` refers to the second argument, and `Item(2)` refers to the third argument.

Finally, the `WScript.Echo` method is used to display each of the arguments passed to the script. The following shows how the script's output appears when executed using the `CScript.exe` execution host and three arguments (`tic`, `tac`, and `toe`):

```
C:\>CScript.exe ArgumentProcessor.vbs tic tac toe
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
The first argument is tic
The second argument is tac
The third argument is toe
```

```
C:\>
```

Similarly, the following output shows what happens when only two arguments are passed to the script:

```
C:\>CScript.exe ArgumentProcessor.vbs tic tac
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Error: Invalid number of arguments.
```

```
C:\>
```

Back to the Story of Captain Adventure

Now let's return to the chapter's programming project, the Story of Captain Adventure. In this programming project, you'll develop a script that displays a story describing how the story's hero, Captain Adventure, first gets his superpowers. Through the development of this script, you'll have the opportunity to put your knowledge of how to work with VBScript constants, variables, and string formatting constants to the test.

Designing the Game

The basic design of this project is to ask the user a bunch of questions (without telling the player what the answers will be used for) and then to use the information provided by the player to build a comical action story about a fictional hero named Captain Adventure.

This project will be completed in five steps.

1. Add the standard documentation template and fill in its information.
2. Define the constants and variables that will be used by the script.
3. Create the splash screen that welcomes the user to the story.
4. Use the `InputBox()` function to create variables that store user-supplied data.
5. Write the story, adding the data stored in the script's variables and constants. In addition, use VBScript string constants to control the manner in which the story text is formatted before finally displaying the story using the `MsgBox()` function.

Beginning the Captain Adventure Script

The first step in putting this project together, now that an outline of the steps involved has been defined, is to open your editor and set up your script template as follows:

```

'*****
'Script Name: Captain Adventure.vbs
'Author: Jerry Ford
'Created: 02/01/14
'Description: This script prompts the user to answer a number of questions
'and then uses the answers to create a comical action adventure story.
'*****

'Perform script initialization activities
Option Explicit

```

This template, introduced in the last chapter, gives you a place to provide some basic documentation about the script that you're developing. In addition, the template includes the `Option Explicit` statement, based on the assumption that just about any script that you'll develop will use at least one variable.

Setting Up Constants and Variables

The next step in creating the Captain Adventure script is to specify the constants and variables that will be used by the script:

```
'Specify the message to appear in each pop-up dialog box title bar  
Const cGameTitle = "Captain Adventure"
```

```
'Specify variables used by the script  
Dim strWelcomeMsg, strName, strVacation, strObject, strFriend  
Dim strFood, strStory
```

The first line of code defines a constant named `cGameTitle`. This constant will be used to define a message that will be displayed in the title bar area of any dialog boxes displayed by the script. This allows you to define the title bar message just once, and to apply it as needed throughout the script without having to retype it each time.

The last line of code defines seven variables that the script will use. The first variable, `strWelcomeMsg`, will store the message text that will be displayed in a splash screen that appears when the script first executes.

In the Real World

Sometimes splash screens are used to remind the user to register the application. In other instances, splash screens are meant to distract the user when applications take a long time to load or may be used to advertise the website of the application or script developer. Adding a splash screen to your script gives you the chance to communicate with the user before the script begins its execution; it can be used to display instructions or other useful information.

The next five variables (`strName`, `strVacation`, `strObject`, `strFriend`, and `strFood`) are used to store data collected from the user; they will be used later in the script in assembling the Captain Adventure story. The last variable, `strStory`, is used to store the fully assembled Captain Adventure story.

Creating a Splash Screen

As I said, adding a splash screen to your script gives you an opportunity to display your website address, game instructions, or other information you think will be useful to the user.

The following statements show one way of building a splash screen. The `strWelcomeMsg` variable is used to define the text that will be displayed in the splash screen. The message text to be displayed is formatted using VBScript string constants to make it more attractive.

```
'Specify the message to be displayed in the initial splash screen  
strWelcomeMsg = "Welcome to the story of ....." & vbCrLf & _
```

```
vbCrLf & "CCC" & space(14) & "A" & vbCrLf & _
"C" & space(17) & "AAA" & vbCrLf & _
"CCCaptain A Adventure gets his super powers" & _
vbCrLf
```

```
' Welcome the user to the story
MsgBox strWelcomeMsg, vbOkOnly + vbExclamation, cGameTitle
```

Finally, the VBScript `MsgBox()` function is used to display the splash screen. In this case, the `vbOkOnly + vbExclamation` constants in the `MsgBox()` function instruct VBScript to display only the OK button and the exclamation mark graphic on the pop-up dialog box. In addition, the `cGameTitle` constant has been added to display the script's custom title bar message.

Collecting User Input

The next five lines of code, shown next, use the VBScript `InputBox()` function to collect data provided by the user. This code contains the following five questions/instructions:

- What is your name?
- Name a place you would like to visit.
- Name a strange object.
- Type the name of a close friend.
- Type the name of your favorite dessert.

```
'Collect story information from the user
strName = InputBox("What is your name?", cGameTitle,"Joe Blow")
strVacation = InputBox("Name a place you would like to visit.", _
    cGameTitle,"Nevada")
strObject = InputBox("Name a strange object.", cGameTitle,"soda can")
strFriend = InputBox("Type the name of a close friend.", _
    cGameTitle,"Barney")
strFood = InputBox("Type the name of your favorite dessert.", _
    cGameTitle,"Pickle")
```

Notice that the user is only given a little bit of information about the type of information the script is looking for. This is intentional and is meant to provide a certain amount of unpredictability to the story line.

You may have also noticed the final argument on each of the `InputBox()` statements. I have added the argument so that each dialog box that is displayed by the script will automatically display a default answer. Providing a default answer in this way helps the user by giving him an idea of the kind of information you're trying to collect.

Assembling and Displaying the Story

The last step in putting together the Captain Adventure script is to assemble the story. This is done by typing out the story's text while inserting references to the script's variables at the appropriate locations in the story. In addition, the `vbCrLf` string constant is used to improve the display of the story.

The entire story is assembled as a single string, which is stored in a variable called `Story`. Finally, the completed story is displayed using the VBScript `MsgBox()` function.

```
' Assemble the Captain Adventure story
strStory = "Once upon a time ....." & vbCrLf & vbCrLf & _
  strName & " went on vacation in the far away land of " & strVacation & _
  ". A local tour guide suggested cave exploration. While in the cave " & _
  strName & " accidentally became separated from the rest of the tour " & _
  "group and stumbled into a part of the cave never visited before. " & _
  "It was completely dark. Suddenly a powerful light began to glow. " & _
  strName & " saw that it came from a mysterious " & strObject & " " & _
  "located in the far corner of the cave room. " & strName & " picked " & _
  "it up and a flash of light occurred and " & strName & " was " & _
  "instantly transported to a far away world. There in front of him " & _
  "was " & strFriend & ", the ancient God of the legendary cave " & _
  "people. " & strFriend & " explained to " & strName & " that " & _
  "destiny had selected him to become Captain Adventure! He was " & _
  "then returned to Earth and told to purchase a Winnebago and travel " & _
  "the countryside looking for people in need of help. To activate " & _
  "the superpowers bestowed by " & strFriend & " all that " & strName & _
  " had to do was pick up the " & strObject & " and say " & Chr(34) & _
  strFood & Chr(34) & " three times in a row." & vbCrLf & vbCrLf & _
  "The End"
```

```
'Display the story
MsgBox strStory, vbOkOnly + vbExclamation, cGameTitle
```

Note the use of `Chr(34)` in the preceding statements. `Chr(34)` is a VBScript function that converts an ANSI code to a character. An ANSI code of 34 represents a double quotation mark.

The Final Result

Run the script and test it to make sure that everything works as expected. Be aware that this script pushes the string length allowed by VBScript to the limit. If the information you supply to the script is too long, some of the story may end up truncated.

Summary

You covered a lot of ground in this chapter. You now know how to define and work with constants and variables, including VBScript's built-in constants and Windows environment variables. In addition, you learned how to apply VBScript string constants to script output to control the manner in which output is displayed. You also learned about the VBScript variant and how to use built-in VBScript functions to convert data from one variant subtype to another. Finally, you learned how to store related collections of data in arrays for more efficient storage and processing, and to develop scripts that can process input passed to them at execution time.

Challenges

1. Modify the Captain Adventure story by collecting additional user input and adding more text to the story line.
2. Try using an array to store the user input collected in the Captain Adventure story instead of storing data in individual variables.
3. Develop your own story for someone you know and email it to him or her as a sort of living greeting card.
4. Experiment with the VBScript string constants when developing your own story to improve the format and presentation of your story's output.

This page intentionally left blank

5

Conditional Logic

Every programming language allows you to perform tests between two or more conditions. This capability is one of the cornerstones of programming logic. It lets you develop scripts that collect input from the user or the user's computer and compare it to one or more conditions. Using the results of the tests, you can alter the execution of your scripts and create dynamic scripts that can adjust their execution according to the data with which they're presented.

By the time you have completed this chapter, you'll have learned the following:

- How to write scripts that test two conditions
- How to write scripts that can test two or more conditions against a single value
- How to write scripts that can test for a variety of different types of conditions
- How to write scripts that work with a variety of built-in VBScript functions

Project Preview: The Planet Trivia Quiz Game

In this chapter, you'll create a game that administers and scores a quiz based on the player's knowledge of the planets in the solar system. The game asks the player a series of questions and then assigns a rank based on the player's final score. Figures 5.1 through 5.3 show some of the interaction between the player and the game when played on a computer running Windows 8.1.

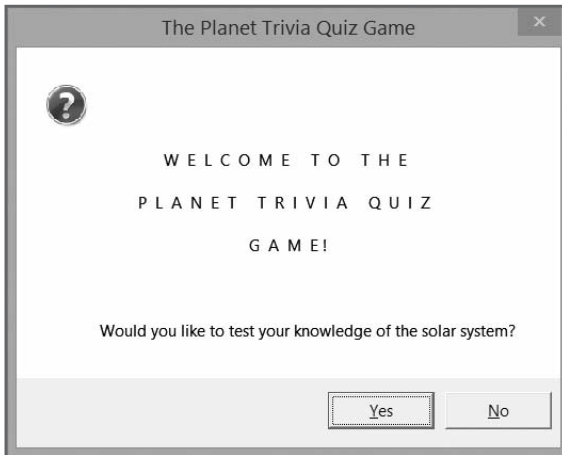


Figure 5.1 The game's splash screen invites the user to take the quiz.

© 2014 Cengage Learning.

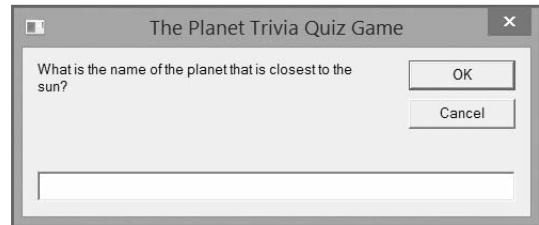


Figure 5.2 The player is presented with a series of questions to answer. © 2014 Cengage Learning.

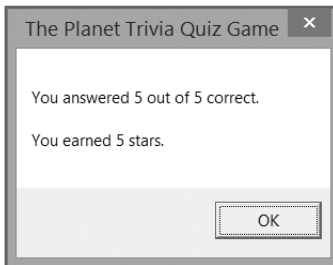


Figure 5.3 When the player finishes the questions, his score is tallied, and a rank is assigned based on the number of questions correctly answered. © 2014 Cengage Learning.

During the development of this game, you will learn how to apply sophisticated conditional logic. In addition, you'll learn how to work with a number of built-in VBScript functions.

Examining Program Data

In any programming language, you need to be able to test whether a condition is true or false to develop complex logical processes. VBScript provides two different statements that perform this function. These statements are as follows:

- **If.** This is a statement that allows or skips the execution of a portion of a program based on the results of a logical expression or condition.
- **Select Case.** This is a formal programming construct that allows a programmer to visually organize program flow when dealing with the results of a single expression.

You already saw short demonstrations of the `If` statement in earlier chapters of this book. This is because even the simplest scripts require some form of conditional logic.

The power and importance of these two statements cannot be overstated. For example, let's say you took a job from someone without knowing exactly what you'd be paid, and now you're finished with the job and are waiting to be paid. As you're waiting, you think about what you want to do with your newfound fortune. After a few moments, you decide that if you're paid \$250, then you'll buy a TV. If you're paid less, you'll buy a radio instead. This kind of test lends itself well to an If statement. Let's rewrite this scenario into a more program-like format:

```
If your pay is equal to $250
    Then Buy a TV
Else
    Buy a Radio
EndIf
```

As you can see, the logic is very straightforward and translates well from English into pseudo code. I used bold text to identify portions of the example to point out the key VBScript programming components that are involved. I'll go into greater detail about what each of these keywords means a little later in the chapter.

Back to our scenario: Perhaps after thinking about it a few more minutes, you decide there are a number of things that you might do with your pay, depending on how much money you receive. In this case, you can use the VBScript `Select Case` statement to outline the logic of your decisions in pseudo code format.

```
Select Case Your Pay
    Case If you get $250 you'll buy a TV
    Case If you get $200 you'll buy a VCR
    Case If you get $100 you'll buy a radio
    Case Else You'll just buy lunch
End Select
```

Again, I have used bold text to identify portions of the example to point out key VBScript programming components involved. In the next two sections of this chapter, I'll break down the components of the `If` and `Select Case` statements into greater detail and show you exactly how they work.

Definition

Pseudo code is a rough, English-like outline or sketch of a script. By writing out the steps you think will be required to write a script in pseudo code, you provide yourself with an initial first-level script design that will serve as a basis for building the final product.

The If Statement

The VBScript `If` statement lets you test two values or conditions and alter the execution of the script based on the results of the test. The syntax of this statement is as follows:

```
If condition Then
    statements
Elseif condition-n Then
    statements
```

```
.  
. .  
. .  
Else  
    statements  
End If
```

Working with the If Statement

The If statement begins with the If keyword and ends with End If. *condition* represents the comparison being performed. For example, you might want to see whether the value of *X* is equal to 250, like this:

```
If X = 250 Then
```

The keyword Then identifies the beginning of a list of one or more statements. *statements* is a placeholder representing the location where you would supply whatever script statements you want executed. For example, the following example displays a complete If statement that tests to see whether a variable has a value of 250. If it does (that is, the test is equal to true), a message is displayed:

```
If X = 250 Then  
    WScript.Echo "Go buy that TV!"  
End If
```

You may add as many statements as you want between the Then and End If keywords.

```
If X = 250 Then  
    WScript.Echo "Go buy that TV!"  
    WScript.Echo "Buy a TV Guide while you are at it."  
    WScript.Echo "And do not forget to say thank you."  
End If
```

But what happens if the tested condition proves false? Well, in the previous test, nothing. However, by adding the Else keyword and one or more additional statements, the script is provided with an additional execution path.

```
If X = 250 Then  
    WScript.Echo "Go buy that TV!"  
    WScript.Echo "Buy a TV Guide while you are at it."  
    WScript.Echo "And do not forget to say thank you."  
Else  
    WScript.Echo "OK. Just purchase the radio for today."  
End If
```

Figure 5.4 provides a flowchart view of the logic used in this example.

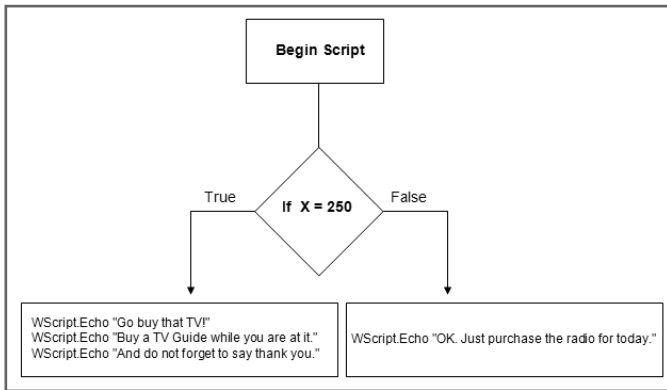


Figure 5.4 A flowchart outlining the logic behind a typical If statement. © 2014 Cengage Learning.

In the Real World

Programmers sometimes begin script development by first creating a flowchart. The flowchart depicts the logical flow of a script or program and serves as a visual tool for script development and provides a valuable documentation tool. Flowchart development can be a big help in the creation of complex scripts. Flowcharts help programmers formalize their thoughts before script development begins.

You can expand the If statement by adding one or more ElseIf keywords, each of which can test another alternative condition. For example, look at the following VBScript statements:

```

If X = 250 Then
    WScript.Echo "Go buy that TV!"
    WScript.Echo "Buy a TV Guide while you are at it."
    WScript.Echo "And do not forget to say thank you."
ElseIf X = 200 Then
    WScript.Echo "Buy the VCR"
ElseIf X = 100 Then
    WScript.Echo "Buy the Radio."
Else
    WScript.Echo "OK. Maybe you had best just eat lunch."
End If
  
```

Nesting If Statements

Another way to use If statements is to embed them within each other. This enables you to develop scripts that test for a condition and then further test other conditions based on the result of the previous test.

To see what I mean, look at the following example (I have bolded the embedded If statement to make it easier to see):

```
X = 250
If X = 250 Then
    If Weekday(date()) = 1 Then
        WScript.Echo "It's Sunday. The TV store is closed on Sundays."
    Else
        WScript.Echo "Go buy that TV!" & vbCrLf & _
            "Buy a TV Guide while you are at it." & vbCrLf & _
            "And do not forget to say thank you."
    End If
Else
    WScript.Echo "OK. Just purchase the radio for today."
End If
```

In this example, the first statement performs a test to see whether the value assigned to a variable named `X` is equal to 250. If it's not equal to 250, the script skips all the statements located between the `If X = 250 Then` line and the `Else` line and displays the message "OK. Just purchase the radio for today." However, if the value of `X` is equal to 250, then the embedded If statement executes. This If statement begins by determining whether the current day of the week is Sunday. If it is, the script informs the user that the TV store is closed. Otherwise, it tells the user to go and make the purchase.

The test performed by the If statement in the previous example deserves a little extra explanation. As you saw, it retrieved a numeric value representing the current day of the week. Here's how to break down the logic used by this statement. First, it executed the built-in VBScript `Date()` function. The value retrieved by this function was then used by the built-in VBScript `Weekday()` function to determinate the numeric value that represents the current day of the week. These values are as follows:

- 1 = Sunday
- 2 = Monday
- 3 = Tuesday
- 4 = Wednesday
- 5 = Thursday
- 6 = Friday
- 7 = Saturday

When this value was established, the If statement simply checked to see if it was equal to 1 (Sunday).

By taking advantage of built-in VBScript functions, you can perform some fairly complex tasks with minimal coding. It's a good idea to always check to see whether VBScript has a built-in function before attempting to write a piece of code to perform a generic task.

Hint

By *embedding*, or nesting one If statement within another If statement, you can develop complex programming logic. There's no limit on the number of If statements you can embed within one another.

RockPaperScissors.vbs Revisited

Okay. You've learned a lot about the If statement, including its syntax and various ways in which it can be used. One of the biggest challenges I faced in coming up with the VBScript examples for the first four chapters of this book was how to create VBScript-based games without using VBScript programming statements that I had not yet covered. For the most part I was successful, but there was one exception: I just could not avoid using the If statement—although I tried to use it as little as possible. In most cases, this meant limiting the completeness of the games presented.

One such game was the one that used the RockPaperScissors.vbs script. Now that I've finally provided a complete review of the If statement, let's revisit the game and see how we can make it better.

```
'Formally declare each variable used by the script before trying to  
'use them
```

```
Dim objWshShell, strAnswer, strCardImage, intGetRandomNumber
```

```
'Create an instance of the WScript object in order to later use  
'the Popup method
```

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
```

```
'Display the rules of the game
```

```
objWshShell.Popup "Welcome to Rock, Paper, and Scissors game. " & _  
    "Here are the " & _  
    "rules of the game: 1. Guess the same thing as the computer " & _  
    "to tie. 2. Paper covers rock and wins. 3. Rock breaks " & _  
    "scissors and wins. 4. Scissors cut paper and win."
```

```
'Prompt the user to select a choice
```

```
strAnswer = InputBox("Type Paper, Rock, or Scissors.", _  
    "Let's play a game!")
```

```
'Time for the computer to randomly pick a choice
```

```
Randomize
```

```
intGetRandomNumber = Round(FormatNumber(Int((3 * Rnd) + 1)))
```

```
'Assign a value to the randomly selected number
```

```
If intGetRandomNumber = 3 then strCardImage = "rock"
```

```
If intGetRandomNumber = 2 then strCardImage = "scissors"
If intGetRandomNumber = 1 then strCardImage = "paper"
```

```
'Display the game's results so that the user can see if he won
objWshShell.Popup "You picked: " & strAnswer & Space(12) & _
"Computer picked: " & strCardImage
```

Figure 5.5 shows the output of a complete game as the script currently is written when executed on a computer running Windows 7.

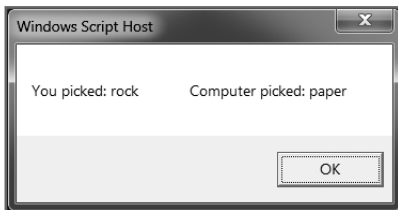


Figure 5.5 Playing Rock, Paper, and Scissors.

© 2014 Cengage Learning.

First, let's update the script by adding the script template that was introduced back in Chapter 3, "VBScript Basics."

```
'*****
'Script Name: RockPaperScissors-2.vbs
'Author: Jerry Ford
'Created: 02/04/14
'Description: This script revisits the RockPaperScissors.vbs script, first
'introduced in Chapter 3, and updates it using advanced conditional logic.
'*****
```

Next, let's rewrite the Dim statement by adding another variable called Results:

```
Dim objWshShell, strAnswer, strCardImage, strResults
```

strResults is used later in the scripts to store the results of the game (that is, who wins and who loses). Next let's add the following statement to the script:

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
```

This statement creates an instance of the objWshShell object. This object's Quit() method is used later in the script to terminate its execution in the event the user fails to provide a valid selection (that is, the player does not pick rock, paper, or scissors).

Now that the variables and objects to be used by the script have been defined, let's assign a default value of None to the strResults variable, like this:

```
strResults = "None"
```

Unless the player provides a correct selection, this value will remain equal to `None` throughout the script's execution and will eventually cause the script to terminate and display an error message. However, if the player supplies a correct response, the response will be assigned to the `strResults` variable and then analyzed by the script.

The original `RockPaperScissors.vbs` script displayed the game's instructions in one pop-up dialog box and then prompted the player to specify a selection of rock, paper, or scissors in a second pop-up dialog box. This works, but using two pop-up dialog boxes is a bit clunky. Let's modify the scripts to display the game's directions and collect the player's input at the same time, like this:

```
strAnswer = InputBox("Please type paper, rock, or scissors " & _  
    "in all lowercase letters." & _  
    vbCrLf & vbCrLf & "Rules:" & vbCrLf & vbCrLf & _  
    "1. Guess the same thing as the computer to tie." & vbCrLf & _  
    "2. Paper covers rock and wins." & vbCrLf & _  
    "3. Rock breaks scissors and wins." & vbCrLf & _  
    "4. Scissors cut paper and win." & vbCrLf, "Let's play a game!")
```

As you can see, I used the VBScript `InputBox()` function to display the pop-up dialog box, and I formatted the instructions for better presentation using the `vbCrLf` constant.

The next two sections of the script remain the same as in the original script:

```
Randomize  
intGetRandomNumber = Round(FormatNumber(Int((3 * Rnd) + 1)))  
If intGetRandomNumber = 3 then strCardImage = "rock"  
If intGetRandomNumber = 2 then strCardImage = "scissors"  
If intGetRandomNumber = 1 then strCardImage = "paper"
```

As explained in Chapter 2, "An Introduction to the Windows Script Host," the first pair of statements results in the selection of a random number with a value between 1 and 3. The next three lines assign a value of "rock," "paper," or "scissors" to each of these values. The rest of the script will be composed of all new code. Instead of simply displaying the player's and the script's selection of rock, paper, or scissors and then leaving it up to the player to figure out who won, the script now performs the analysis. To begin, add the following lines to the bottom of the script:

```
If strAnswer = "rock" Then  
    If intGetRandomNumber = 3 Then strResults = "Tie"  
    If intGetRandomNumber = 2 Then strResults = "You Win"  
    If intGetRandomNumber = 1 Then strResults = "You Lose"  
End If
```

This set of statements executes only if the player typed `rock`. Three `If` statements then compare the user's selection to the script's randomly selected decisions and determine the results of the game.

Now replicate this collection of statements two times, and modify each set as follows to add tests for the selection of both scissors and paper:

```
If strAnswer = "scissors" Then
    If intGetRandomNumber = 3 Then strResults = "You Lose"
    If intGetRandomNumber = 2 Then strResults = "Tie"
    If intGetRandomNumber = 1 Then strResults = "You Win"
End If
```

```
If strAnswer = "paper" Then
    If intGetRandomNumber = 3 Then strResults = "You Win"
    If intGetRandomNumber = 2 Then strResults = "You Lose"
    If intGetRandomNumber = 1 Then strResults = "Tie"
End If
```

Now add the following statements to the script:

```
If strResults = "None" Then
    objWshShell.Popup "Sorry. Your answer was not recognized. " & _
        "Please type rock, paper, or scissors in all lowercase letters."
    WScript.Quit
End If
```

These statements execute only if the player fails to provide a correct response when playing the game. If this happens, the value of `strResults` is never changed and will still be set to `None` as assigned at the beginning of the script. In this case, the `objWshShell` object's `Popup()` and `Quit()` methods are used to display an error message and then end the game.

Now let's wrap up this script by adding these last few lines of code:

```
objWshShell.Popup "You picked: " & space(12) & strAnswer & vbCrLf & _
    vbCrLf & "Computer picked: " & space(2) & strCardImage & vbCrLf & _
    vbCrLf & "======" & vbCrLf & vbCrLf & "Results: " & _
    strResults
```

These statements are executed only if the player provided a valid response. They use the `objWshShell` object's `Popup()` method to display the results of the game, including both the player's and the script's selections.

Okay. That's all there is to it. Save and execute the script. Figure 5.6 shows the initial pop-up dialog box displayed by the script when executed on a computer running Windows 7.

Figure 5.7 shows the results of a typical game.

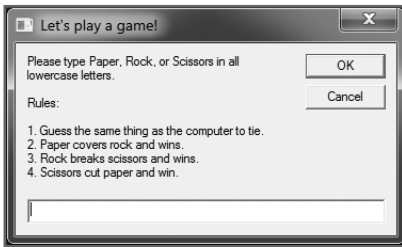


Figure 5.6 The new version of RockPaperScissors.vbs displays a friendlier initial dialog box. © 2014 Cengage Learning.

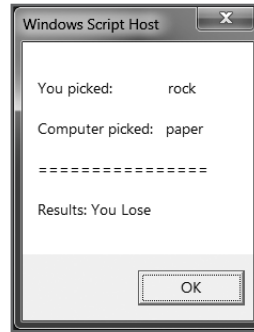


Figure 5.7 The results of a typical game of the new version of RockPaperScissors.vbs. © 2014 Cengage Learning.

The Select Case Statement

The If statement provides a great tool for testing two expressions. Using ElseIf, you can modify the If statement to perform additional tests. VBScript supplies another statement, Select Case, that also lets you perform comparative operations. Functionally, it's not really very different from the If statement. However, the Select Case statement is better equipped to perform large numbers of tests against a single expression.

Here is the syntax of the Select Case statement:

```
Select Case expression
  Case value
    statements
    .
    .
    .
  Case value
    statements
  Case Else
    statements
End Select
```

The Select Case statement begins with Select Case. Then it specifies the expression to be compared against one or more values specified in Case statements that follow the Select Case statement and precede the End Select statement. Optionally, a Case Else statement can be added to provide an alternative course of action should none of the Case statements' values match up against the expression specified by the Select Case statement.

To demonstrate how to work with the Select Case statement, I have rewritten most of the logic implemented in the RockPaperScissors.vbs script. As the following complete script shows, not only did I reduce the number of lines of code required for the script to work, but I also improved the script's readability:

```

*****
'Script Name: RockPaperScissors-3.vbs
'Author: Jerry Ford
'Created: 02/04/14
'Description: This script revisits the RockPaperScissors-2.vbs script,
'replacing some of the If statements' logic with a Case Select statement.
*****

'Perform script initialization activities
Dim objWshShell, strAnswer, strCardImage, strResults, intGetRandomNumber
Set objWshShell = WScript.CreateObject("WScript.Shell")

strResults = "None"

'Prompt the user to select a choice
strAnswer = InputBox("Please type paper, rock, or scissors " & _
    "in all lowercase letters." & _
    vbCrLf & vbCrLf & "Rules:" & vbCrLf & vbCrLf & _
    "1. Guess the same thing as the computer to tie." & vbCrLf & _
    "2. Paper covers rock and wins." & vbCrLf & _
    "3. Rock breaks scissors and wins." & vbCrLf & _
    "4. Scissors cut paper and win." & vbCrLf, "Let's play a game!"))

'Time for the computer to randomly pick a choice
Randomize
intGetRandomNumber = Round(FormatNumber(Int((3 * Rnd) + 1)))
If intGetRandomNumber = 3 then strCardImage = "rock"
If intGetRandomNumber = 2 then strCardImage = "scissors"
If intGetRandomNumber = 1 then strCardImage = "paper"

Select Case strAnswer
Case "rock"
    If intGetRandomNumber = 3 Then strResults = "Tie"
    If intGetRandomNumber = 2 Then strResults = "You Win"
    If intGetRandomNumber = 1 Then strResults = "You Lose"
Case "scissors"
    If intGetRandomNumber = 3 Then strResults = "You Lose"
    If intGetRandomNumber = 2 Then strResults = "Tie"
    If intGetRandomNumber = 1 Then strResults = "You Win"

```

```

Case "paper"
  If intGetRandomNumber = 3 Then strResults = "You Win"
  If intGetRandomNumber = 2 Then strResults = "You Lose"
  If intGetRandomNumber = 1 Then strResults = "Tie"
Case Else
  objWshShell.Popup "Sorry. Your answer was not recognized. " & _
    "Please type rock, paper, or scissors in all lowercase letters."
  WScript.Quit
End Select

objWshShell.Popup "You picked: " & space(12) & strAnswer & vbCrLf & _
  vbCrLf & "Computer picked: " & space(2) & strCardImage & vbCrLf & _
  vbCrLf & "======" & vbCrLf & vbCrLf & "Results: " & _
  strResults

```

Performing More Complex Tests with VBScript Operators

Up to this point in the book, every example of an `If` or a `Select Case` statement that you have seen has involved a single type of comparison: equality. This is a powerful form of comparison, but there will be times when your scripts will need to test for a wider range of values. For example, suppose you want to write a script that asks the user to type his age so you can determine whether the user is old enough to play your game. (Maybe you don't want a user to play the game if he is younger than 18.) It would be time-consuming to write a script that used 100 `If` statements or one `Select Case` statement with 100 corresponding `Case` statements just to test a person's age. Instead, you could save a lot of time by comparing the user's age against a range of values. To accomplish this task, you could use the VBScript less-than operator (`<`) as follows:

```

intUserAge = InputBox("How old are you?")
If intUserAge < 18 Then
  MsgBox "Sorry but you are too young to play this game."
  WScript.Quit()
Else
  MsgBox "OK. Let's play!"
End If

```

In this example, the VBScript `InputBox()` function collects the user's age and assigns it to a variable called `intUserAge`. An `If` statement then checks to see whether `intUserAge` is less than 18, and if it is, the game is stopped.

Another way you could write the previous example is using the VBScript less-than-or-equal to operator (`<=`), like this:

```

If intUserAge <= 17 Then

```

If you use the <= operator, this statement will not execute if the user is 17 or fewer years old. VBScript also supplies greater-than operator (>) and greater-than-or-equal-to operator (>=), allowing you to invert the logic used in the preceding example:

```
intUserAge = InputBox("How old are you?")
If intUserAge > 17 Then
    MsgBox "OK. Let's play!"
Else
    MsgBox "Sorry but you are too young to play this game."
    WScript.Quit()
End If
```

Table 5.1 lists VBScript comparison operators.

TABLE 5.1 VBSCRIPT COMPARISON OPERATORS

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

© Jerry Lee Ford, Jr. All Rights Reserved.

VBScript does not impose an order or precedence on comparison operators like it does with arithmetic operators. Instead, each comparison operation is performed in the order in which it appears, going from left to right.

Back to the Planet Trivia Quiz Game

Now let's return to where we began this chapter, by developing the Planet Trivia Quiz game. In this program, you will create a VBScript that presents the player with a quiz about Planet Trivia. The game presents questions, collects the player's answers, scores the final results, assigns a rank to the player based on his score, and finally creates a summary text report. By working your way through this project, you will work more with both the If and Select Case statements. You'll also learn how to work with a number of built-in VBScript functions.

Game Development

The following steps outline the process you'll need to go through to complete the development of the game:

1. Add the standard documentation template and fill in its information.
2. Define the constants and variables that will be used by the script.
3. Create the splash screen that welcomes the user to the game and determines whether the user wants to play the game.
4. Use the `InputBox()` function to display questions, collect the player's answers, and to add logic to determine whether the player's answers are right or wrong.
5. Use the `Select Case` statement to determine the rank to be assigned to the player, based on the number of correctly answered questions.
6. Display the player's score and rank.

Beginning the *Planet Trivia Quiz Game*

Begin this script by opening your script editor and cutting and pasting your script template from another script. Then go back and modify the template with information relevant to the Planet Trivia Quiz game:

```
*****
'Script Name: PlanetTrivia.vbs
'Author: Jerry Ford
'Created: 02/04/14
'Description: This script creates a Planet Trivia Quiz game.
*****

'Perform script initialization activities
Option Explicit
```

Setting Up Constants and Variables

The next step is to define the variables and constants used by the script:

```
Dim intPlayGame, strSplashImage, strAnswerOne, strAnswerTwo, strAnswerThree
Dim strAnswerFour, strAnswerFive, intNumberCorrect, strRank
Dim objFsoObject

Const cTitlebarMsg = "The Planet Trivia Quiz Game"

'Start the user's score at zero
intNumberCorrect = 0
```

The `intNumberCorrect` variable is used to count the number of quiz answers the player gets right. I set `intNumberCorrect` equal to 0 here to ensure that it has a value. It is possible that the player will miss every answer; if that were to happen, this variable might not otherwise get set. I'll explain what each of these variables is used for as we go through the rest of the script-development process.

Creating a Splash Screen

Let's create a spiffy splash screen that asks the user whether he wants to play the game. As you can see, I added a graphic to spice up things a bit. Graphic development of this type takes a little time, as well as some trial and error.

```
'Display the splash screen and ask the user if he or she wants to play
strSplashImage = vbCrLf & vbCrLf & vbCrLf & space(15) & _
    "W E L C O M E      T O      T H E" & vbCrLf & vbCrLf & _
    space(9) & "P L A N E T      T R I V I A      Q U I Z" & vbCrLf & _
    vbCrLf & space(35) & "G A M E !" & vbCrLf & vbCrLf & vbCrLf & vbCrLf & _
    "Would you like to test your knowledge of the solar system?"
intPlayGame = MsgBox(strSplashImage, 36, cTitlebarMsg)
```

```
If intPlayGame = 6 Then 'User elected to play the game
```

The splash screen is created using the VBScript `InputBox()` function. It displays the invitation to play the game as well as Yes and No buttons. The value of the button the user clicks is assigned to the `PlayGame` variable (that is, `PlayGame` will be set equal to 6 if the player clicks on the Yes button).

Now let's check to see whether the user wants to play the game.

```
If intPlayGame = 6 Then 'User elected to play the game
'Insert statements that make up the game here
.
.
.
Else 'User doesn't want to play
    MsgBox "Thank you for taking the Planet Trivia Quiz © Jerry Ford 2014." & _
        vbCrLf & vbCrLf & "Please play again soon!", , cTitlebarMsg
    WScript.Quit()
End If
```

As you can see, the first statement checks to see whether the user clicked the Yes button. I left some room to mark the area where you will need to add the statements that actually make up the game in case the user does want to play. If the user clicked No, then the VBScript displays a “thank you” message and terminates its execution using the `WScript` object's `Quit()` method.

Display Quiz Questions and Collect the Player's Answers

The next step is to add the questions that make up the game. The following questions make up the quiz:

- What is the name of the planet that is closest to the sun?
- What is the name of the red planet?
- Venus is how many planets away from the sun?
- What planet was named after the god of the sea?
- What is the name of your favorite planet?

The statements that display and grade the first quiz questions are as follows:

```
strAnswerOne = InputBox("What is the name of the planet that is closest to " & _  
    "the sun?", cTitlebarMsg)
```

```
If LCase(strAnswerOne) = "mercury" Then  
    intNumberCorrect = intNumberCorrect + 1  
End If
```

First the VBScript `InputBox()` function displays the question. The answer typed by the user is then assigned to a variable named `strAnswerOne`. Next, an `If` statement is used to interrogate the player's answer and determine whether it's correct. The VBScript `LCase()` function is used to convert the answer the player types to all lowercase. That way, it doesn't matter how the player types the answer. For example, `MERCURY`, `mercury`, `MeRcUrY`, and `Mercury` would all end up as `mercury`. Finally, if the player provides the correct answer, then the value of `intNumberCorrect` is increased by 1.

As you can see, the second quiz question, shown next, is processed exactly like the first question. The only difference is the content of the question itself and the name of the variable used to store the player's answer to the question.

```
strAnswerTwo = InputBox("What is the name of the red planet?", cTitlebarMsg)
```

```
If LCase(strAnswerTwo) = "mars" Then  
    intNumberCorrect = intNumberCorrect + 1  
End If
```

The statements that make up and process the quiz's third question are shown next.

```
strAnswerThree = InputBox("Venus is how many planets away from the " & _  
    "sun?", cTitlebarMsg)
```



```
If CStr(strAnswerThree) = "2" Then
    intNumberCorrect = intNumberCorrect + 1
End If
```

The statements that make up the fourth question follow the same pattern as the first two questions.

```
strAnswerFour = InputBox("What planet was named after the god of the " & _
    "sea?", cTitlebarMsg)
```

```
If LCase(strAnswerFour) = "neptune" Then
    intNumberCorrect = intNumberCorrect + 1
End If
```

The construction of the fifth question, shown next, merits some additional examination. First, the fourth statement uses the VBScript `LCase()` function to convert the player's answer to all lowercase. The VBScript `Instr()` function then takes the answer and searches the string `mercuryvenusearthmarsjupitersaturnuranusneptune` to see whether it can find a match. This string contains a list of names belonging to the solar system's planets.

```
strAnswerFive = InputBox("What is the name of your favorite planet?",
    cTitlebarMsg)
```

```
If Len(strAnswerFive) > 3 Then
    If Instr(1, "mercuryvenusearthmarsjupitersaturnuranusneptune",
        LCase(strAnswerFive), 1) <
        <> 0 Then
        intNumberCorrect = intNumberCorrect + 1
    End If
End If
```

So the `InStr()` function begins its search starting with the first character of the string to see whether it can find the text string that it's looking for (that is, mercury, venus, earth, mars, jupiter, saturn, uranus, or neptune). The syntax of the `Instr()` function is as follows:

```
InStr([start, ]string1, string2[, compare])
```

`start` specifies the character position in the script, from left to right, where the search should begin. `string1` identifies the string to search. `string2` identifies the text to search for, and `compare` specifies the type of search to perform. A value of 0 specifies a binary comparison, and a value of 1 specifies a textual comparison.

The `InStr()` function returns the position of the first occurrence of string within another string. If it does not find a matching text string in the list, then it will return to 0, in which case the user provided

the wrong answer. Otherwise, it will return the starting character position where the search string was found. If the search string is found in the list, then the value returned by the `InStr()` function will be greater than 1, in which case the value of `intNumberCorrect` will be incremented by 1.

However, it is always possible that the player doesn't know the name of a planet, and that he or she will just type a character or two, such as the letter A. Because the letter A is used in at least one of the planet's names, the player would end up getting credit for a correct answer to the question. Clearly, this is not good. To try to keep the game honest, I used the VBScript `Len()` function to be sure that the user provided at least a four-character name (that is, the length of the shortest name belonging to any planet). This way, the player must know at least the first four characters of a planet's name to get credit for a correct answer.

Scoring the Player's Rank

At this point, the script has enough logic to display all five questions and determine which ones the player got correct. In addition, it has been keeping track of the total number of correct answers. What you need to do next is add logic to assign the player a rank based on the number of correctly answered questions. This can be done using a `Select Case` statement, like this:

```
Select Case intNumberCorrect
    Case 5 'User got all five answers right
        strRank = intNumberCorrect & " stars."
    Case 4 'User got four of five answers right
        strRank = intNumberCorrect & " stars."
    Case 3 'User got three of five answers right
        strRank = intNumberCorrect & " stars."
    Case 2 'User got two of five answers right
        strRank = intNumberCorrect & " stars."
    Case 1 'User got one of five answers right
        strRank = intNumberCorrect & " star."
    Case 0 'User did not get any answers right
        strRank = intNumberCorrect & " stars."
End Select
```

The variable `intNumberCorrect` contains the number of answers that the player has correctly answered. The value of this variable is then compared against six possible cases to determine how many stars to award the player.

Displaying the Player's Score and Rank

The last thing the game does is display the player's score and rank in a pop-up dialog box:

```
MsgBox "You answered " & intNumberCorrect & " out of 5 correct." & _
    vbCrLf & vbCrLf & "You earned " & _
    strRank, , cTitlebarMsg
```

As you can see, there is not much to this last statement. All you need to do is to use the VBScript `MsgBox()` function, the `strNumberCorrect` and `strRank` variables, and the `vbCrLf` constant to display the message for the player to see.

The Fully Assembled Script

That's it! You have all of the information you need to create the Planet Trivia Quiz game. When you are ready, start the game and put it through its paces. When you do, be sure to supply different combinations of correct and incorrect answers and then validate that the game is correctly processing your answer. When you are confident that things are working like they are supposed to, pass the game along to some of your Trekkie friends and see what they think.

Summary

This chapter covered a lot of ground. You learned how to use the `If` and `Case Select` statements in a number of different ways. Using this new information, you updated the Rock, Paper, and Scissors game and created the Planet Trivia Quiz game.

Challenges

1. Modify the Planet Trivia Quiz game so that it asks the player for his name and then uses the player's name at the end of the game to address him according to his rank.
2. Modify the Planet Trivia Quiz game so that it displays the correct answer for any question that the player misses.
3. Expand the Planet Trivia Quiz game by adding more questions. Store a list of questions in an array and then use a `For...Each...Next` loop to display and process both the questions and the player's answers.

6

Processing Collections of Data

In this chapter, you'll learn how to use a number of VBScript statements that can help you develop scripts capable of processing extremely large amounts of information—in most cases with only a handful of script statements. Using these statements, you can establish loops within your scripts to let the user iteratively enter as much data as needed, to process the contents of array, to read the content's files, and to control the execution of VBScript games. I'll also show you how to create shortcuts for your scripts, as well as how to place them on the Windows desktop and Programs menu. Specifically, you will learn the following:

- How to work with five different types of VBScript loops
- How to use loops to control the execution of your scripts (and games)
- How to programmatically create Windows shortcuts and use them to configure Windows resources such as the desktop and Programs menu

Project Preview: The Guess a Number Game

In this chapter's project, you'll create a script that plays a number guessing game. The game generates a random number between 1 and 100, then instructs the player to try to guess it. As the player enters guesses, the game provides the player with hints to help him figure out the number. If the player types an invalid guess, the game will let him know that only numeric input is accepted. The player may quit at any time by simply clicking on the Cancel button or by failing to type a guess before clicking OK. When the player guesses the correct answer, the game displays the number of guesses it took him to find the correct answer. Figures 6.1 through 6.4 provide a sneak peek of the game's interaction when executed on a computer running Windows 8.1.

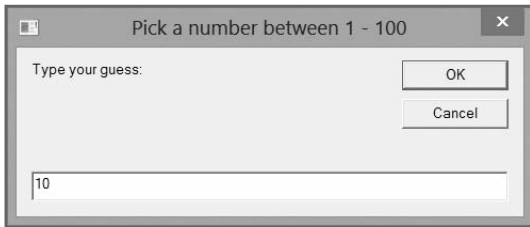


Figure 6.1 The Guess a Number game begins by prompting the player to type a number between 1 and 100. © 2014 Cengage Learning.

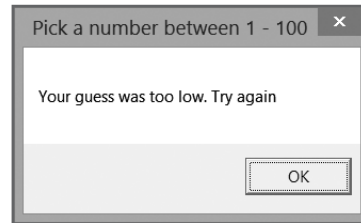


Figure 6.2 The game tells the player to try again if his guess is too low. © 2014 Cengage Learning.

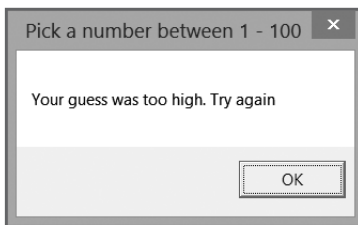


Figure 6.3 The game tells the player to try again if his guess is too high. © 2014 Cengage Learning.

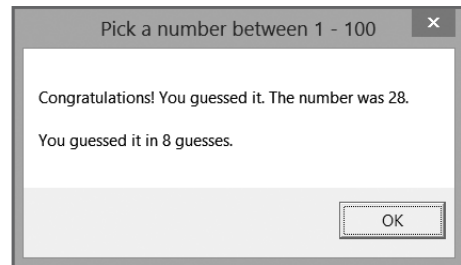


Figure 6.4 When the player correctly guesses the game's number, the player is congratulated. © 2014 Cengage Learning.

The game uses a VBScript loop to continue executing until either the player guesses the correct answer or quits. By developing and working with this game, you will solidify your understanding of iterative programming while also learning specifically how to apply a loop using VBScript.

Adding Looping Logic to Scripts

One of VBScript's best programming features is its strong support for *looping* or *iterative statements*. VBScript provides five different statements that can create loops. Loops provide your scripts with the capability to process large collections of data using a minimal number of programming statements that are repeatedly executed, either for each member of the collection or for a specified number of times.

The following list provides a high-level description of each of VBScript's looping statements:

- **For...Next.** This establishes a loop that iterates for a specified number of times.
- **For...Each...Next.** This establishes a loop that iterates through all the properties associated with an object.
- **Do...While.** This establishes a loop that iterates for as long as a stated condition continues to be true.

Definition

A *loop* is a collection of statements repeatedly executed to facilitate the processing of large amounts of data.

- **Do...Until.** This establishes a loop that iterates until a stated condition finally becomes true.
- **While...Wend.** This establishes a loop that iterates for as long as a condition continues to be true.

The For...Next Statement

The For...Next statement is used to create loops that execute a specific number of times. For example, if you're creating a game that requires the player to enter five guesses, you could use a For...Next loop to control the logic that supports the data-collection portion of the script.

The syntax for the For...Next statement is as follows:

```
For counter = begin To end [Step StepValue]
    statements
Next
```

counter is a variable used to control the execution of the loop. *begin* is a numeric value that specifies the starting value of the *counter* variable. *end* specifies the ending value for the counter variable (that is, the value that, when reached, terminates the loop's execution). *StepValue* is an optional setting that specifies the increment that the For...Next statement uses when incrementing the value of *counter* (that is, the value added to *counter* at the end of each iteration). If omitted, the value assigned to *StepValue* is always 1.

To better understand the operation of a For...Next loop, look at one example that collects data without using a loop and one that collects the same data using a For...Next loop. In the following example, let's assume you're creating a game in which the player is expected to enter the name of his five favorite foods. You could always handle this type of situation as follows:

```
Dim strFoodList
strFoodList = " "
strFoodList = strFoodList & " " & InputBox("Type the name of a food " & _
    "that you really like.")
strFoodList = strFoodList & " " & InputBox("Type the name of a food " & _
    "that you really like.")
strFoodList = strFoodList & " " & InputBox("Type the name of a food " & _
    "that you really like.")
strFoodList = strFoodList & " " & InputBox("Type the name of a food " & _
    "that you really like.")
strFoodList = strFoodList & " " & InputBox("Type the name of a food " & _
    "that you really like.")
MsgBox "You like : " & strFoodList
```

As you can see, this example repeats the same statement over and over again to collect user input. Then, as proof that it did its job, it displays the data it collected using the MsgBox() function.

Collecting five pieces of data like this is a bit of a chore. Now imagine a situation in which you want to collect a lot more data. Instead of typing the same statement over and over again, as done in the previous example, you can use the For...Next loop.

```
Dim intCounter, strFoodList
strFoodList = " "

For intCounter = 1 To 5
    strFoodList = strFoodList & " " & InputBox("Type the name of a " & _
        "food that you really like.")
Next

MsgBox "You like : " & strFoodList
```

Figure 6.5 shows the output produced by this example when executed on a computer running Windows 8.1.



Figure 6.5 Using a For...Next loop to collect and process user input.
© 2014 Cengage Learning.

Notice this new script is two lines shorter than the previous example. Unlike the previous example, other than the value of the loop's ending value, this script does not have to be modified to accommodate the collection of additional data. For example, to change the For...Next loop so that it can accommodate the collection of 10 pieces of data, all you'd have to do is modify it like this:

```
For intCounter = 1 To 10
    strFoodList = strFoodList & " " & InputBox("Type the name of a " & _
        "food that you really like.")
Next
```

Optionally, you can use the Exit For statement to break out of a For...Next loop at any time, like this:

```
Dim intCounter, strFoodList, strNewFood
strFoodList = " "

For intCounter = 1 To 5
    strNewFood = InputBox("Type the name of a food that you really like.")
    If strNewFood = "beans" Then
        MsgBox "Sorry, but I don't want to talk to anyone who likes beans!"
```

Exit For

```
End If
strFoodList = strFoodList & " " & strNewFood
Next

MsgBox "You like : " & strFoodList
```

In this example, the assignment of data has been split into two different statements. The first of these statements assigns the name of the food entered by the user to a variable called `strNewFood`. The value of `strNewFood` is then added to the list of foods liked by the user only if it is not beans, in which case the script displays a message and then terminates the execution of the `For...Next` loop. As a result, only the foods entered by the user up to the point where beans was typed are displayed.

Let's look at one last example before we examine the other loop statements supported by VBScript. In this example, the `For...Next` statement's optional keyword `Step` has been added to change the behavior of the loop.

```
Dim intCounter
For intCounter = 1 To 9 Step 3
    WScript.Echo intcounter
Next
```

In this example, the script will display the value of the *counter* variable, which is used to control the loop's execution. Instead of counting to nine by ones, the script will count by threes, as demonstrated here:

```
C:\>CScript TextScript.vbs
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
1
4
7
```

```
C:\>
```

The For Each...Next Statement

VBScript's `For Each...Next` statement is a programming tool for working with all the properties associated with objects. Every object has a number of properties associated with it. Using the `For Each...Next` loop, you could write a script to loop through an object's properties.

The syntax of the For...Each...Next statement is as follows:

```
For Each element In collection
    statements
Next [element]
```

element is a variable representing a property associated with the *collection* (or object). Look at the following example:

```
Dim objFsoObject, objFolderName, strMember, strFileList, strTargetFolder

Set objFsoObject = CreateObject("Scripting.FileSystemObject")
Set objFolderName = objFsoObject.GetFolder("C:\Temp")

For Each strMember in objFolderName.Files
    strFileList = strFileList & strMember.name & vbCrLf
Next

MsgBox strFileList, , "List of files in " & objFolderName
```

This example begins by defining its variables and then establishing an instance of the `FileSystemObject` object. It then uses the `FileSystemObject` object's `GetFolder()` method to set a reference to a folder. Next, using the folder reference, a For Each...Next loop processes all the files (which, in this case, are considered to be properties of the folder) stored within the folder. As the For Each...Next loop executes, it builds a list of files stored within the folder and uses the `vbCrLf` constant to format the list in an attractive manner. Figure 6.6 shows the results displayed when this script is executed on a computer running Windows 7.

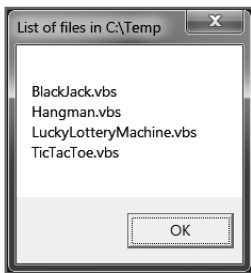


Figure 6.6 Using a For Each...Next loop to process the contents of a folder.

© 2014 Cengage Learning.

For Each...Next loops also are excellent programming tools for processing the contents of arrays. For example, the following statements are all that are needed to process and display an array called `astrGameArray`, and to display each of its elements:

```
For Each intCount In astrGameArray
    strMessage = strMessage & intCounter & vbCrLf
Next
WScript.Echo strMessage
```

To learn more about arrays and see a more complete example of a script that uses the For Each...Next statement, refer to the section “Processing Array Contents” in Chapter 4, “Constants, Variables, Arrays, and Dictionaries.”

The Do...While Statement

The Do...While statement creates a loop that runs as long as a specified condition is true. VBScript supports two different versions of the Do...While loop. The syntax for the first version of the Do...While loop is as follows:

```
Do While condition
    statements
Loop
```

condition is expressed in the form of an expression, like this:

```
intCounter = 0
Do While intCounter < 10
    intCounter = intCounter + 2
Loop
```

In this example, the expression (`intCounter < 10`) allows the loop to continue as long as the value of `intCounter` is less than 10. The value of `intCounter` is initially set to 0, but is increased by 2 every time the loop executes. As a result, the loop iterates five times.

As the `While` keyword has been placed at the beginning of the loop, the loop will not execute if the value of `counter` is already 10 or greater.

The syntax for the second format of the Do...While statement is as follows:

```
Do
    statements
Loop While condition
```

As you can see, the `While` keyword had been moved from the beginning to the end of the loop. Therefore, the loop will always execute at least once, even if the condition is initially false.

Let's look at another example of the Do...While loop in action. In this example, the Do...While loop is set up to collect names and phone numbers for an address book. The loop uses the VBScript `InputBox()` function to collect the names and phone numbers. The names and addresses are added to a variable string and formatted such that, when displayed, each entry is listed on a separate line. The user may enter as many names and numbers as he wants. When done adding new address book entries, the user types `Quit` as the final entry.

```
Dim intCounter, strAddressBook, strAddressEntry

intCounter = 0

Do While strAddressEntry <> "Quit"
    strAddressEntry = InputBox("Please type a name, a space, and then " & _
        "the person's phone number", "Personal Address Book")
    If strAddressEntry <> "Quit" Then
        strAddressBook = strAddressBook & strAddressEntry & vbCrLf
        intCounter = intCounter + 1
    End If
Loop

MsgBox strAddressBook, , "New Address Book Entries = " & intCounter
```

Figure 6.7 shows the results that are displayed on a computer running Windows 7 when four names are entered.

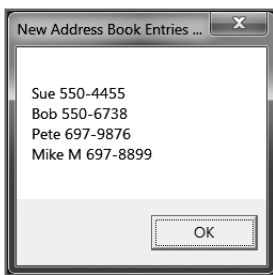


Figure 6.7 Using a Do...While loop to collect new address book entries.

© 2014 Cengage Learning.

Alternatively, you could have written this as shown next. In this example, the `While` keyword and its associated condition have been moved to the end of the loop. However, the script still operates exactly as in the previous example.

```
Dim intCounter, strAddressBook, strAddressEntry

intCounter = 0

Do
    strAddressEntry = InputBox("Please type a name, a space, and then " & _
        "the person's phone number", "Personal Address Book")
    If strAddressEntry <> "Quit" Then
        strAddressBook = strAddressBook & strAddressEntry & vbCrLf
        intCounter = intCounter + 1
    End If
```

```
Loop While strAddressEntry <> "Quit"
```

```
MsgBox strAddressBook, , "New Address Book Entries = " & intCounter
```

One of the dangers of working with loops is that you may accidentally create a loop that has no way of terminating its own execution. This is an endless loop. Endless loops run forever, needlessly consuming computer resources and degrading a computer's performance. For example, look at the following:

```
intCounter = 0
Do While intCounter < 10
    intCounter = intCounter + 1
WScript.Echo intCounter
Loop
```

Definition

An *endless loop* is a piece of code that either has no terminating condition, or has a terminating condition that can never be met, and as such it iterates indefinitely.

When executed, this script counts from 1 to 10.

Now look at the next script:

```
intCounter = 0
Do While intCounter < 10
    intCounter = intCounter - 1
WScript.Echo intCounter
Loop
```

It looks almost exactly like the previous example, only instead of incrementing the value of `intCounter` by 1, it increments the value of `intCounter` by `-1`, creating an endless loop. One way to protect against the creation of an endless loop is to put in a safety net, like this:

```
intCounter = 0
Do While intCounter < 10
    intCounter = intCounter - 1
    intNoExecutions = intNoExecutions + 1
WScript.Echo intCounter
If intNoExecutions > 99 Then
    Exit Do
End If
Loop
```

As you can see, I added to the script a variable called `intNoExecutions` that I then used to keep track of the number of times that loop iterated. If the loop iterates 100 times, then something is wrong. So I added an `If` statement to test the value of `intNoExecutions` each time the loop is processed and to execute the `Exit Do` statement in the event that something goes wrong. Of course, there is no substitute for good program design and careful testing.

The Do...Until Statement

The VBScript Do...Until statement creates a loop that executes as long as a condition is false (that is, until it becomes true). VBScript supports two versions of the Do...Until statement. The syntax for the first version is as follows:

```
Do Until condition
    statements
Loop
```

Let's look at an example that demonstrates how this loop works. In this example, shown next, the script prompts the player to answer a question and uses a Do...Until loop to allow the user up to three chances to correctly answer the question.

```
Dim intMissedGuesses, strPlayerAnswer

intMissedGuesses = 1

Do Until intMissedGuesses > 3
    strPlayerAnswer = InputBox("Where does Peter Pan live?")
    If strPlayerAnswer <> "Neverland" Then
        intMissedGuesses = intMissedGuesses + 1
        If intMissedGuesses < 4 Then
            MsgBox "Incorrect: You have " & 4 - intMissedGuesses & _
                " guesses left. Please try again."
        Else
            MsgBox "Sorry. You have used up all your chances."
        End If
    Else
        intMissedGuesses = 4
        MsgBox "Correct! I guess that you must believe in Faith, Trust " & _
            "and Pixie Dust!"
    End If
Loop
```

In this example, the loop has been set up to execute until the value of a variable named `intMissedGuesses` becomes greater than 3. The variable is initially set equal to 1 and is incremented by 1 each time the loop executes, unless the player provides a correct answer, in which case the script sets the value of `intMissedGuesses` to 4 to arbitrarily terminate the loop's execution.

Figure 6.8 demonstrates the execution of this script on a computer running Windows 8.1 by showing the pop-up dialog box that appears if the player guesses incorrectly on his first attempt to answer the question.

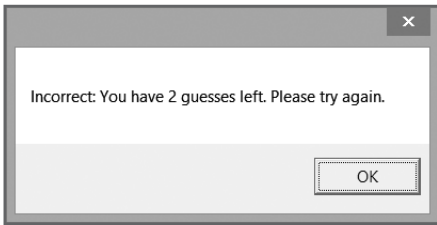


Figure 6.8 Using a Do...Until loop to provide the player with three chances to correctly answer a question. © 2014 Cengage Learning.

The syntax of the second form of the Do...Until statement is as follows:

```
Do
    statements
Loop Until condition
```

As you can see, the Until keyword and its associated condition have been moved from the beginning to the end of the loop, thus ensuring the loop executes at least once.

The While...Wend Statement

The While...Wend statement creates a loop that executes as long as a tested condition is true.

The syntax for this loop is as follows:

```
While condition
    Statements
Wend
```

The Do...While and Do...Until loops provide the same functionality as the While...Wend loop. The general rule of thumb, therefore, is that you should use one of the Do loops in place of this statement. However, I'd be remiss if I failed to show you how this statement works, so take a look at the following example:

```
Dim intCounter, strCountList

intCounter = 0

While intCounter < 10
    intCounter = intCounter + 1
    strCountList = strCountList & intCounter & vbCrLf
Wend

MsgBox "This is how to count to 10:" & vbCrLf & vbCrLf & _
    strCountList, , "Counting Example"
```

This example begins by initializing two variables. `intCount` is used to control the loop's execution. `strCountList` is used to build a formatted script containing the numbers counted by the script. The loop itself iterates 10 times. Figure 6.9 shows the output created by this example on a computer running Windows 7 when run using the `WScript.exe` execution host.

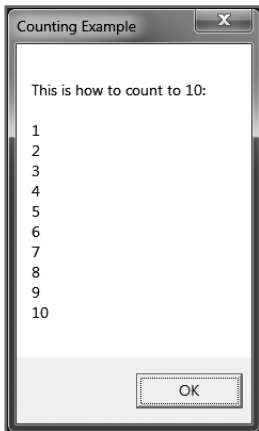


Figure 6.9 Counting to 10 using a `While...Wend` loop.

© 2014 Cengage Learning.

Back to the Guess a Number Game

Let's turn our attention back to the Guess a Number game. In this game, the player is prompted to guess a randomly generated number between 1 and 100. Each time the player takes a guess, the script will check to see if the correct number was guessed. If not, the script will provide a hint to help the player on his next guess.

Developing this script will enhance your knowledge and understanding of working with the `Do...Until` loop. You will also work with the `If` statement, and learn how to work with a number of new built-in VBScript functions.

Designing the Game

The Guess a Number game begins by asking the player to guess a number between one and 100 and then helps the user guess the number by providing hints. This project has five steps. These steps are as follows:

1. Add the standard documentation template and define any variables, constants, or objects used by the script.
2. Generate a random number between one and 100.
3. Create a loop that runs until the player either guesses the correct answer or gives up.
4. Test the player's answer to see whether it's valid.
5. Test the player's answer to see whether it is too low, too high, or correct.

As a kind of project bonus, after you have completed the Guess a Number game, I'll show you how to create a VBScript desktop shortcut for it. I'll also show you how to use shortcuts to configure the Windows 7 Programs menu and the Windows 8.1 Apps group.

Beginning the Guess a Number Game

Begin by creating a new script and adding your script template.

```
'*****
'Script Name: GuessANumber.vbs
'Author: Jerry Ford
'Created: 02/20/14
'Description: This script plays a number-guessing game with the user
'*****

'Initialization Section
Option Explicit
```

Next, create a constant and assign it the text message to be used in the title bar of the script's pop-up dialog boxes.

```
Const cGreetingMsg = "Pick a number between 1 - 100"
```

Define four variables as shown. Use `intUserNumber` is to store the player's numeric guess. `intRandomNo` stores the script's randomly generated number. `strOkToEnd` is a variable the script uses to determine whether the game should be stopped, and `intNoGuesses` keeps track of the number of guesses the player makes.

```
Dim intUserNumber, intRandomNo, strOkToEnd, intNoGuesses
```

Finally, set the initial value of `intNoGuesses` to 0, like this:

```
intNoGuesses = 0
```

Generating the Game's Random Number

The following statements are next and are responsible for generating the game's random number:

```
'Generate a random number
Randomize
intRandomNo = FormatNumber(Int((100 * Rnd) + 1))
```

The `Randomize` statement ensures that a random number is generated each time the game is played. The last statement uses the following built-in VBScript functions to generate a number between one and 100.

- **Rnd()**. This returns a randomly generated number.
- **Int()**. This returns the integer portion of a number.
- **FormatNumber()**. This returns an expression that has been formatted as a number.

Creating a Loop to Control the Game

Now you'll need to set up the Do...Until loop that controls the game's execution. In this example, the loop executes until the value assigned to the strOkToEnd variable is set to yes.

```
Do Until strOkToEnd = "yes"
    'Prompt users to pick a number
    intUserNumber = InputBox("Type your guess:",cGreetingMsg)
    intNoGuesses = intNoGuesses + 1
    .
    .
    .
Loop
```

As you can see, the only statement inside the loop, for now, prompts the player to guess a number and keeps track of the number of guesses made by the player.

Testing Player Input

Now let's put together the code that performs validation of the data supplied by the player.

```
'See if the user provided an answer
If Len(intUserNumber) <> 0 Then
    'Make sure that the player typed a number
    If IsNumeric(intUserNumber) = True Then
        .
        .
        .
    Else
        MsgBox "Sorry. You did not enter a number. Try again.", , cGreetingMsg
    End If
Else
    MsgBox "You either failed to type a value or you clicked on Cancel. " & _
        "Please play again soon!", , cGreetingMsg
    strOkToEnd = "yes"
End If
```

The first validation test is performed using the built-in VBScript Len() function. It is used to ensure that the player actually typed a number before clicking the OK button. If the player's input is not 0 characters long, the game continues to the next test. Otherwise, an error message is displayed, and the value of strOkToEnd is set to yes, terminating the loop and ending the game.

If the length test is passed, then the script performs a second validation test on the player's input. This time, the built-in VBScript IsNumeric() function is used to make sure that the player typed a number instead of

a letter or other special character. If a number was typed, then the game continues. If a number was not typed, then an error message is displayed, but the game continues with the next iteration of the loop.

Determine Whether the Player's Guess Is High, Low, or Correct

There are three more sets of statements that need to be added to the script. They will be inserted one after another, just after the If statement that performs the game's second validation test.

The first of these three sets of statements is shown here. It begins by verifying that the user's guess matches the game's randomly selected number. Then it displays a message congratulating the player, showing the random number, and showing the number of guesses that it took for the player to guess it. Finally, the value of `strOkToEnd` is set equal to `yes`. This terminates the loop and allows the game to end.

```
'Test to see if the user's guess was correct
If FormatNumber(intUserNumber) = intRandomNo Then
MsgBox "Congratulations! You guessed it. The number was " & _
    intUserNumber & "." & vbCrLf & vbCrLf & "You guessed it " & _
    "in " & intNoGuesses & " guesses.", ,cGreetingMsg
    strOkToEnd = "yes"
End If
```

The second of the three sets of statements provides the player with help if his guess is too low. The value of `strOkToEnd` is set equal to `no`. This ensures that the loop that controls the game will continue.

```
'Test to see if the user's guess was too low
If FormatNumber(intUserNumber) < intRandomNo Then
    MsgBox "Your guess was too low. Try again", ,cGreetingMsg
    strOkToEnd = "no"
End If
```

Finally, the last collection of statements provides the player with help if his guess is too high. The value of `strOkToEnd` is set equal to `no`. This ensures that the loop that controls the game will continue.

```
'Test to see if the user's guess was too high
If FormatNumber(intUserNumber) > intRandomNo Then
    MsgBox "Your guess was too high. Try again", ,cGreetingMsg
    strOkToEnd = "no"
End If
```

The Final Result

Okay, it's time to run the script and see whether it works as promised (don't worry, it will). After testing to see whether the script works as expected, retest it to see whether you can break it. For example, try feeding it special characters or letters instead of numbers. Once you're satisfied with the operation of the script, keep reading. I have one more little goodie for you in this chapter.

Creating Shortcuts for Your Game

Until now, you have been running your scripts in one of two ways. One is by opening the Windows Console and typing the name of an execution host followed by the path and filename of your scripts at the Windows command prompt. The other is by locating the folder in which the script resides and opening it (that is, double-clicking it).

Windows provides shortcuts as a convenient alternative for executing Windows applications and scripts from the Windows desktop. A shortcut provides access to a Windows resource without requiring the user to find or even know the actual location of the resource that it represents. For example, just about any new application that you install on your computer automatically adds an application shortcut to the Programs menu located on the Windows Start menu. In addition, most application installation procedures offer to add a shortcut for the application on the Windows desktop.

Using VBScript and the WSH, you can create a setup script that configures shortcuts for your VBScript games in any of these locations. Of course, you can always manually create shortcuts for your scripts, but the advantage of scripting their setup is that, once written, you can re-create these shortcuts on any computer. For example, if you purchase a new computer, all you'll have to do is copy your VBScripts from your older computer and then run your VBScript setup script, and all your shortcuts will be re-created. Likewise, if you give copies of your VBScript games to all your friends, all they'll have to do to set up shortcuts for the scripts is to run the setup script.

Definition

Shortcuts are links or pointers to Windows objects. These objects can be just about anything, including Windows applications, folders, files, printers, disk drives, and scripts.

Examining Shortcut Properties

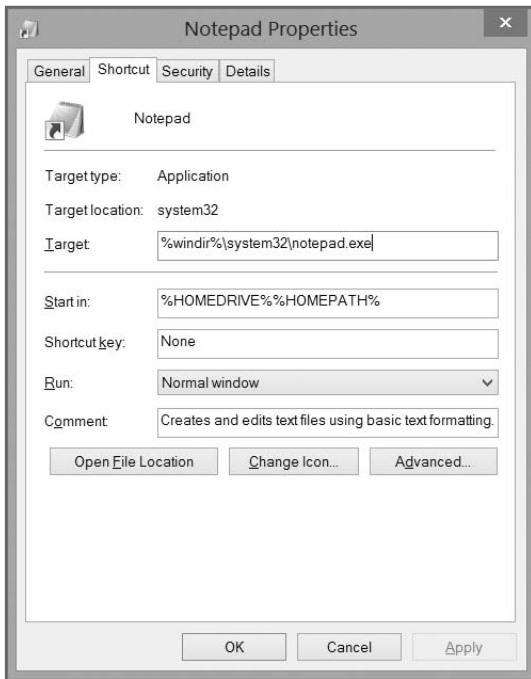
Windows shortcuts are identified by a small arrow in the lower-left corner of the icon that represents them. Shortcuts contain information, in the form of properties, about the Windows resources with which they are associated. The most important of these properties is the path and name of the Windows resource that the shortcut represents.

You can view the properties associated with any shortcut by right-clicking the shortcut and selecting Properties. The shortcut's Properties dialog box appears. Click the Shortcut tab to view these properties, as shown in Figure 6.10.

Creating Desktop Shortcuts

You can create a desktop shortcut in just five simple steps. To demonstrate, let's create a shortcut for the GuessANumber.vbs game on the Windows desktop.

The first step in creating the game's shortcut is to establish an instance of the `WshShell` object. The script will need to use this object's `SpecialFolders` property to access the folder that represents the Windows desktop. In addition, you'll need to use the `WshShell` object to instantiate the `WshShortcut` object in order to set shortcut properties.



Definition

Special folders are a Windows management tool used to organize and manage the contents of a number of Windows features, including the Programs menu and desktop.

Figure 6.10 Examining the properties associated with a shortcut to the Windows Notepad application.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

The following statement establishes an instance of the `WshShell` object:

```
Set objWshShl = WScript.CreateObject("WScript.Shell")
```

The second step in creating the shortcut is to set up a reference to the folder where the shortcut is to reside. In Windows, everything, including the Windows desktop, is represented as a folder. Therefore, to add a shortcut to the Windows desktop, all you have to do is save the shortcut in a special folder called "Desktop" by specifying a value for the `WshShell` object's `SpecialFolder` property.

```
strDesktopFolder = objWshShl.SpecialFolders("Desktop")
```

The third step required to set up the desktop shortcut is to use the `WshShell` object's `CreateShortcut()` method to define the shortcut and instantiate the `WshShortcut` object.

```
Set objNewShortcut = objWshShl.CreateShortcut(strDesktopFolder & _
    "\\GuessANumber.lnk")
```

`strDesktopFolder` provides a reference to the location of the Windows desktop and `\\GuessANumber.lnk` is the name to be assigned to the shortcut.

The fourth step in creating the new shortcut is to configure properties associated with the shortcut. The `WshShortcut` object provides access to these properties, which are listed in Table 6.1.

TABLE 6.1 PROPERTIES OF THE WSHSHORTCUT OBJECT

Property	Description
Arguments	Sets arguments to be passed to the application or script associated with the shortcut
Description	Adds a comment to the shortcut
Hotkey	Sets a keyboard keystroke sequence that can be used to activate the application associated with the shortcut
IconLocation	Sets the shortcut's icon
TargetPath	Sets the path and file name of the object associated with the shortcut
WindowStyle	Sets the window style used when the application associated with the shortcut is opened (e.g., normal, minimized, or maximized)
WorkingDirectory	Sets the default working directory or folder for the application associated with the shortcut

© Jerry Lee Ford, Jr. All Rights Reserved.

Only the `TargetPath` property must be set to create a shortcut. Configuration of the remaining shortcut properties is optional. The following statement configures the `TargetPath` property by setting it to `C:\GuessANumber.vbs`:

```
objNewShortcut.TargetPath = "C:\ GuessANumber.vbs"
```

Examples of how to set other properties are as follows:

```
objNewShortcut.Description = "Guess a Number Game"
objNewShortcut.Hotkey = "Ctrl+Alt+G"
```

The first of these two statements adds a description to the shortcut. Once created, this description can be viewed by moving the pointer over the shortcut's icon for a few moments. The second statement defines a keyboard keystroke sequence that, when executed, will activate the shortcut and thus open its associated Windows resources (that is, run your script). In this case, pressing the `Ctrl+Alt+G` keys at the same time will run the VBScript.

The fifth and final step in creating the shortcut is to save it using the `WshShortcut` object's `Save()` method, like this:

```
objNewShortcut.Save()
```

Let's put all five of these statements together to complete the script:

```
Set objWshShl = WScript.CreateObject("WScript.Shell")
strDesktopFolder = objWshShl.SpecialFolders("Desktop")
Set objNewShortcut = objWshShl.CreateShortcut(strDesktopFolder & _
    "\\GuessANumber.lnk")
objNewShortcut.TargetPath = "c:\GuessANumber.vbs"
objNewShortcut.Save()
```

Trick

It's just as easy to delete a shortcut using VBScript and the WSH as it is to create one. For example, create and run the following script to delete the shortcut the previous script created:

```
Set objWshShl = WScript.CreateObject("WScript.Shell")
strTargetFolder = objWshShl.SpecialFolders("Desktop")
Set objFsoObject = CreateObject("Scripting.FileSystemObject")
Set objNewShortcut = objFsoObject.GetFile(strTargetFolder & _
    "\\GuessANumber.lnk")
objNewShortcut.Delete
```

The first statement establishes an instance of the `WshShell` object. The second statement uses the `WshShell` object's `SpecialFolders` property to identify the location of the shortcut. The third statement creates an instance of the VBScript `FileSystemObject` object. The fourth statement uses the `FileSystemObject` object's `GetFile()` method to instantiate the `File` object and create a reference to the shortcut. The final statement deletes the shortcut using the `File` object's `Delete()` method.

Understanding How to Work with Special Folders

Windows operating systems use folders for a number of purposes. For example, folders are used to store system files. You also use folders to store your own personal files. As you just learned, the Windows desktop is a special folder. Windows supports a number of special folders:

- Desktop
- Programs
- Favorites
- Startup

By adding and removing shortcuts to and from Windows special folders, you can change their contents.

Using Shortcuts to Add Your Script to the Programs Menu and Apps Group

To work with the folders that make up the Programs menu, you need to create a reference to the Program's special folder. Once established, you can add shortcuts to the Programs menu as demonstrated in the following example:

```
Set objWshShl = WScript.CreateObject("WScript.Shell")
strTargetFolder = objWshShl.SpecialFolders("Programs")
Set objNewShortcut = objWshShl.CreateShortcut(strTargetFolder & _
    "\\GuessANumber.lnk")
objNewShortcut.TargetPath = "c:\GuessANumber.vbs"
objNewShortcut.Save
```

Figure 6.11 demonstrates how the Programs menu now looks with its new shortcut.

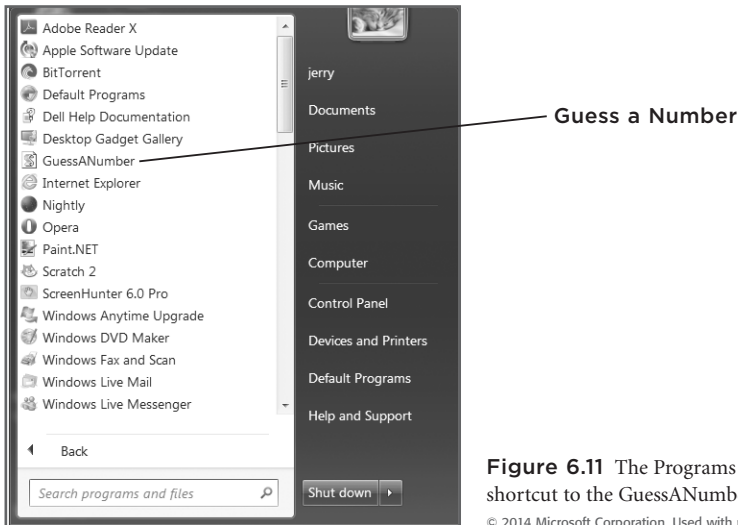


Figure 6.11 The Programs menu, with a shortcut to the GuessANumber.vbs game.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

If you run this example on a computer running Windows 8.1, it will add a shortcut for your script to the Apps group, as shown in Figure 6.12.

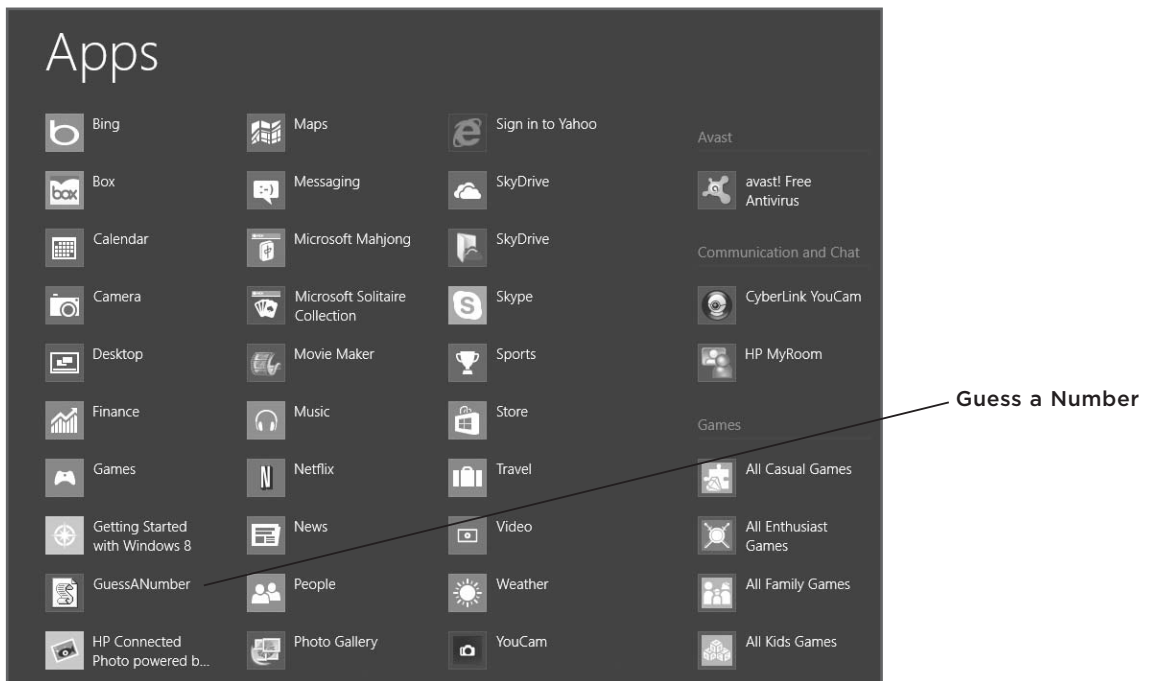


Figure 6.12 The Windows 8.1 Apps group, with a shortcut for the GuessANumber.vbs game.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

A Complete Shortcut Script

Now let's put together some of the shortcut examples you worked on previously to make a new script that creates shortcuts for `GuessANumber.vbs` on the Windows desktop and Programs menu.

```

'*****
'Script Name: ShortcutMaker.vbs
'Author: Jerry Ford
'Created: 02/20/14
'Description: This script creates shortcuts for the GuessANumber.vbs
'VBScript 'on the Windows desktop and Programs menu.
'*****

'Initialization Section
Option Explicit

Dim objWshShl, strTargetFolder, objDesktopShortcut, objProgramsShortcut
Dim strAppDataPath

'Establish an instance of the WshShell object
Set objWshShl = WScript.CreateObject("WScript.Shell")

'Create the desktop shortcut - Works on Windows 7 and Windows 8.1
strTargetFolder = objWshShl.SpecialFolders("Desktop")
Set objDesktopShortcut = objWshShl.CreateShortcut(strTargetFolder + _
    "\\GuessANumber.lnk")
objDesktopShortcut.TargetPath = "C:\\GuessANumber.vbs"
objDesktopShortcut.Description = "Guess a Number Game"
objDesktopShortcut.Hotkey = "Ctrl+Alt+G"
objDesktopShortcut.Save

'Create the Programs menu shortcut on Windows 7 and a Apps group
'shortcut on Windows 8.1
strTargetFolder = objWshShl.SpecialFolders("Programs")
Set objProgramsShortcut = objWshShl.CreateShortcut(strTargetFolder & _
    "\\GuessANumber.lnk")
objProgramsShortcut.TargetPath = "c:\\GuessANumber.vbs"
objProgramsShortcut.Save

```


I achieved a few economies of scale here. First, I had to instantiate the `WshShell` object only once. I also reused the `strTargetFolder` variable over and over again. However, I thought that it made the script more readable to assign a different variable to each special folder reference. Run this script on a computer running Windows 7 and you should see shortcuts for `GuessANumber.vbs` added to the Windows desktop and Programs menu. Run the script on a computer running Windows 8.1 and you should see a shortcut on the desktop and in the Apps group.

Summary

In this chapter, you learned about loops and how to apply them to your VBScripts. You developed your understanding of this fundamental programming concept through the creation of the Guess a Number game. You also learned how to programmatically work with Windows shortcuts, including creating shortcuts for your scripts. You also learned how to configure a number of Windows features, including the Windows desktop and Programs menu.

Challenges

1. Modify the Guess a Number Game by providing the players with better hints. For example, if a user's guess is within 20 numbers of the answer, tell the player that he is getting warm. As the player gets even closer to the correct guess, tell him that he is getting very hot.
2. Change the Guess a Number game to increase the range of numbers from one to 100 to one to 1,000.
3. Rewrite the Guess a Number game so it uses a `Do...While` statement in place of a `Do...Until` statement.
4. Using `ShortcutMaker.vbs` as a starting point, write a new script that creates one or more shortcuts for your favorite VBScript game. Alternatively, if you keep all your VBScripts in one location, create a shortcut to that folder.

7

Using Procedures to Organize Scripts

By now you've seen and worked on a number of VBScript projects, and all of these scripts have been organized the same way. First, you set up script initialization processes (defining variables, constants, objects, and so on). Then you sequentially wrote the rest of the script as one big collection of statements. You then used the `If` and `Select Case` statements to organize your scripts. Finally, by embedding statements within one another, you further refined your scripts' organization. In this chapter, you will learn how to further improve the organization of your VBScripts using procedures. Specifically, you will learn the following:

- How to create your own customized functions
- How to create reusable collections of statements using subroutines
- How to break down scripts into modules of code to make them easier to manage
- How to control variable scope within your scripts using procedures

Project Preview: The BlackJack Lite Game

In this chapter, you will create a game called BlackJack Lite. This game is based on the classic blackjack game played in casinos around the world. In this game, both the player and the computer are dealt a single card, face up. The object of the game is to try to get as close as possible to a value of 21 without going over. The player can ask for as many extra cards (hits) as desired and can stop (stick) at any time. If the player goes over 21, he busts. Otherwise, the computer plays its hand, stopping only after either reaching a total of 17 or more or busting. Figures 7.1 through 7.4 demonstrate the game in action as played on a computer running Windows 8.1.

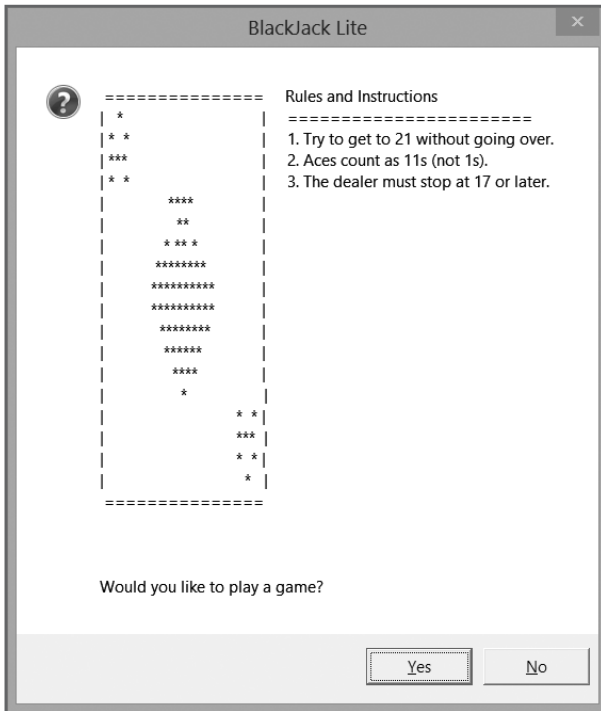


Figure 7.1 The game's splash screen invites the user to play a game of BlackJack Lite. © 2014 Cengage Learning.

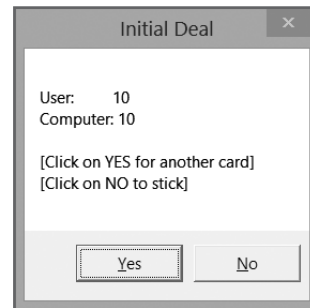


Figure 7.2 If the user accepts the offer to play, the initial hands are dealt. © 2014 Cengage Learning.



Figure 7.3 The user plays until either busting or holding. © 2014 Cengage Learning.



Figure 7.4 The computer then plays and the results of the game are shown. © 2014 Cengage Learning.

By the time you've worked your way through this chapter and completed the BlackJack Lite game, you will have gained a solid understanding of how to use procedures. You will be able to improve the overall organization and functionality of your VBScripts and tackle even more challenging projects.

Improving Script Design with Procedures

VBScript procedures improve the overall organization and readability of scripts, giving you a way to group related statements and execute them as a unit. Once written, a VBScript procedure can be called on from any location in your script and can be executed over and over again as needed. This enables you to create scripts that are smaller and easier to maintain.

VBScript provides support for two different types of procedures:

- **Subroutine.** This is a VBScript procedure that executes a set of statements without returning a result.
- **Function.** This is a VBScript procedure that executes a set of statements and, optionally, returns a result to the statement that called it.

Definition

A *procedure* is simply a collection of VBScript statements that, when called, are executed as a unit.

Trick

I recommend using procedures as the primary organization tool for all VBScripts. By organizing a script into procedures, you break it down into a collection of units. This allows you to separate processes from one another, making it easier to develop scripts in a modular fashion, one component at a time.

Introducing Subroutines

The VBScript Sub procedure is used to create subroutines. Subroutines are great for grouping together statements that perform a common task from which a result is not required. When called, subroutines execute their statements and then return processing control back to the calling statement.

The syntax for this type of procedure is as follows:

```
[Public | Private] Sub name [(arglist)]  
    statements  
End Sub
```

`Private` is an optional keyword that specifies the subroutine cannot be called by other procedures within the script, thus limiting the ability to reference it. `Public` is an optional keyword that specifies the subroutine can be called by other procedures within the script. `name` is the name assigned to the subroutine. As with variables, a subroutine's name must be unique within the script that defines it. `arglist` represents a list of one or more comma-separated arguments that can be passed to the subroutine for processing, and `statements` represents the statements that make up the subroutine.

For example, the next subroutine is called `DisplaySplashScreen()`. It does not accept any arguments and it does not return anything back to the VBScript statement that calls it. What it does is display a script's splash screen any time it is called.

```
Sub DisplaySplashScreen()  
    MsgBox "Thank you for playing the game. © Jerry Ford 2014." & _  
        vbCrLf & vbCrLf & "Please play again soon!", 4144, "Test Game"  
End Sub
```

You can execute this subroutine by calling it from anywhere within your script using the following statement:

```
DisplaySplashScreen()
```

The following example is a rewrite of the previous subroutine, only this time the subroutine accepts an argument. The argument passed to the subroutine will be a message. Using a subroutine in this manner, you can develop scripts that display all their pop-up dialog boxes using one subroutine.

```
Sub DisplaySplashScreen(strMessage)  
    MsgBox strMessage, 4144, "Test Game"  
End Sub
```

You can call this subroutine from anywhere within your script like this:

```
DisplaySplashScreen("Thank you for playing the game. © Jerry Ford " &  
    "2014." & vbCrLf & vbCrLf & "Please play again soon!")
```

Creating Custom Functions

Functions are almost exactly like subroutines. Functions can do anything that a subroutine can do. In addition, a function can return a result to the statement that called it. As a result (to keep things simple), I usually use functions, rather than subroutines, within my VBScripts.

The syntax for a function is as follows:

```
[Public | Private] Function name [(arglist)]  
    statements  
End Function
```

`Private` is an optional keyword that specifies that the function cannot be called by other procedures within the script, thus limiting the ability to reference it. `Public` is an optional keyword that specifies that the function can be called by other procedures within the script. *name* is the name assigned to the function. As with variables, a function's name must be unique within the script that defines it. *arglist* represents a list of one or more comma-separated arguments that can be passed to the function for processing, and *statements* represents the statements that make up the function.

Let's look at an example of a function that does not return a result to its calling statement:

```
Function DisplaySplashScreen()  
    MsgBox "Thank you for playing the game. © Jerry Ford 2014." & _  
        vbCrLf & vbCrLf & "Please play again soon!", 4144, "Test Game"  
End Function
```

As written, this function performs the exact same operation as the subroutine you saw previously. This function can be called from anywhere in your script using the following statement:

```
DisplaySplashScreen()
```

As with subroutines, you may pass any number of arguments to your functions, as long as commas separate the arguments, like this:

```
Function DisplaySplashScreen(strMessage)
    MsgBox strMessage, 4144, "Test Game"
End Function
```

Once again, this function is no different from the corresponding subroutine example you just saw, and can be called as follows:

```
DisplaySplashScreen("Thank you for playing the game. © Jerry Ford " &_
    "2014." & vbCrLf & vbCrLf & "Please play again soon.")
```

Functions also can be set up to return a result to their calling statement. This is achieved by creating a variable within the function that has the same name as the function, and by setting the variable equal to the result that you want the function to return.

Again, this technique can best be demonstrated with an example:

```
strPlayersName = GetPlayersName()
MsgBox "Greetings " & strPlayersName

Function GetPlayersName()
    GetPlayersName = InputBox("What is your first name?")
End Function
```

The first statement calls a function named `GetPlayersName()`. The second statement displays the results returned by the function and stored in the variable called `PlayersName`. The next three lines are the actual function, which consists of a single statement that collects the player's name and assigns it to a variable named `GetPlayersName` so that it can be passed back to the calling statement.

Another way to call a function is to reference it as part of another VBScript statement, like this:

```
MsgBox "Greeting " & GetPlayersName()
```

Improving Script Manageability

As I said, by organizing your VBScripts into procedures, you make them more manageable, allowing you to create larger and more complex scripts without adding mounds of complexity. As an example, say you're developing a game that performs the following five major activities:

- It initializes variables, constants, and objects used by the script.
- It asks the player whether he wants to play the game.

- It collects the player's name.
- It displays a story, substituting the player's name at predetermined locations within the story.
- It displays a closing dialog box, inviting the player to play again on another day.

One way to design your script would be to first define the variables, constants, and object references, and then create a series of functions and subroutine calls from the script's main processing section. The rest of the script would then consist of individual functions and subroutines, each of which would be designed to perform one of the activities outlined in the previous list.

In the Real World

One sign of a world-class programmer is the path that he leaves behind—in other words, the professional way in which the programmer organizes and documents his scripts. One organizational technique used by experienced programmers is to group all functions and subroutines together in one place, apart from the initialization and main processing sections of the script. This makes them easy to locate and maintain. Usually, you'll find a script's functions and subroutines located at the bottom of the script. I suggest you modify your script template to include a "procedure" section for this purpose.

Writing Reusable Code

One of the biggest advantages of using functions and subroutines is the capability to create reusable code within your VBScripts. Any time you find yourself needing to perform the same task over and over in a script—such as displaying messages in pop-up dialog boxes or retrieving random numbers—consider creating a function or subroutine. Then, by using a single statement to call the appropriate procedure, you can reuse the statements located within the procedure over and over again.

Functions and subroutines help make for smaller scripts. They also make script maintenance and enhancement much easier and quicker. For example, it's a lot easier to change one line of code located in a procedure than it is to make that same change in numerous places throughout a script.

The Guess a Number Game Revisited

So far, you have seen examples of small pieces of code that work with functions and subroutines. Now let's take a look at how to apply procedures to a larger script. To begin, go back and review the Guess a Number game in Chapter 6, "Processing Collections of Data." This script, like all other scripts in this book prior to this chapter, was written without the use of procedures.

I deliberately avoided using procedures in the script from Chapter 6, so I had to use other techniques to organize the script's programming logic. What I chose to do was to put everything in the script's main processing section as follows:

- I added statements to generate a random number.
- I added a `Do...Until` loop to control the game's execution.

- I embedded an If statement within the Do...Until loop to ensure the player typed a number.
- I embedded a second If statement within the previous If statement to make sure the data the player typed was numeric.
- I embedded three more If statements within the previous If statement to determine whether the player's guess was low, high, or correct.

As you can see, I had to embed a lot of statements within one another to organize the script into a workable game. As the script itself was not exceptionally large, this was a manageable task. However, had the script been much larger or more complex, it would have been difficult to keep track of all the embedded logic.

Now that you understand what procedures are and what they're used for, let's take a moment and go back and redesign the Guess a Number game using them. One way of doing this is as follows:

```

'*****
'Script Name: GuessANumber-2.vbs
'Author: Jerry Ford
'Created: 02/01/14
'Description: This script plays a number-guessing game with the user
'*****

'Initialization Section
Option Explicit

Const cGreetingMsg = "Pick a number between 1 - 100"
Dim intUserNumber, intRandomNo, strOkToEnd, intNoGuesses, strBadData

strOkToEnd = "no"
intNoGuesses = 0

'Main Processing Section

RandomNumber()      'Get the game's random number
PlayTheGame()       'Start the game
WScript.Quit()      'End the game

'Procedure Section

'Generate the game's random number

```



```

Function RandomNumber()
    'Generate a random number
    Randomize
    intRandomNo = FormatNumber(Int((100 * Rnd) + 1))
End Function

'Set up a Do...Until loop to control the execution of the game
Function PlayTheGame()
    'Loop until the user either guesses correctly or clicks on Cancel
    Do Until strOkToEnd = "yes"

        'Prompt user to pick a number
        intUserNumber = InputBox("Type your guess:", cGreetingMsg)
        intNoGuesses = intNoGuesses + 1
        strBadData = "no"

        'Go see if there is anything wrong with the player's input
        ValidateInput()

        If strOkToEnd <> "yes" Then      'The player typed in something
            If strBadData <> "yes" Then 'The player typed in a number
                TestAnswer()          'Let's see how good the player's guess was
            End If
        End If

    Loop
End Function

'Determine if there are any problems with the data entered by the player
Function ValidateInput()
    'See if the player provided an answer
    If Len(intUserNumber) = 0 Then
        MsgBox "You either failed to type a value or you clicked on " & _
            "Cancel. Please play again soon!", , cGreetingMsg
        strOkToEnd = "yes"
    Else
        'Make sure that the player typed a number
        If IsNumeric(intUserNumber) = False Then
            MsgBox "Sorry. You did not enter a number. Try again.", , _

```

```
        cGreetingMsg
        strBadData = "yes"
    End If
End If
End Function

'Determine if the player's guess is too low, too high, or just right
Function TestAnswer()
    'Test to see if the user's guess was correct
    If FormatNumber(intUserNumber) = intRandomNo Then
        MsgBox "Congratulations! You guessed it. The number was " & _
            intUserNumber & "." & vbCrLf & vbCrLf & "You guessed it " & _
            "in " & intNoGuesses & " guesses.", ,cGreetingMsg
        strOkToEnd = "yes"
    End If

    'Test to see if the user's guess was too low
    If FormatNumber(intUserNumber) < intRandomNo Then
        MsgBox "Your guess was too low. Try again", ,cGreetingMsg
        strOkToEnd = "no"
    End If

    'Test to see if the user's guess was too high
    If FormatNumber(intUserNumber) > intRandomNo Then
        MsgBox "Your guess was too high. Try again", ,cGreetingMsg
        strOkToEnd = "no"
    End If
End Function
```

As you can see, the script's initialization section remained unchanged except for the addition of one more variable, which is to indicate that the player has typed an invalid character. However, the main processing section is now quite different. Instead of having all the script's statements embedded within it, this section now drives the script by maintaining high-level control over a collection of functions, each of which performs a specific process for the script.

The main processing section now does three things:

- It calls a function that gets the game's random number (`RandomNumber()`).
- It calls a function that controls the play of the game (`PlayTheGame()`).
- It ends the game by executing the `WScript` object's `Quit()` method.

The `RandomNumber()` function generates the random number used by the game. The `PlayTheGame()` function controls play of the game itself. Instead of making this a really large function, I simplified it a bit by removing and modifying the two `If` statements that perform input validation and placing them within their own function called `ValidInput()`. Likewise, I moved and modified the three `If` statements that determine whether the player's guess was low, high, or correct to their own function called `TestAnswer()`. The only other modification made to the script was the addition of the following statements in the `PlayTheGame()` function. These statements were needed to test the values of variables manipulated in the `ValidateInput()` and `TestAnswer()` functions:

```
If strOkToEnd <> "yes" Then      'The player typed in something
    If strBadData <> "yes" Then  'The player typed in a number
        TestAnswer()           'Let's see how good the player's guess was
    End If
End If
```

Working with Built-in VBScript Functions

VBScript provides a large collection of built-in functions that you can add to your scripts to save yourself time and effort. Obviously, leveraging the convenience and power of these built-in VBScript functions is a smart thing to do.

In fact, VBScript's built-in functions are so essential to VBScript development that it's difficult to write a script of any complexity without using them. I have already demonstrated this fact many times throughout the book. A complete list of all VBScript built-in functions appears in Appendix D, "Built-in VBScript Functions."

Limiting Variable Scope with Procedures

You have seen and worked with variables throughout this book. Thus far, all the variables that you have worked with have had a *global* or *script-level* scope, meaning they could be accessed from any point within the script. Any variable that is defined outside of a VBScript procedure (that is, function or subroutine) is global in scope.

In contrast, any variable defined within a procedure is *local in scope*, meaning it exists and can only be referenced within the procedure that defines it. The best way to demonstrate the concept of global and local variable scope is in an example. The following script creates two variables, one at the beginning of the script and the other within a function:

```
Option Explicit
```

```
Dim intFirstRandomNumber
```

```
intFirstRandomNumber = GetRandomNumber()
MsgBox "The first random number is " & intFirstRandomNumber

GenerateLocalizedVariable()
MsgBox "The second random number is " & intSecondRandomNumber

WScript.Quit()

Function GenerateLocalizedVariable()
    Dim intSecondRandomNumber
    intSecondRandomNumber = GetRandomNumber()

    MsgBox "The second random number is " & intSecondRandomNumber
End Function

Function GetRandomNumber()
    'Generate a random number between 1 and 10
    Randomize
    GetRandomNumber = FormatNumber(Int((10 * Rnd) + 1))
End Function
```

When you run this script, the first variable is defined at the beginning of the script, making it a global variable. The value of the variable is then immediately displayed. The second variable is defined within a function named `GenerateLocalizedVariable()`. As a result, the variable can be referenced only within this function, as proven when the function's `MsgBox()` statement displays its value. When the `GenerateLocalizedVariable()` function completes its execution, processing control returns to the statement that called the function. This statement is immediately followed by another `MsgBox()` statement, which attempts to display the value of the variable defined in the `GenerateLocalizedVariable()` function. However, this variable was destroyed as soon as that function ended, so instead of seeing the variable's value, an error message is displayed. Figure 7.5 shows the error message that is displayed if this script is run on Windows 7.

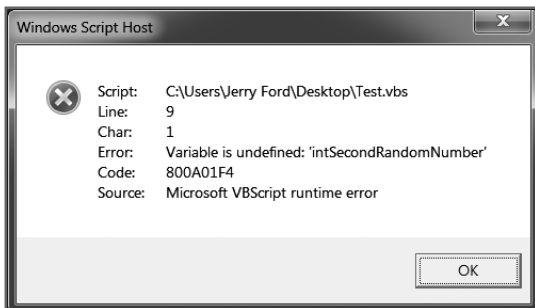


Figure 7.5 Attempting to access a localized variable outside the procedure that defined it results in an error.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

Back to the BlackJack Lite Game

Now let's return to the development of the BlackJack Lite game. In this game, you'll develop your own version of the casino game blackjack. BlackJack Lite is a card game that pits the player against the computer. The object of the game is to come as close to 21 as possible without going over and to beat the computer's hand. The computer, like a real casino blackjack dealer, waits for the player to finish before playing. The computer must then take hits until its hand busts (goes over 21) or reaches at least 17, at which time it must stop.

Designing the Game

The BlackJack Lite game is more complex than the other VBScripts that you've seen so far in this book. The game itself has a large number of different functions to perform. For example, the initial hand must be dealt for both the player and the computer. Then the game has to facilitate the dealing of cards to the player and later to the computer. In addition, numerous smaller processes must occur along the way.

Because this script is rather complex, I've decided to organize it into procedures. Each procedure will be assigned a specific activity to perform. As part of my preparation for the design of the game, I have drawn a high-level flowchart of the game's overall structure and processing logic. This flowchart is shown in Figure 7.6.

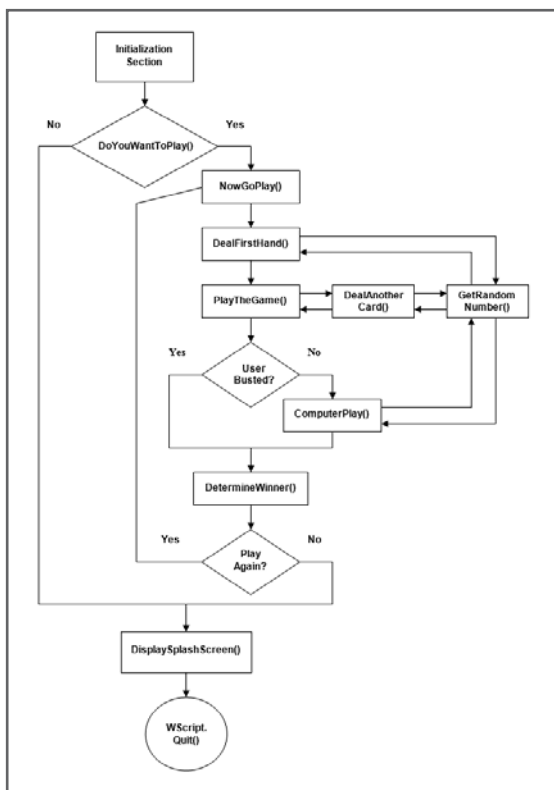


Figure 7.6 A flowchart outlining the overall design and execution flow of the BlackJack Lite game. © 2014 Cengage Learning.

The script consists of nine functions. These functions and their purposes are as follows:

- **DoYouWantToPlay()**. This displays the game's splash screen and invites the user to play the game.
- **NowGoPlay()**. This controls the overall execution of the game, calling upon other procedures as required.
- **DealFirstHand()**. This presents the initial cards for both the player and the computer.
- **PlayTheGame()**. This asks the player whether he would like another card and determines when the player has either busted or decided to hold.
- **DealAnotherCard()**. This retrieves another card for the player's hand.
- **GetRandomNumber()**. This function is called by several other functions in the script. It returns a random number between 1 and 13, representing the value of a playing card.
- **ComputerPlay()**. This plays the computer's hand, taking hits until either the computer's hand is busted or is greater than 17.
- **DetermineWinner()**. This compares the player's hand to the computer's hand to determine who has won. It then offers to let the player play another hand. If the player accepts, the `NowGoPlay()` function is called, starting a new hand.
- **DisplaySplashScreen()**. This displays information about the game and its author, and invites the player to return and play again before finally ending the game.

Setting Up the Initialization Section

You begin the development of the BlackJack Lite game the same way that you've begun all your other games: by first creating a new file and adding in your VBScript template, and then setting up the variables, constants, and object references in the script's initialization section.

```

'*****
'Script Name: BlackJack.vbs
'Author: Jerry Ford
'Created: 02/03/14
'Description: This script creates a scaled-down version of the casino
'blackjack card game
'*****

'Initialization Section

Option Explicit

Dim intPlayGame, strCardImage, intUserCard, intComputerCard, intAnotherCard
Dim intUserNextCard, strUserDone, intNewComputerCard, intPlayAgain
Dim strUserBusted, strTextMsg

```

```
strUserDone = "False"  
strUserBusted = "False"
```

This game is fairly lengthy and requires a number of variables:

- **intPlayGame.** This stores the player's reply when asked if he wants to play a game.
- **strCardImage.** This stores the message displayed in the game's initial pop-up dialog box.
- **intUserCard.** This stores the total value of the cards dealt to the user.
- **intComputerCard.** This stores the total value of the cards dealt to the computer.
- **intAnotherCard.** This stores the player's reply to the question of whether he wants another card.
- **intUserNextCard.** This stores the value returned by the function that retrieves a random number and is later added to the value of the `intUserCard` variable.
- **strUserDone.** This stores a value indicating whether the player is ready to hold.
- **intNewComputerCard.** This stores the value returned by the function that retrieves a random number and is later added to the value of the `intComputerCard` variable.
- **intPlayAgain.** This stores the player's reply when asked whether he wants to play another game.
- **strUserBusted.** This stores a value indicating whether the player has busted.
- **strTextMsg.** This stores text to appear in pop-up dialog boxes displayed by the game.

Developing the Logic for the Main Processing Section

The script's main processing section is very small and consists only of a call to the `DoYouWantToPlay()` function to determine whether the player wants to play the game, followed by the `Select Case` statement, which determines the player's reply.

```
'Main Processing Section  
  
'Ask the user if he or she wants to play  
intPlayGame = DoYouWantToPlay()  
  
Select Case intPlayGame  
    Case 6 'User clicked on Yes  
        NowGoPlay()  
    Case 7 'User clicked on No  
        DisplaySplashScreen()  
End Select
```

If the player clicks the Yes button, then the `NowGoPlay()` function is called. Otherwise, the `DisplaySplashScreen()` function is called. This function displays a pop-up dialog box providing information about the game and then terminates the game's execution.

Creating the DoYouWantToPlay() Function

This function displays the game's initial pop-up dialog box and invites the player to play a game of BlackJack Lite. Much of the text displayed in this pop-up dialog box is dedicated to creating an image depicting the ace of spades. Also included on this pop-up dialog box is a brief set of instructions.

```
function DoYouWantToPlay()
    strCardImage = " =====" & Space(5) & "Rules and " & _
        "Instructions" & vbCrLf & _
        "| * " & vbTab & Space(22) & "|" & Space(5) & "===== " & _
        "===== " & vbCrLf & _
        "| * * " & vbTab & Space(22) & "|" & Space(5) & "1. Try to get" & _
        " to 21 without going over." & vbCrLf & _
        "| *** " & vbTab & Space(22) & "|" & Space(5) & "2. Aces count" & _
        " as 11s (not 1s)." & vbCrLf & _
        "| * * " & vbTab & Space(22) & "|" & Space(5) & "3. The dealer " & _
        "must stop at 17 or later." & vbCrLf & _
        "|" & Space(15) & "****" & vbTab & Space(6) & "|" & vbCrLf & _
        "|" & Space(17) & "***" & vbTab & Space(6) & "|" & vbCrLf & _
        "|" & Space(14) & "* ** *" & vbTab & Space(6) & "|" & vbCrLf & _
        "|" & Space(12) & "*****" & Space(13) & "|" & vbCrLf & _
        "|" & Space(11) & "*****" & Space(11) & "|" & vbCrLf & _
        "|" & Space(11) & "*****" & Space(11) & "|" & vbCrLf & _
        "|" & Space(13) & "*****" & Space(12) & "|" & vbCrLf & _
        "|" & Space(14) & "*****" & vbTab & Space(6) & "|" & vbCrLf & _
        "|" & Space(16) & "*****" & vbTab & Space(6) & "|" & vbCrLf & _
        "|" & Space(17) & " * " & Space(18) & "|" & vbCrLf & _
        "|" & vbCrLf & vbCrLf & "* * |" & vbCrLf & _
        "|" & vbCrLf & vbCrLf & "*** |" & vbCrLf & _
        "|" & vbCrLf & vbCrLf & "* * |" & vbCrLf & _
        "|" & vbCrLf & vbCrLf & Space(1) & " * |" & vbCrLf & _
        " =====" & vbCrLf & vbCrLf & vbCrLf & vbCrLf & _
        "Would you like to play a game?"

    DoYouWantToPlay = MsgBox(strCardImage, 36, "BlackJack Lite")
End Function
```

If the player clicks the Yes button, the value of DoYouWantToPlay() is set to 6. This value will later be tested in the script's main processing section to determine whether the game should continue.

Creating the NowGoPlay() Function

The `NowGoPlay()` function is called when the player clicks the Yes button on the script's initial pop-up dialog box, indicating that he wants to play the game.

```
function NowGoPlay()  
    DealFirstHand()  
    PlayTheGame()  
    If strUserBusted = "False" Then  
        ComputerPlay()  
    End If  
    DetermineWinner()  
End Function
```

This function controls the actual play of the game. It is made up of calls to several other functions. First it calls the `DealFirstHand()` function, which deals both the user and the computer their initial cards. Next it calls the `PlayTheGame()` function, which allows the user to continue to take hits or hold, and determines whether the player busted. The `NowGoPlay()` function then checks the value of the `strUserBusted` variable to see whether the game should continue. If the user decides to hold, then the `ComputerPlay()` function is called so the computer's (or dealer's) hand can finish being dealt. Regardless of whether the user busts or the `ComputerPlay()` function is called, eventually control returns to the `NowGoPlay()` function and the `DetermineWinner()` function is called. This function determines the winner of the hand and gives the player an opportunity to play another hand.

Creating the DealFirstHand() Function

The `DealFirstHand()` function makes two calls to the `GetRandomNumber()` function to deal both the player's and the computer's initial cards:

```
function DealFirstHand()  
    intUserCard = GetRandomNumber()  
    intComputerCard = GetRandomNumber()  
End Function
```

Creating the PlayTheGame() Function

The `PlayTheGame()` function, shown next, sets up a `Do...Until` loop that executes until the player either busts or decides to hold. The first statement in the loop prompts the player to decide whether he wants another card. If the player clicks Yes, the `DealAnotherCard()` function is called. The `PlayTheGame()` function then checks to see whether the player has busted, setting the value of the `strUserBusted` and `strUserDone` variables if appropriate (`True`). If the player decides to hold and clicks the No button, the value of `strUserDone` is also set to `True`.

```
function PlayTheGame()
  Do Until strUserDone = "True"

  intAnotherCard = MsgBox("User: " & Space(8) & intUserCard & vbCrLf & _
    "Computer: " & intComputerCard & vbCrLf & vbCrLf & _
    "[Click on YES for another card]" & vbCrLf & _
    "[Click on NO to stick]", 4, "Initial Deal")

  Select Case intAnotherCard
    Case 6 'User clicked on Yes
      MsgBox "You decided to take a hit."
      DealAnotherCard()
    Case 7 'User clicked on No
      strUserDone = "True"
  End Select

  If intUserCard > 21 then
    strUserBusted = "True"
    strUserDone = "True"
  End If

  Loop
End Function
```

Creating the GetRandomNumber() Function

When called, the `GetRandomNumber()` function generates a random number from 1 to 13. If the randomly generated number is equal to 1 (equivalent to an ace) then a value of 11 is returned. If the randomly generated number is greater than 10 (equivalent to a jack, queen, or king) then a value of 10 is returned. Any other random value is returned as is.

```
function GetRandomNumber()
  Randomize
  GetRandomNumber = Round(FormatNumber(Int((13 * Rnd) + 1)))
  If GetRandomNumber = 1 then GetRandomNumber = 11
  If GetRandomNumber > 10 then GetRandomNumber = 10
End Function
```

Creating the DealAnotherCard() Function

The DealAnotherCard() function is called from the PlayTheGame() function when the player elects to take a hit (asks for another card). This function consists of two statements. The first statement assigns the card number returned to it from the GetRandomNumber() function to a variable named intUserNextCard. The second statement tallies the player's hand by adding the value of the new card to the cards already in the player's hand.

```
function DealAnotherCard()  
    intUserNextCard = GetRandomNumber()  
    intUserCard = intUserCard + intUserNextCard  
End Function
```

Creating the ComputerPlay() Function

The ComputerPlay() function, shown here, is responsible for dealing the computer's hand. It uses a Do...While loop to continue dealing the computer's hand until either the computer's hand exceeds a total of 17 but remains under 21 or it busts.

```
function ComputerPlay()  
    Do While intComputerCard < 17  
        intNewComputerCard = GetRandomNumber()  
        intComputerCard = intComputerCard + intNewComputerCard  
    Loop  
End Function
```

Inside the Do...While loop are two statements. The first statement deals the computer a new card by calling the GetRandomNumber() function. The second statement uses the value returned by the GetRandomNumber() function to update the computer's hand (that is, its total).

Creating the DetermineWinner() Function

The DetermineWinner() function, shown here, checks to see whether the value of strUserBusted is set to True. It also checks to see whether the computer has busted by checking to see whether the value of intComputerCard is greater than 21. If either of these conditions is true, the script assigns an appropriate text message to the strTextMsg variable. This variable is used as input in an InputBox() pop-up dialog box that shows the player the results of the game. If neither of these conditions is true, the DetermineWinner() function performs three tests to determine whether the player won, the computer won, or there was a tie. The function then sets the value of strTextMsg accordingly.

```
function DetermineWinner()  
    If strUserBusted = "True" Then  
        strTextMsg = "The user has busted!"  
    Else
```

```

If intComputerCard > 21 then
    strTextMsg = "The Computer has busted!"
Else
    If intUserCard > intComputerCard Then strTextMsg = "The user wins!"
    If intUserCard = intComputerCard Then strTextMsg = "Push (i.e. Tie)!"
    If intUserCard < intComputerCard Then strTextMsg = "The " & _
        "Computer wins!"
    End If
End If

intPlayAgain = MsgBox(strTextMsg & vbCrLf & vbCrLf & "User: " & _
    Space(8) & intUserCard & vbCrLf & "Computer: " & intComputerCard & _
    vbCrLf & vbCrLf & vbCrLf & _
    "Would you like to play another game?", 4, "Initial Deal")

If intPlayAgain = 6 Then
    strUserBusted = "False"
    strUserDone = "False"
    NowGoPlay()
End If

DisplaySplashScreen()
End Function

```

Finally, the game's results are displayed by showing the value of `strTextMsg` and the value of the player's and the computer's final hands. The pop-up dialog box also asks the player whether he would like to play again. If the player clicks the Yes button, the `NowGoPlay()` function is called and the game starts over again. Otherwise, the `DisplaySplashScreen()` function is called and the game ends.

Creating the `DisplaySplashScreen()` Function

This final function displays the game's splash screen, providing a little information about the game and its creator and inviting the player to return and play the game again another time.

```

function DisplaySplashScreen()
    MsgBox "Thank you for playing BlackJack Lite © Jerry Ford 2014." & _
        vbCrLf & vbCrLf & "Please play again soon!", 4144, "BlackJack Lite"
    WScript.Quit()
End Function

```

After displaying the splash screen in a pop-up dialog box, the `DisplaySplashScreen()` function ends the game by executing the `WScript.Quit()` method.

The Final Result

Take a few minutes to double-check all your work and then give this game a whirl. This is a pretty big script, so you may have to fix a few syntax errors introduced by typos you may have made when keying in the script. Once everything is working correctly, you should have a really cool game to share with—and impress—all your friends!

Summary

In this chapter, you learned how to use procedures to streamline the organization of your VBScripts, enabling you to develop larger and more complex scripts and, of course, games. In addition, you learned how to create reusable units of code, enabling you to make your scripts smaller and easier to manage. Finally, you learned how to control variable scope by localizing variables within procedures.

Challenges

1. Give the BlackJack Lite game's splash screen a more polished look by providing additional information in the "Rules and Instructions" section of the dialog box.
2. Improve the BlackJack Lite game by adding logic to include the selection of the card's suit (club, heart, spade, or diamond).
3. Once you have modified the BlackJack Lite game to assign cards that include both the card's suit and number, add logic to ensure that the same card is not used twice in the same hand.
4. Add scorekeeping logic to the BlackJack Lite game and display the number of won and lost hands at the end of each game.

III

Advanced Topics

Chapter 8: Storing and Retrieving Data

Chapter 9: Handling Script Errors

**Chapter 10: Using the Windows Registry
to Configure Script Settings**

**Chapter 11: Working with Built-in VBScript
Objects**

**Chapter 12: Combining Different Scripting
Languages**

**Chapter 13: Working with the Windows
Management Instrumentation**

Chapter 14: Adding a GUI to Your Scripts

This page intentionally left blank

8

Storing and Retrieving Data

Now that you've learned the basics of VBScript programming using the WSH, it's time to tackle more advanced topics. In this chapter, you'll learn how to work with and administer Windows files and folders, including storing data in reports and creating log files. You'll see how to open and programmatically read the contents of text files to process script input. You'll learn how to retrieve script configuration settings stored in external files and then use this information to control the way your scripts execute. Finally, I'll show you how to automate file and folder management by using VBScript to copy, move, and delete individual and groups of files and folders. Specifically, you will learn the following:

- How to create and write data to text files
- How to open and process data stored in text files
- How to copy, move, and delete files and folders
- How to retrieve script configuration settings from external files

Project Preview: The Lucky Lottery Number Picker

This chapter shows you how to create the Lucky Lottery Number Picker game, which randomly generates lottery ticket numbers. The player needs only to specify how many lottery tickets he plans to purchase; the game then generates the appropriate numbers. By default, the game assumes that it should generate six numbers for each lottery ticket the player wants to purchase. However, by editing an external configuration file that stores the game's execution settings, the player can modify the game to generate any amount of numbers per play.

Figures 8.1 through 8.3 show the Lucky Lottery Number Picker in action on a computer running Windows 7.

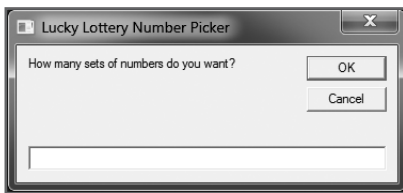


Figure 8.1 The game begins by asking the player how many different sets of lottery numbers should be generated. © 2014 Cengage Learning.

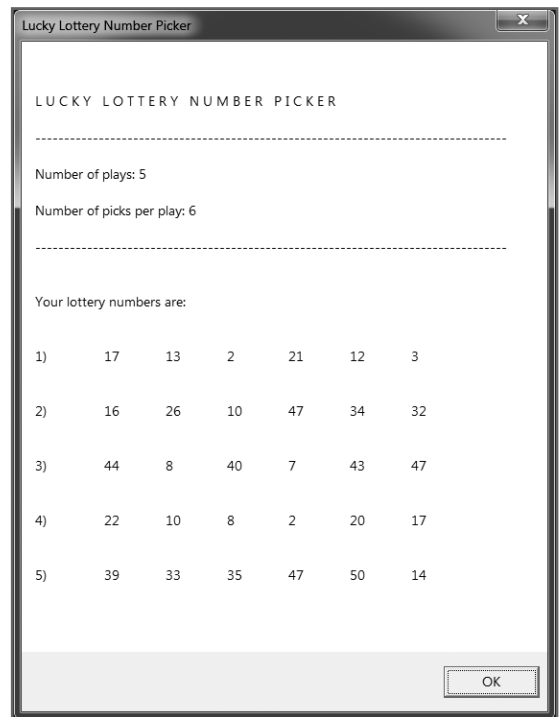


Figure 8.2 By default, the game displays configuration information at the top of its output followed by the lottery numbers. © 2014 Cengage Learning.

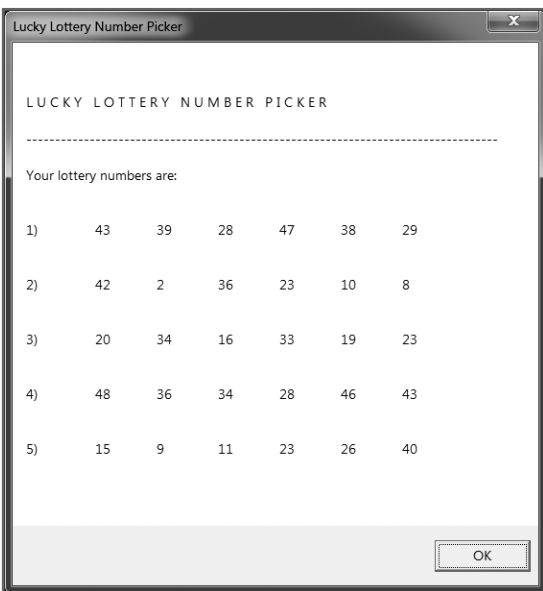


Figure 8.3 By changing script configuration settings stored in an external configuration file, the player can tell the script to provide only summary level information. © 2014 Cengage Learning.

By the time you've completed this chapter and created the Lucky Lottery Number Picker game, you will have mastered the building blocks required to work with and administer Windows files and folders. By learning how to store script configuration settings in external files, you'll also learn how to make your VBScripts easier to control and modify.

Working with the Windows File System

The WSH core object model provides the capability to interact with all sorts of Windows resources, such as the Windows desktop and Registry. However, it fails to provide any access to the Windows file system, so you cannot use it to access local disk drives or to work with files and folders. Instead of providing this functionality as part of the WSH core object model, Microsoft chose to implement it via the `FileSystemObject` object, which is one of VBScript's run-time objects. Refer to Table 3.4 in Chapter 3, "VBScript Basics," for a complete listing of VBScript's run-time objects.

The `FileSystemObject` object is VBScript's primary run-time object. All other run-time objects, except for the `Dictionary` object, are derived from it. To use the `FileSystemObject` object, you must instantiate it as shown here:

```
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
```

The first step in setting up an instance of the `FileSystemObject` object is to use the `Set` statement to associate a variable with it. This is accomplished by using the `WScript` object's `CreateObject()` method and specifying the `FileSystemObject` object as `Scripting.FileSystemObject`. Once instantiated, you can interact with the `FileSystemObject` object by referencing the variable that has been set up, thus providing access to all `FileSystemObject` object properties and methods.

To jump-start your understanding of the `FileSystemObject` object and how to use it, let's begin with an example. In this example, a VBScript is created that uses the `FileSystemObject` object to retrieve and display the properties associated with a file named `Sample.txt`. The script begins by instantiating the `FileSystemObject` object and associating it with a variable named `objFso`. Next, the `FileSystemObject` object's `GetFile()` method retrieves a reference to the `File` object that specifically refers to `Sample.txt`, which is located in the `C:\Temp` folder.

The main processing of the script then makes a series of procedure calls. The `CreateDisplayString()` function uses several `File` object properties to collect information about the `Sample.txt` file. The next two functions display the information that has been collected about the file and then terminate the script's execution.

```
'*****
'Script Name: ExtractFileProperties.vbs
'Author:      Jerry Ford
'Created:     02/10/14
'Description: This script demonstrates how to retrieve file information
'*****

'Initialization Section
Option Explicit
```

```
On Error Resume Next
```

```
Dim objFso, strInputFile, strDisplayString
```

```
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
```

```
Set strInputFile = objFso.GetFile("C:\Temp\Sample.txt")
```

```
'Main Processing Section
```

```
CreateDisplayString()
```

```
DisplayMessage()
```

```
TerminateScript()
```

```
'Procedure Section
```

```
Function CreateDisplayString()
```

```
    strDisplayString = "C:\Temp\Sample.txt" & vbCrLf & _  
        vbCrLf & "Created on:    " & vbTab & strInputFile.DateCreated & _  
        vbCrLf & "Last Modified: " & vbTab & strInputFile.DateLastModified & _  
        vbCrLf & "Last Accessed: " & vbTab & strInputFile.DateLastAccessed & _  
        vbCrLf
```

```
End Function
```

```
Function DisplayMessage()
```

```
    MsgBox strDisplayString
```

```
End Function
```

```
Function TerminateScript()
```

```
    'Stop the execution of this script
```

```
    WScript.Quit()
```

```
End Function
```

The main thing to take away from this example is that it interacts with the Windows file system using properties belonging to the File object to collect information about a given file. To work with the File object, you have to use the FileSystemObject object's GetFile() method, which first requires that you set up an instance of the FileSystemObject object.

If you run this script on a computer running Windows 8.1, you'll see output similar to that shown in Figure 8.4.

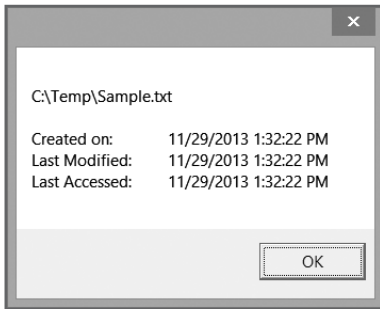


Figure 8.4 Using `FileSystemObject` object properties to retrieve information about a file. © 2014 Cengage Learning.

Opening and Closing Files

Now that you know how to instantiate the `FileSystemObject` object within your VBScripts and have seen an example of how to use it to reference other run-time objects and their associated properties, you are ready to start learning how to work with files and folders.

Before you open a file or create a new file, you must determine whether the file already exists. You can do this using the `FileSystemObject` object's `FileExists()` method as demonstrated here:

```
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
If (objFso.FileExists("C:\Temp\Sample.txt")) Then
    . . .
End If
```

To begin working with a file, you must open it. This is done using the `FileSystemObject` object's `OpenTextFile()` method, which requires that you provide the following pieces of information:

- Name and path of the file
- How to open the file
- Whether to create a new file if the file does not already exist

Table 8.1 defines constants and the values you will use to tell the `OpenTextFile()` method how to open the file.

TABLE 8.1 `OPENTEXTFILE()` CONSTANTS

Constant	Description	Value
<code>ForReading</code>	Opens a file in preparation for reading	1
<code>ForWriting</code>	Opens a file in preparation for writing	2
<code>ForAppending</code>	Opens a file allowing text to be written to the end of the file	8

Table 8.2 outlines the two available options that determine what the `OpenTextFile()` method should do if the file does not already exist.

TABLE 8.2 OPENTEXTFILE() FILE CREATION OPTIONS

Value	Description
True	Open a file if it already exists; create and open a new file if it does not already exist
False	Open a file if it already exists; otherwise, take no additional action

© Jerry Lee Ford, Jr. All Rights Reserved.

You must be careful to always specify the appropriate constant value when telling the `OpenTextFile()` method how to open a file. For example, if you accidentally open a file in `ForWriting` mode when you actually meant to append to the end of the file, then you will overwrite the contents already stored in the file.

Let's look at a VBScript that puts what you have just learned into action. In this example, the script opens a file named `Sample.txt`, which resides in the `Temp` directory on the computer's `C:` drive. If the file exists, the script opens it. If the file doesn't already exist, the script creates it. Once opened, the script writes a few lines of text and then closes the file.

```

*****
'Script Name: FileCreate.vbs
'Author:      Jerry Ford
'Created:     02/10/14
'Description: This script demonstrates how to create, open, and write
'a line of text to a text file.
*****

'Initialization Section
Option Explicit

Dim objFso, objFileHandle, strFileName
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
strFileName = "C:\Temp\Sample.txt"

'Main Processing Section

OpenTextFile()
WriteTextOutput()
CloseTextFile()
TerminateScript()

```

```
'Procedure Section
```

```
Function OpenTextFile()  
    'If exists open it in append mode, otherwise create and open a new file  
    If (objFso.FileExists(strFileName)) Then  
        Set objFileHandle = objFso.OpenTextFile(strFileName, 8)  
    Else  
        Set objFileHandle = objFso.OpenTextFile(strFileName, 2, "True")  
    End If  
End Function
```

```
Function WriteTextOutput()  
    'Write three lines of text to the text file  
    objFileHandle.WriteLine "Once upon a time there was a little boy who"  
    objFileHandle.WriteLine "lived in a shoe. Unfortunately it was two sizes"  
    objFileHandle.WriteLine "too small!"  
End Function
```

```
Function CloseTextFile()  
    'Close the file when done working with it  
    objFileHandle.Close()  
End Function
```

```
Function TerminateScript()  
    'Stop the execution of this script  
    WScript.Quit()  
End Function
```

It is very important that you always remember to close any open files before allowing your scripts to end. You do this by using the `FileSystemObject` object's `Close()` method as shown here:

```
objFileHandle.Close()
```

If you forget to close a file after working with it, the file might become corrupted because the end-of-file marker has not been created for it.

Trap

You can open a file using only one mode at a time. In other words, if you open a file in `ForReading` mode, your script cannot write new text to the file unless you first close the file and then open it again in `ForWriting` mode.

If you save and run this script on a computer running Windows 8.1 and then open the Sample.txt file using Windows Notepad, you'll see the output shown in Figure 8.5.

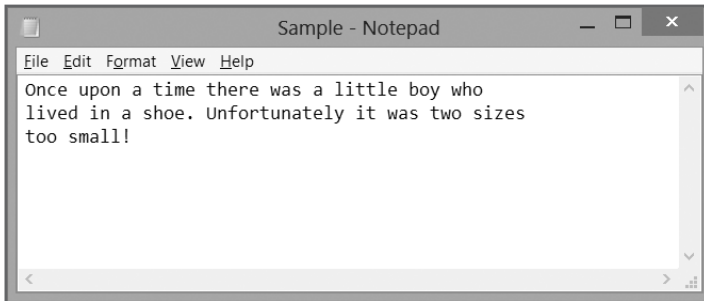


Figure 8.5 Writing to a new or existing text file. © 2014 Microsoft Corporation. Used with permission from Microsoft.

Writing to Files

You have a number of different options when it comes to how your VBScripts write text to files:

- Writing one or more characters at a time
- Writing a line at a time
- Writing blank lines

Writing text one or more characters at a time is good when you need to carefully format text output, such as when you want to create reports with data lined up in columns. On the other hand, writing out one line of text at a time is convenient when the format is more free form. Finally, inserting or writing blank lines into text files helps you improve the file's appearance, such as adding space between the heading and the text in a formal report.

Writing Characters

To write text to a file one or more characters at a time, you need to use the `FileSystemObject` object's `Write()` method. This method does not append a carriage return to the end of any text that it writes. If two back-to-back write operations are performed using the `Write()` method, for example, the text from the second write operation is inserted into the text file on the same line immediately following the text written by the first write operation. To see how this works, take a look at the following example:

```
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
Set objFileHandle = _
    objFso.OpenTextFile("C:\Temp\Sample.txt", 2, "True")
objFileHandle.Write("Once upon a time there ")
objFileHandle.Write("were three little bears.")
objFileHandle.Close()
```

If you save and run this example on a computer running Windows 8.1, you'll see that the output added to the text file by the script is placed on the same line, as shown in Figure 8.6.

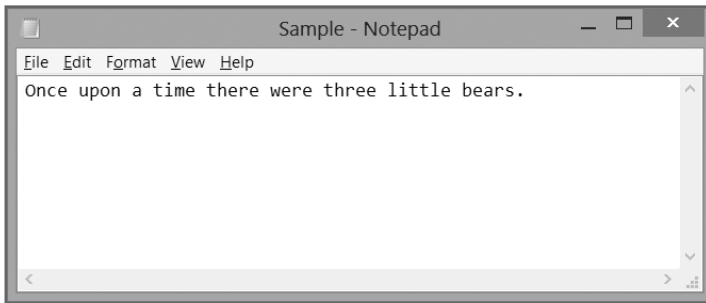


Figure 8.6 Writing characters to a text file. © 2014 Microsoft Corporation. Used with permission from Microsoft.

Writing Lines

You can modify the previous example to write text output to the file a line at a time by replacing the `Write()` method with the `WriteLine()` method:

```
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
Set objFileHandle = _
    objFso.OpenTextFile("C:\Temp\Sample.txt", 2, "True")
objFileHandle.WriteLine("Once upon a time there were three little bears.")
objFileHandle.Close()
```

If you save and run this example, you'll find that the results are almost exactly the same as those produced by the previous example. The one difference is that the cursor is now positioned at the beginning of the next row in the text file when you run this example, whereas the cursor was left at the end of the first row when you ran the previous example. This is because the `WriteLine()` method automatically appends a linefeed to the end of each write operation.

Adding Blank Lines

You can add blank lines to the output generated by your VBScripts to make it look better by using the `FileSystemObject` object's `WriteBlankLines()` method. This method executes by writing a blank line and then advancing the cursor down to the beginning of the next row in the file.

The following example demonstrates how to use the `FileSystemObject` object's `WriteBlankLines()` method:

```
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
Set strInputFile = objFso.GetFile("C:\VBScriptGames\Hangman.vbs")
Set objFileHandle = _
    objFso.OpenTextFile("C:\Temp\Sample.txt", 2, "True")
```



```
objFileHandle.WriteLine(1)
objFileHandle.WriteLine("C:\VBScriptGames\Hangman.vbs Properties")
objFileHandle.WriteLine(1)
objFileHandle.WriteLine("-----")
objFileHandle.WriteLine(1)
objFileHandle.WriteLine("Creation Date: " & strInputFile.DateCreated)
objFileHandle.WriteLine(1)
objFileHandle.WriteLine("Last Modified: " & strInputFile.DateLastModified)
objFileHandle.WriteLine(1)
objFileHandle.WriteLine("-----")

objFileHandle.Close()
```

Figure 8.7 shows how the Sample.txt file looks after the script has executed on a computer running Windows 8.1.

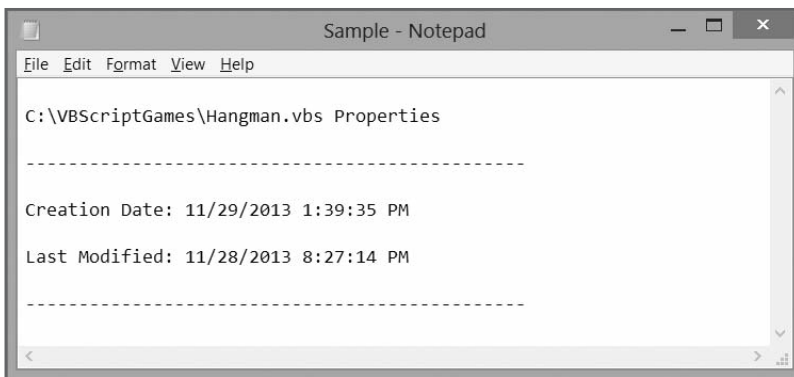


Figure 8.7 Blank lines improve the presentation of reports created by your VBScripts.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

Reading from Files

VBScript makes it just about as easy to read from a file as it is to write to one. However, before you attempt to read or process the contents of a text file, you should first make sure the file has data in it. If the file has no data, opening the file and attempting to read it is pointless.

You can use the `TestStream` object's `AtEndOfStream` property to determine whether a file contains data. Not only should you check the `AtEndOfStream` property before you begin reading a file, but your script should continue to check this property before each additional read operation to make sure data is still in the file for the script to read. In other words, you need to keep checking to make sure that your script has not reached the end-of-file marker.

To demonstrate how all this works, let's build an example. For starters, create a new VBScript and add the following statements to it:

```
Dim objFso, objFileHandle, strDisplayString
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
Set objFileHandle = objFso.OpenTextFile("C:\Temp\Sample.txt", 1)
```

The first statement defines variables to be used by the script. The next statement instantiates the `FileSystemObject` object. The third statement sets up an object reference to the `Sample.txt` file and opens it in `ForReading` mode. Now add the following statements to the script:

```
Do While objFileHandle.AtEndOfStream = False
    strDisplayString = strDisplayString & objFileHandle.ReadLine() & vbCrLf
Loop
MsgBox strDisplayString
```

The first statement sets up a `Do...While` loop that checks on each iteration to determine whether the end-of-file marker has been reached. As long as the end of the file has not been reached, another line of the file is read and appended to the end of a variable named `strDisplayString`.

Finally, close the file by adding the following statement:

```
objFileHandle.Close
```

If you run this script on a computer running Windows 8.1, you'll see output similar to that shown in Figure 8.8.

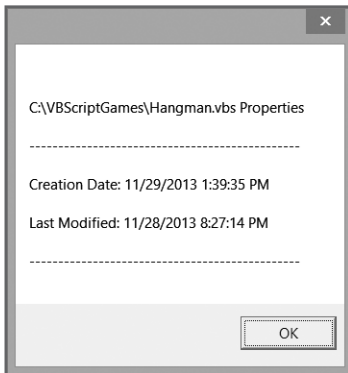


Figure 8.8 Creating a VBScript that reads and displays the content of text files.

© 2014 Cengage Learning.

This example showed you how to read an entire file, a line at a time. Other options that are available to you when reading files include the following:

- Skipping lines
- Reading one or more characters at a time
- Reading the entire file at one time

Skipping Lines

Often text reports and other files begin with some type of heading or other information that you might not be particularly interested in. In these cases, you can skip the reading of these lines using the `FileSystemObject` object's `Skip()` and `SkipLine()` methods. The `Skip()` method allows your script to skip a specific number of characters at a time, whereas the `SkipLine()` method allows your script to skip entire lines.

To use the `Skip()` method, you must pass it the number of characters that you want skipped as demonstrated here:

```
objFileHandle.Skip(25)
```

The `SkipLine()` method does not accept any arguments, so it can only be used to skip one line at a time. However, by wrapping the execution of this method up within a loop, you can skip as many lines as you want as demonstrated here:

```
For intCounter = 1 To 5
    objFileHandle.SkipLine()
Next
```

Reading Files Character by Character

To read a file one or more characters at a time, you need to work with the `FileSystemObject` object's `Read()` method. This might be necessary for reading files with fixed length data. For example, you might have a report file that lists a certain category or information in a column that begins at character position 20 and ends at character position 30 on each line of the report. Using the `Read()` method, you could develop a VBScript that reads the file and pulls out only the information stored in that column for each line of the file.

To get a glimpse of the `Read()` method in action, consider the following example. This script uses a `For` loop and the `SkipLine()` method to skip the reading of the first five lines of the `Sample.txt` file. The `Skip()` method skips the first 15 characters on the next line in the file (line 6). The `Read()` method reads the next 22 characters from the file. The file is then closed and the text that was read from the file is displayed.

```
Dim objFso, objFileHandle, intCounter, strDisplayString
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
Set objFileHandle = objFso.OpenTextFile("C:\Temp\Sample.txt", 1)

For intCounter = 1 To 5
    objFileHandle.SkipLine()
Next

objFileHandle.Skip(15)
strDisplayString = objFileHandle.Read(22)
objFileHandle.Close

MsgBox "C:\Temp\Sample.txt was created on " & strDisplayString
```

Reading a File All at Once

Sometimes all you need to do is read an entire file in a single operation, such as when the file to be read is already formatted to produce the output that you want. You can do this using the `FileSystemObject` object's `ReadAll()` method as demonstrated here:

```
Dim objFso, objFileHandle, strDisplayString
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
Set objFileHandle = objFso.OpenTextFile("C:\Temp\Sample.txt", 1)

strDisplayString = objFileHandle.ReadAll()
objFileHandle.Close()
MsgBox strDisplayString
```

As you can see, the script reads the entire file using just one `ReadAll()` operation and then displays what it read using a `MsgBox()` statement. The output displayed when you run this script is exactly the same as that shown in Figure 8.8, in which the script read the entire text file a line at a time before displaying the file's contents.

Managing Files and Folders

In addition to using VBScript and the WSH to read and write to and from text files, you can use them to help administer all your files and folders. By *administer*, I mean to help keep them organized—or more specifically, to automate the organization process. For example, if your computer is connected to a local area network, you can create a VBScript that makes copies of all the files it finds in certain folders and stores them on a network server. That way, if something ever happens to your original files, you can always recover them by retrieving a backup copy.

The `FileSystemObject` object provides several methods for developing scripts that can automate file administration:

- **CopyFile()**. This copies one or more files.
- **MoveFile()**. This moves one or more files.
- **DeleteFile()**. This deletes one or more files.
- **FileExists()**. This determines whether a file exists.

The `FileSystemObject` object also provides a large collection of methods for automating folder administration:

- **CopyFolder()**. This copies one or more folders.
- **MoveFolder()**. This moves one or more folders.
- **DeleteFolder()**. This deletes one or more folders.
- **FolderExists()**. This determines whether or not a folder exists.
- **CreateFolder()**. This creates a new folder.

Trick

As an alternative to working with methods belonging to the `FileSystemObject` object, you also can work with methods that jointly belong to the `File` and `Folder` objects:

- `Copy()`. This copies a single file or folder.
- `Delete()`. This removes a single file or folder.
- `Move()`. This moves a single file or folder.

There are a few drawbacks to these methods. First, using these methods requires that you instantiate both the `FileSystemObject` object and the `File` or `Folder` object, which is just a little more work. Second, they work with just one file at a time as opposed to the methods belonging to the `FileSystemObject` object, which can work on more than one file or folder at a time. Finally, because the `File` and `Folder` objects share the same methods, it's sometimes easy to make a mistake and accidentally mess things up. For example, if you have a file and a folder in the same location with the same or a similar name, it's easy to accidentally delete the wrong one.

Copying, Moving, and Deleting Files

Using the `FileSystemObject` object's `CopyFile()` method, you can create a VBScript that can copy one or more files, as demonstrated in the following example:

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.CopyFile "C:\Temp\Sample.txt", "C:\VBScriptGames\Sample.txt"
```

In this example, a file named `Sample.txt` located in the `Temp` folder on the `C:` drive is copied to the `VBScriptGames` folder on the `C:` drive. By modifying this example, as shown here, you can change the script so that it copies more than one file. Specifically, this example results in all files named “Sample” being copied, regardless of their file extensions.

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.CopyFile "C:\Temp\Sample.*", "C:\VBScriptGames"
```

Trick

Wildcard characters make it possible to copy, move, or delete more than one file or folder at a time. The `?` and `*` wildcard characters enable pattern matching. The `?` character is used to specify a single character match. The `*` character is used to match against an unlimited number of characters. For example, specifying `*.txt` results in a match with all files that have a `.txt` file extension. Specifying `sampl?.txt` results in a match only with files that have a six-character-long file name with “Sampl” as the first five characters, and a `.txt` file extension.

Copying One or More Files

You can copy one or more files using the `CopyFile()` method. This method also supports an additional parameter that allows you to specify what to do if the script attempts to copy a file to a folder that already contains a file with the same name. You specify a value of either `True` or `False` for this parameter. A value of `True` tells `CopyFile()` to replace or override files with duplicate file names. A value of `False` tells `CopyFile()` to cancel the copy operation for any files with matching file names.

The following example demonstrates how to copy all files with a `.txt` file extension located in the `C:\Temp` folder to a folder called `C:\VBScriptGames` without allowing any duplicate files already located in the destination folder to be overridden:

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.CopyFile "C:\Temp\*.txt", "C:\VBScriptGames", "False"
```

This next example does the exact same thing as the previous example, except that it allows duplicate files to be overridden:

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.CopyFile "C:\Temp\*.txt", "C:\VBScriptGames", "True"
```

Trick

Remember, you can avoid errors by using the `FileSystemObject` object's `FileExists` and `FolderExists` properties to verify whether a file or folder exists before manipulating it.

Moving One or More Files

The difference between moving and copying files is that after you copy a file, you end up with two copies in two places, whereas when you move a file, only the one file exists in its new location. You can move files from one folder to another using the `FileSystemObject` object's `MoveFile()` method:

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.MoveFile "C:\Temp\*.txt", "C:\VBScriptGames"
```

In this example, all files with a `.txt` file extension are moved from the `C:\Temp` folder into the `C:\VBScriptGames` folder.

Deleting One or More Files

You can delete one or more files using the `FileSystemObject` object's `DeleteFile()` method:

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.DeleteFile "C:\VBScriptGames\*.txt"
```

In this example, all the files in the `C:\VBScriptGames` folder with a `.txt` file extension are deleted.

Creating a New Folder

You can create new folders by using the `FileSystemObject` object's `CreateFolder()` method. For example, the following script checks to see whether a folder named `VBScriptGames` already exists on the computer's `C:` drive. If it does not exist, the script creates it.

```
Dim objFso, strNewFolder
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
If (objFso.FolderExists("C:\VBScriptGames") = False) Then
    Set strNewFolder = objFso.CreateFolder("C:\VBScriptGames")
End If
```

Trap

Always check to be sure that a folder does not exist before trying to create it. If the folder that you are trying to create already exists, your script will get an error.

Copying Folders

The only differences between copying a folder and copying a file are that you specify the `CopyFolder()` method instead of the `CopyFile()` method and specify a folder name instead of a file name. Of course, not only is the specified folder copied to a new location, but also all its contents are copied, as demonstrated in the following example:

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.CopyFolder "C:\Temp", "C:\VBScriptGames\Temp"
```

Here, a complete copy of the `Temp` folder and everything stored in it is replicated inside the `C:\VBScriptGames` folder.

Trick

If you want, you can give the new copy of the specified folder a new name when copying it by simply specifying a new folder name:

```
objFso.CopyFolder "C:\Temp", "C:\VBScriptGames\Temporary"
```

If a folder with the same name already exists in the destination specified by the `CopyFolder()` method, the contents of the source folder are added to the files and folders that are already present. You can tell the `CopyFolder()` method what to do if duplicate file and folder names are found in the destination folder by adding an optional third parameter and setting its value to either `True` or `False`. Specifying a value of `True` causes matching files to be overridden. Specifying a value of `False` prevents matching files from being overridden. For example, the following VBScript statements prevent files with duplicate file names from being overridden:

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.CopyFolder "C:\Temp", "C:\VBScriptGames\Temp", "False"
```

This next example allows files with duplicate names to be overridden:

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.CopyFolder "C:\Temp", "C:\VBScriptGames\Temp", "True"
```

Moving Folders

You can use the `FileSystemObject` object's `MoveFolder()` method to move folders from one location to another. Of course, when you move a folder, you also move all its contents to the new destination. Take a look at the following example:

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.MoveFolder "C:\Temp", "C:\VBScriptGames\Temp"
```

The `Temp` folder and all its contents are moved from the root of the `C:` drive to the `C:\VBScriptGames` folder.

Deleting Folders

You can use the `FileSystemObject` object's `DeleteFolder()` method to delete one or more folders. This method deletes the folder and any subfolders or files stored inside it. To see how it works, look at the following example, which deletes a folder named `Temp` from the `C:\VBScriptGames` folder.

```
Dim objFso
Set objFso = WScript.CreateObject("Scripting.FileSystemObject")
objFso.DeleteFolder "C:\VBScriptGames\Temp"
```

Trap

Be extra careful when using the `DeleteFolder()` method. When executed, this method deletes not only the specified folder, but also anything stored within it.

Storing Script Configuration Settings in External Files

Up to this point in the book, all the scripts you've seen have been controlled by configuration settings embedded within the scripts themselves. By *configuration settings*, I mean constants and variables that were set up to store data that was then used to control how the scripts executed. For example, I've controlled the text that the scripts display in pop-up dialog boxes by assigning a text string to a constant that I've defined at the beginning of each script. To change this display text for a given script, you must open the script and modify it. However, every time you open a script to make even the most simple change, you run the risk of accidentally making a typo that breaks something.

It's often a good idea to remove or externalize script configuration settings. One way of doing this is to store the script configuration settings in external text files from which your scripts can then open and retrieve the settings. This is accomplished using INI (pronounced "eye en eye") files. *INI* or *initialization* files are plain-text files that have an .ini file extension. Programmers use INI files to store configuration settings for the operating systems, hardware settings, and software settings.

The nice thing about using INI files is that if you make a mistake when editing them, and as a result your scripts break, it's much easier to find your typo in the INI file than it would be in your script. Also, if you plan to share your scripts with other people—especially people without programming backgrounds—once explained, they'll find modifying INI files relatively easy, whereas editing your scripts might overwhelm them.

Note

You also can externalize script configuration settings by storing them in the Windows Registry. You'll learn how to do this in Chapter 10, "Using the Windows Registry to Configure Script Settings."

INI File Structure

INI files have a specific structure that you need follow when creating them, as shown here:

```
;Sample INI file
```

```
[Section1]  
key1=value1  
key2=value2
```

For starters, INI files are organized into sections. Each section's beginning has a section header enclosed within a pair of matching brackets. In the example, Section1 is the only section. Sections are made up of zero or more key-value pairs. In the example, there are two key-value pairs. You can think of a key as being akin to a variable name and a value as being the data that is assigned to the key.

INI files also can have comments, which begin with the ; character, as demonstrated in the previous INI file. INI files also can contain any number of blank lines, which can be added to the INI file to make it easier to read.

INI files are processed in a top-down order. They can have any number of sections, and these sections can contain any number of key-value pairs. INI files are typically named after the script or program that is associated with them. For example, if you create an INI file to be used by a VBScript named INIDemo.vbs, you would probably name its INI file INIDemo.ini.

A Working Example

To better understand how to work with INI files, let's look at an example. This example consists of a script named INIDemo.vbs that is designed to retrieve configuration settings from an INI file named INIDemo.ini. Figure 8.9 illustrates the format of the INI file, as shown on a computer running Windows 7.

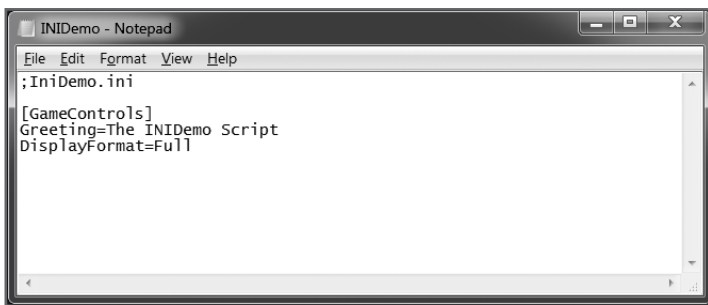


Figure 8.9 The INI file used by the INIDemo.vbs script contains a single section made up of two key-value pairs.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

The VBScript statements that make up the INIDemo.vbs script are shown here. The script is relatively short, but it is a little involved, so I embedded a lot of comments to help explain what the script is doing.

```
Set objFso = CreateObject("Scripting.FileSystemObject")

strIniFile = "C:\VBScriptGames\INIDemo.ini" 'Specify INI file location

If (objFso.FileExists(strIniFile)) Then 'Make sure INI file exists

    'Open for reading
    Set objOpenFile = objFso.OpenTextFile(strIniFile, 1)

    Do Until Mid(strInput, 1, 14) = "[GameControls]" 'Find right section
        strInput = objOpenFile.ReadLine 'Read line from the INI file
    Loop

    'Read until end of the file
    Do Until objOpenFile.AtEndOfStream = "True"
```

```

strInput = objOpenFile.ReadLine 'Read a line from the file
If Mid(strInput, 1, 1) = "[" Then
    Exit do 'A new section has been found
End If

If Len(strInput) <> 0 Then 'If not a blank line
    intFindEquals = Instr(strInput, "=") 'Locate the equals character
    strKeyName = Mid(strInput, 1, intFindEquals - 1) 'set key value

    Select Case strKeyName 'Match up key value to scripts settings
        Case "Greeting"
            strGreetingMsg = _
                Mid(strInput, intFindEquals + 1, Len(strInput))
        Case "DisplayFormat"
            strDisplayType = _
                Mid(strInput, intFindEquals + 1, Len(strInput))
    End Select
End If

Loop

objOpenFile.Close() ' close the INI file when done reading it

End If

'Display the configuration setting retrieved from the INI file
MsgBox "Configuration setting: strGreetingMsg = " & strGreetingMsg & _
    vbCrLf & vbCrLf & "Configuration setting: strDisplayType = " & _
    strDisplayType

```

The script begins by instantiating an instance of the `FileSystemObject` object. Next, a variable named `strIniFile` is used to store the script's INI file location. The `FileSystemObject` object's `FileExists()` method is used to verify that the INI file exists. The `FileSystemObject` object's `OpenTextFile()` method is then used to open the INI file in `ForReading` mode.

A `Do...Until` loop executes until the "GameControls" section is located. This is accomplished using the built-in VBScript `Mid()` function. After the section is found, a second `Do...Until` loop executes and runs until the end of the file is reached (until `objOpenFile.AtEndOfStream = "True"`). However, if a new section is found while the rest of the INI file is being read, the `Do...Until` loop is terminated using an `Exit Do` statement. Next, the VBScript `Len()` function is used to determine whether the current line is blank. If it's not blank, then the key portion of the key-value pair is processed by first locating the equals sign and then assigning all text before the equals sign to a variable named `strKeyName`.

Trick

The trick to extracting script configuration settings from key-value pairs in INI files is the script's use of the built-in VBScript `Mid()` function. This function retrieves or parses out a specified number of characters from a string. The `Mid()` function has the following syntax:

```
Mid(string, StartPosition[, Length])
```

string represents the string that the `Mid()` function is to parse. *StartPosition* identifies the character position within the specified string where the parsing operation should begin. *Length* is optional. When identified, *Length* specifies the number of characters to be returned. If omitted, then all characters from the start position to the end of the string are returned.

After a key has been processed, a `Select Case` statement is set up to inspect the value associated with the key to determine what it is equal to. After the “GameControls” section has been processed, the `Do...Until` loop terminates and the script displays the configuration settings that were extracted from the INI file as shown in Figure 8.10, which shows the results on a computer running Windows 7.

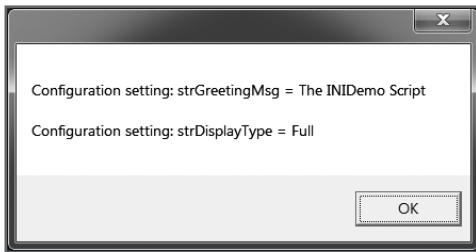


Figure 8.10 The output displayed by the `INIDemo.vbs` script demonstrates how to extract configuration settings from INI files. © 2014 Cengage Learning.

Of course, a script that only displays the configuration settings that it extracts from its INI file isn't really that useful. However, it does provide a working example of how to process INI files. You'll get the chance to modify this example by adapting its logic to work with the Lucky Lottery Number Picker game.

Back to the Lucky Lottery Number Picker

The heart of the Lucky Lottery Number Picker game resides in the script's main processing section, which contains a collection of function calls and two loops that control the generation of as many sets of lottery numbers as the player asked for. Script configuration settings are stored in an external INI file, which is retrieved by the script at execution. The configuration settings are used to specify the following:

- How many lottery numbers are required to complete a full set
- The message text to be displayed in the title bar or the pop-up dialog boxes displayed by the script
- The range of numbers from which lottery numbers are to be selected
- Whether to display the results generated by the script in full or summary format

Designing the Game

In total, the script will consist of 10 functions, each of which is designed to perform a specific task. The names of these 10 functions and the tasks they perform are as follows:

- **SetVariableDefaults()**. This establishes default values for a number of script variables.
- **ProcessScriptIniFile()**. This retrieves configuration settings from the script's external INI file.
- **CollectPlayerInput()**. This prompts the player to specify the number of lottery numbers to be generated.
- **GetRandomNumber()**. This generates random lottery numbers.
- **ProcessRandomNumber()**. This ensures that duplicate lottery numbers are not generated.
- **DetermineIfSetIsComplete()**. This determines when a full set of lottery numbers has been generated.
- **BuildDisplayString()**. This assembles the display string that will be used to show the player the lottery numbers generated by the script.
- **ResetVariableDefaults()**. This resets default variables to prepare the script for the generation of additional sets of lottery numbers.
- **DisplayFinalResults()**. This displays the lottery numbers generated by the script.
- **DisplaySplashScreen()**. This displays information about the script and its author.

Designing the Script's INI File

The first step in creating the Lucky Lottery Number Picker game is to create the game's INI file, which will be named `LuckyLotteryNumberPicker.ini`. The complete text of this INI file is shown here:

```
;LuckyLotteryNumberPicker.ini file
```

```
[GameControls]
Greeting=Lucky Lottery Number Picker
DisplayFormat=Full
NoOfPicks=6
RangeOfNumbers=50
```

The INI file consists of a single section named "GameControls." A total of four key-value pairs have been defined.

Setting Up the Initialization Section

As with all the scripts in this book, development begins with the script's initialization section as shown here:

```

'*****
'Script Name: LuckyLotteryNumberPicker.vbs
'Author: Jerry Ford
'Created: 02/08/14
'Description: This script randomly picks lottery numbers
'*****

'Initialization Section

Option Explicit

Dim aintLotteryArray(10) 'Stores randomly generated lottery numbers
Dim blnAllNumbersPicked 'Determines when a set of numbers has been created
Dim blnInputValidated   'Set to True when the player enters a valid number

Dim intNumberCount      'Tracks the number of picks for a given play
Dim intNoOfValidPicks   'Tracks the number of valid selections for a given set
Dim intNoOfPlays        'Determines how many sets of lottery numbers to create
Dim intSetCount         'Used to track how many sets have been generated
Dim intRandomNo        'Used to store randomly generated lottery numbers
Dim intNoOfPicksToSelect 'Specifies how many numbers to generate for each set
Dim intRangeOfNumbers  'Specifies range to use when generating random numbers
Dim strLotteryList      'Displays a string showing one set of lottery numbers
Dim strDisplayString    'Used to display the list of selected lottery numbers
Dim strDisplayType     'Specifies whether to show full or summary data
Dim strTitleBarMsg     'Specifies title bar message in pop-up dialog boxes

```

Because the script uses an array and a large number of variables, I chose to define them individually and to document each variable's purpose by adding a comment just to the right of each variable.

Developing the Logic for the Main Processing Section

The script's main processing section controls the overall execution of the script. It consists of 10 function calls and two loops:

```

SetVariableDefaults()
ProcessScriptIniFile()
CollectPlayerInput()

For intSetCount = 1 to intNoOfPlays
    Do Until blnAllNumbersPicked = "True"
        GetRandomNumber()

```

```
        ProcessRandomNumber()  
        DetermineIfSetIsComplete()  
    Loop  
    BuildDisplayString()  
    ResetVariableDefaults()  
Next  
  
DisplayFinalResults()  
DisplaySplashScreen()
```

The first loop is controlled by a For statement that is responsible for making sure that the script generates the number of sets of lottery numbers specified by the player. The second loop is controlled by a Do...Until statement and is responsible for making sure that a full count of numbers is generated for each set (or play).

Building the SetVariableDefaults() Function

The SetVariableDefaults() function, shown here, is responsible for establishing default values for a number of variables used by the script. The first two variables are Boolean and are used to determine when a full set of lottery numbers has been generated and when the player has specified a valid number of plays. The second pair of variables is used to store integer data. The first variable in this pair is used to keep track of the number of lottery numbers generated for each play. The second variable is used to track the number of sets of lottery numbers as the script is generating them.

```
Function SetVariableDefaults()  
    blnAllNumbersPicked = "False"  
    blnInputValidated = "False"  
    intNumberCount = 0  
    intNoOfValidPicks = 0  
End Function
```

Building the ProcessScriptIniFile() Function

The ProcessScriptIniFile() function, shown here, is responsible for reading in script configuration settings from the game's INI file. Because of the unique task assigned to this function, I chose to make it completely self contained. Therefore, it begins by defining its own objects and variables. To make the purpose of each variable clear, I documented each one by adding comments to the right of each variable when defined as well as to key statements throughout the function.

```
Function ProcessScriptIniFile()  
  
    Dim FsoObject           'Sets up a reference to the FileSystemObject  
    Dim OpenFile           'Sets up a reference to the script's INI file
```

```

Set FsoObject = WScript.CreateObject("Scripting.FileSystemObject")

Dim intEquals          'Used to parse INI file data
Dim strKeyName        'Represents a key in the script's INI file
Dim strSourceFile     'Specifies the name of the script's INI file
Dim strInput          'Represents a line in the script's INI file

strSourceFile = "LuckyLotteryMachine.ini" 'Identify script's INI file

If (FsoObject.FileExists(strSourceFile)) Then 'Make sure INI file exists

    'Open for reading
    Set OpenFile = FsoObject.OpenTextFile(strSourceFile, 1)

    Do Until Mid(strInput, 1, 15) = "[GameControls]" 'Find right section
        strInput = OpenFile.ReadLine 'Read line from the INI file
    Loop

    'Read until end of file reached
    Do Until OpenFile.AtEndOfStream = "True"

        strInput = OpenFile.ReadLine 'Read a line from the file

        If Mid(strInput, 1, 1) = "[" Then
            Exit do 'If executed, new sections have been found
        End If

        If Len(strInput) <> 0 Then 'Executes if a blank line is not found

            intEquals = Instr(strInput, "=") 'Locate the equals character
            strKeyName = Mid(strInput, 1, intEquals - 1) 'Set key value

            Select Case strKeyName 'Match up key value to script settings
                Case "Greeting"
                    strTitleBarMsg = Mid(strInput, intEquals + 1, Len(strInput))
                Case "DisplayFormat"
                    strDisplayType = Mid(strInput, intEquals + 1, Len(strInput))
                Case "NoOfPicks"
                    intNoOfPicksToSelect = Cint(Mid(strInput, intEquals + 1, _

```



```

        Len(strInput)))
    Case "RangeOfNumbers"
        intRangeOfNumbers = Cint(Mid(strInput, intEquals + 1, _
        Len(strInput)))
    End Select

End If

Loop

OpenFile.Close()'Close the INI file when done reading it

Else

    MsgBox "The INI file is missing. Unable to execute."
    WScript.Quit()

End If

End Function

```

The function begins by instantiating an instance of the `FileSystemObject` object. It then specifies the location of its INI file. Next, it checks to make sure that the INI file exists and then opens it. The function then reads the INI file until it finds the “GameControls” section. Once found, the function begins reading the rest of the INI file. The function then parses through the key-value pairs and assigns values to matching script variables using a `Select Case` statement.

Building the `CollectPlayerInput()` Function

The `CollectPlayerInput()` function is responsible for collecting and validating player input. The overall execution of this function is controlled by the following `Do...Until` loop, which executes as long as a Boolean variable named `blnInputValidated` is not equal to `True`:

```

Function CollectPlayerInput()

    Do Until blnInputValidated = "True"

        intNoOfPlays = InputBox("How many sets of numbers do " & _
        "you want?", strTitleBarMsg)

        If IsNumeric(intNoOfPlays) <> True Then
            MsgBox "Sorry. You must enter a numeric value. Please " & _
            "try again.", ,strTitleBarMsg
        End If
    Loop
End Function

```

```
Else
  If Len(intNoOfPlays) = 0 Then
    MsgBox "Sorry. You must enter a numeric value. Please " & _
      "try again.", ,strTitleBarMsg
  Else
    If intNoOfPlays = 0 then
      MsgBox "Sorry. Zero is not a valid selection. Please " & _
        "try again.", ,strTitleBarMsg
    Else
      blnInputValidated = "True"
    End If
  End If
End If
Loop
```

```
End Function
```

Three validation tests are performed. The first test uses the VBScript `IsNumeric()` function to ensure that the input is numeric. The second test uses the `Len()` function to ensure that the player actually typed in input as opposed to simply clicking OK or Cancel. The last validation test checks to make sure that the player did not enter a value of 0. If the input provided by the player passes all three of these tests, then a value of `True` is assigned to `blnInputValidated` and the function finishes executing.

Building the `GetRandomNumber()` Function

The `GetRandomNumber()` function, shown here, is responsible for retrieving random numbers for the script. It begins with the `Randomize` statement to ensure that numbers are randomly generated. Next, a random number is generated. The range from which the number is created is dictated by the value assigned to `intRangeOfNumber`, which was previously established by retrieving its value from the script's INI file.

```
Function GetRandomNumber()
  Randomize
  intRandomNo = cInt(FormatNumber(Int((intRangeOfNumbers * Rnd) + 1)))
End Function
```

Building the `ProcessRandomNumber()` Function

The `ProcessRandomNumber()` function, shown here, is responsible for ensuring that the same lottery number is not picked twice for a given play or set. It accomplishes this by establishing an array named `aintLotteryArray`. The array is set up to handle up to 11 entries, based on the assumption that this is large enough to handle any amount of lottery numbers a given lottery game might require.

```
Function ProcessRandomNumber()  
  Select Case intRandomNo  
    Case aintLotteryArray(0)  
    Case aintLotteryArray(1)  
    Case aintLotteryArray(2)  
    Case aintLotteryArray(3)  
    Case aintLotteryArray(4)  
    Case aintLotteryArray(5)  
    Case aintLotteryArray(6)  
    Case aintLotteryArray(7)  
    Case aintLotteryArray(8)  
    Case aintLotteryArray(9)  
    Case aintLotteryArray(10)  
    Case Else  
      strLotteryList = strLotteryList & " " & intRandomNo & vbTab  
      intNoOfValidPicks = intNoOfValidPicks + 1  
      aintLotteryArray(intNumberCount) = intRandomNo  
      intNumberCount = intNumberCount + 1  
    End Select  
  End Function
```

This function begins by comparing the value of the last lottery number that was generated to the numbers stored in the array. The first time through, there won't be any lottery numbers stored in the array yet. As a result, the lottery number is stored as the first entry in the array. Also, the lottery number is added to a string that is stored in a variable named `strLotteryList`, which is used elsewhere in the script. Finally, the total number of valid lottery numbers is tracked by adding 1 to `intNoOfValidPicks` each time a unique lottery number is generated.

Each time this function is called, it checks to see whether the most recently generated random number matches any of the numbers already stored in the array. If it does, nothing happens; otherwise, that number is added to the array.

Building the `DetermineIfSetIsComplete()` Function

The `DetermineIfSetIsComplete()` function, shown here, compares the value stored in `intNoOfValidPicks` to the value stored in `intNoOfPicksToSelect` to determine whether a complete set of lottery numbers has been generated. If a complete set has been generated, then `DetermineIfSetIsComplete()` sets the value assigned to `blnAllNumbersPicked` equal to `True`. Otherwise, the value assigned to this variable remains set equal to `False`.

```
Function DetermineIfSetIsComplete()  
  If intNoOfValidPicks = intNoOfPicksToSelect Then
```

```
        blnAllNumbersPicked = "True"  
    End If  
End Function
```

Building the BuildDisplayString() Function

The `BuildDisplayString()` function, shown here, uses the string stored in the `strLotteryList` variable—which is created by the `ProcessRandomNumber()` function—to build a larger string made up of all the sets of lottery numbers generated by the game. This string is later used to display the game's result to the player. To make the displayed output more attractive, this function uses the `vbTab` constant to organize output into a multi-column format.

```
Function BuildDisplayString()  
    strLotteryList = intSetCount & ")" & vbTab & strLotteryList  
    strDisplayString = strDisplayString & strLotteryList & _  
        vbCrLf & vbCrLf & vbCrLf  
End Function
```

Building the ResetVariableDefaults() Function

The `ResetVariableDefaults()` function, shown here, is used to reset variable values back to their initial default settings after a full set of lottery numbers has been generated. This readies the script to begin generating additional sets of numbers.

```
Function ResetVariableDefaults()  
    blnAllNumbersPicked = "False"  
    intNoOfValidPicks = 0  
    intNumberCount = 0  
    strLotteryList = ""  
End Function
```

Building the DisplayFinalResults() Function

The `DisplayFinalResults()` function, shown next, is responsible for displaying all the sets of lottery numbers that are generated. It displays this information in one of two formats based on the value assigned to `strDisplayType`, which is a variable whose value was set earlier in the script by retrieving its value from the script's INI file. If `strDisplayType` is equal to `Full`, then the function displays information regarding the number of lottery numbers generated per set as well as the total number of sets created, followed by the numbers that made up each set. However, if the value assigned to `strDisplayType` is equal to anything other than `Full`, then only the sets of lottery numbers are displayed.

```
Function DisplayFinalResults()  
    If strDisplayType = "Full" Then  
        MsgBox vbCrLf & _
```

```

"L U C K Y   L O T T E R Y   N U M B E R   P I C K E R" & _
vbCrLf & vbCrLf & _
"-----" & _
"-----" & vbCrLf & vbCrLf & _
"Number of plays: " & intNoOfPlays & vbCrLf & vbCrLf & _
"Number of picks per play: " & intNoOfPicksToSelect & _
vbCrLf & vbCrLf & _
"-----" & _
"-----" & vbCrLf & vbCrLf & vbCrLf & _
"Your lottery numbers are: " & vbCrLf & vbCrLf & vbCrLf & _
strDisplayString, , strTitleBarMsg

```

Else

```

MsgBox vbCrLf & _
"L U C K Y   L O T T E R Y   N U M B E R   P I C K E R" & _
vbCrLf & vbCrLf & _
"-----" & _
"-----" & vbCrLf & vbCrLf & _
"Your lottery numbers are: " & vbCrLf & vbCrLf & vbCrLf & _
strDisplayString, , strTitleBarMsg

```

End If

End Function

Building the DisplaySplashScreen() Function

This last function in the script displays the game's splash screen, providing information about the game and its creator as well as an invitation for the player to return and play again another time.

```

Function DisplaySplashScreen()
    MsgBox "Thank you for using the Lucky Lottery Number Picker " & _
        "© Jerry Ford 2014." & vbCrLf & vbCrLf & "Please play again " & _
        "soon!", 4144, strTitleBarMsg
    WScript.Quit()
End Function

```

The last statement in the function terminates the script's execution using the `WScript.Quit()` method.

The Final Result

Okay. At this point you should have all of the information required to complete the Lucky Lottery Number Picker game. If you have not done so yet, go ahead and create your own copy of the game and then crank it up and see how it works. After you've cleaned out any errors that you might have made when typing, you'll have a pretty cool script.

Summary

In this chapter, you learned how to create and store data in text files. You also learned how to open text files and read or process their contents as input. In addition, you learned how to use properties and methods belonging to the `WSH FileSystemObject` object to perform assorted file administration tasks, including copying, moving, and deleting individual and groups of files and folders.

You now understand the fundamentals of working with files and folders and know everything you need to begin developing scripts that can create reports and log files. On top of all this, you also can now develop and leverage the power of INI files as a repository for externalizing script configuration settings.

Challenges

1. As it is currently written, the Lucky Lottery Number Picker game attempts to display as many sets of lottery numbers as it is asked for. However, depending on the screen resolution, only so many sets of numbers can be displayed at one time. The result is that when too many sets of numbers are specified, some won't be visible to the player. To remedy this, modify the script so that it will display only 10 sets of numbers at a time using as many pop-up dialog boxes as required to display all the script's output.
2. Provide the player with the capability to save the lottery numbers generated by the game to a text file. This allows players to print their numbers and take the list with them when they go to purchase their lottery tickets.
3. The Lucky Lottery Number Picker game is set up so that it always displays a closing splash screen before ending. Modify the script and its associated INI file so that the player can enable or disable the display of the splash screen.
4. As it is currently written, the Lucky Lottery Number Picker game prevents the player from entering nonnumeric input such as letters and special characters. It also forces the player to enter *something*. (The player can't just click OK or Cancel.) However, there is no logic in the script to prevent negative numbers from being accepted. As unlikely as it may be for the player to enter a negative number, it's a good idea to modify the script to prevent them from being accepted anyway.

This page intentionally left blank

9

Handling Script Errors

Every programmer, no matter how good he may be, runs into errors when writing and testing scripts and programs. Like any other programming language, VBScript is subject to many types of errors. Errors may be inevitable, but you can minimize their number and lessen their effects. In this chapter, I'll demonstrate a number of scripting errors and show you how to deal with them. Specifically, you will learn the following:

- How to fix errors by reading and analyzing error messages
- How to write VBScripts that can ignore errors and keep going
- How to create error-handling routines to recover from many error situations
- How to generate test errors to validate the performance of your error-handling routines
- How to keep a record of errors using log files and the Windows application event log

Project Preview: The Hangman Game

This chapter's programming project is the creation of a VBScript version of the classic children's game Hangman. Developing this game will require you to use all the VBScript knowledge that you've accumulated so far, including applying advanced conditional logic, organizing critical processes into procedures, and validating player input to prevent errors from prematurely terminating the game.

The Hangman game begins by presenting the player with a number of blank spaces representing the letters of the game's mystery word. The player is then prompted to guess the letters that make up the word. If the player guesses the word before making six incorrect guesses, he wins. Otherwise, the game ends by displaying the mystery word, and the player is asked if he would like to play again. Unfortunately, because of the display limitations of the WSH, you won't be able to actually animate a hanging in the event the player loses. Still, by creating a well-formatted output, the player will probably never even notice.

Figures 9.1 through 9.4 demonstrate the overall flow of the game from beginning to end on a computer running Windows 8.1.



Figure 9.1 The Hangman game begins with a graphical welcome message and an invitation to play. © 2014 Cengage Learning.

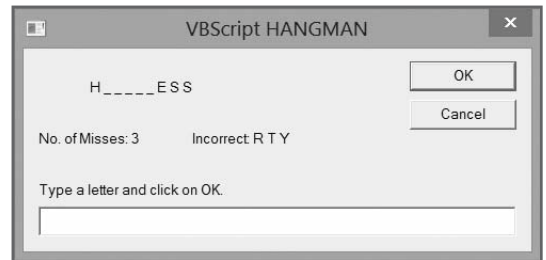


Figure 9.2 The game displays both the letters that the player has correctly guessed and the letters the player has incorrectly guessed. © 2014 Cengage Learning.

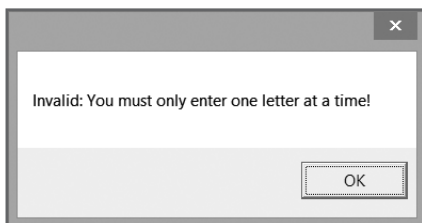


Figure 9.3 A number of possible messages may be displayed if the player does not play the game correctly.

© 2014 Cengage Learning.

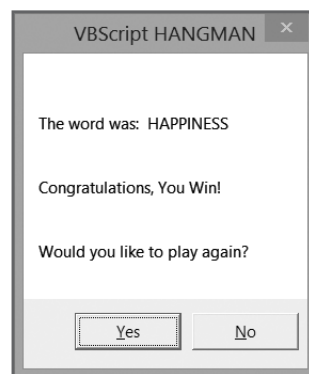


Figure 9.4 Each game ends by displaying the hidden word, the results of the game, and an invitation to play again.

© 2014 Cengage Learning.

Understanding VBScript Errors

Errors can appear, even in small scripts, for many reasons. Errors may be generated when a script is first loaded or interpreted. Errors occurring at this stage are referred to as *syntax errors*. Syntax errors are often the result of typos. For example, you might accidentally type a single quote when you meant to type a double quote. Syntax errors also occur when you inadvertently mistype a VBScript keyword. Because syntax errors are discovered during the initial loading of a script, they are usually easily caught and corrected during script development.

Errors can also be generated during script execution. These types of errors are referred to as *run-time errors*. Run-time errors appear only when the statements that generate them are executed. As a result, some run-time errors might not be detected when the script executes (if the statement containing the error is not executed). For example, a run-time error might be hidden deep within a function or subroutine that is seldom called.

With proper testing of all the components of a script, most run-time errors can be discovered and fixed during script development. I say “most” because not all run-time errors can be caught. For example, those caused by unforeseen circumstances may be impossible to detect during script development. Perhaps the person running the script incorrectly supplied input in a manner that you could not have anticipated, or perhaps something is wrong with the environment in which the script is being executed. Or maybe the hard disk has become full, preventing your VBScript from writing to a file, or the network goes down as your script is executing a file copy or move operation. In cases such as these, often the best you can do is to end the script gracefully, without confusing the user or making the situation worse.

Another category of error to which scripts are susceptible is logical errors. Logical errors are mistakes made by the script developer. For example, instead of looping 10 times, you might accidentally set up an endless loop. Another example of a logical error is a situation in which a script adds two numbers that should have been multiplied.

Definition

A *syntax error* is an error that occurs as a result of improperly formatted statements within scripts.

Definition

A *run-time error* is an error that occurs when a script tries to perform an illegal action, such as multiplying a numeric and a character value.

Definition

A *logical error* is an error produced as a result of a programming mistake on the part of the script developer.

Understanding Error Messages

VBScript error messages are generated for both syntax and run-time errors. These errors are displayed in the form of pop-up dialog boxes, as shown in Figure 9.5, which shows the result of a script executed on a computer running Window 7. Each error message displays information about the error, including a brief description of the error, an error number, and the source of the error.

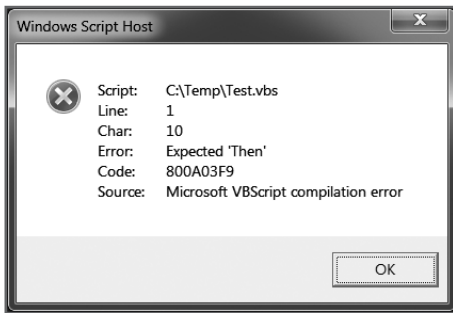


Figure 9.5 A typical VBScript error message.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

The error message displayed in Figure 9.5 is the result of a syntax error. As you can see, a lot of useful information about the error is automatically provided. The ability to interpret and understand this information is critical for troubleshooting and fixing your VBScripts.

The following information has been provided in this error message:

- **Script.** The name and location of the VBScript that produced the error
- **Line.** The line number within the VBScript where the error was detected
- **Char.** The column number position within the line where the error was detected
- **Error.** A brief description of the error
- **Code.** An error number identifying the type of error
- **Source.** The resource that reported the error

You can see in Figure 9.5 that a VBScript named Test.vbs located in C:\Temp generated the error. Line 6 of the script that generated this error looks like the following statement:

```
If X > 5 MsgBox "Hello World!"
```

This If statement uses the VBScript MsgBox() function to display a text string. The error message indicates that the problem is that VBScript expected to find the Then keyword and did not. If you look at the middle of this statement, you'll see that, in fact, the Then keyword is absent. To correct this error, you would add the missing keyword, like this:

```
If X > 5 Then MsgBox "Hello World!"
```

To verify that the error has been eliminated, you could then save and run the script again.

Preventing Logical Errors

Your VBScripts will do exactly what you tell them to do, even if that's not what you really mean for them to do. Therefore, it's extremely important that you plan your VBScript project carefully. For example, you should begin with a pseudo code outline and then translate that into a flowchart, outlining each of the script's major components. You should, as much as possible, develop the script a component at a time,

testing each component as you go. I'll show you some different ways to test individual script components as you develop the Hangman game.

Logical errors often make their presence known by presenting incorrect or unexpected results, and can be the most difficult type of error to track down. Unlike syntax and run-time errors, which display messages that describe the nature of their problems, logical errors force you to look through some or all of your script a line at a time to find the faulty logic. The good news is that with careful planning and design, logical errors can be avoided.

Dealing with Errors

There are many measures you can take to prevent errors from occurring in your VBScripts. I've already mentioned the need to plan and carefully design and test your scripts. In addition, you can avoid many errors by taking the following advice:

- Provide a simple, easy-to-use interface (such as pop-up dialog boxes).
- Provide clear instructions so the user will understand exactly what is expected of him.
- Reuse code from existing scripts whenever possible by cutting and pasting code that has already been thoroughly tested.
- Validate input data as much as possible.
- Explicitly declare all your variables.
- Use a consistent naming scheme for all constants, variables, object references, arrays, functions, and subroutines.
- Be on guard for endless loops.
- Do your best to anticipate and handle specific situations where errors are likely to occur.

Unfortunately, errors will occur. There are three basic ways that you can deal with them as they arise in your VBScripts:

- Simply let them happen and deal with the consequences as problems are uncovered.
- Tell VBScript to ignore errors and keep going.
- Attempt to anticipate where errors are most likely to occur and try to handle them in a way that either terminates the script's execution gracefully or allows the script to recover and keep going.

Letting Errors Happen

Errors are going to happen. One way of dealing with them is to simply let them happen and instruct users to report them when they occur, along with as much information as possible about what the user was doing when the error occurred. That way, you can attempt to reproduce the error, figure out what caused it, and then fix it.

Normally I would not recommend this approach. After all, your reputation as a VBScript guru depends on the soundness and reliability of your scripts. However, a cost is associated with every VBScript that you write. You may measure this cost in terms of time, effort, or by some other scale. Each time you sit down to create a new script, you must make a judgment call as to how much time and energy you have available to put into the project. You also need to consider the consequences of an error occurring in the script that you're developing. After all, it is entirely possible to develop a simple script in a matter of minutes and spend another hour or more trying to make it bulletproof, only to find that something has gone wrong anyway.

If you're developing an extremely important script that will have high visibility and for which you will be held accountable if a problem arises, then you'll want to do everything you can to keep errors from happening. On the other hand, if you have been asked to create a "quick and dirty" script to help someone perform a noncritical task, you might be able to get away with simply letting any errors occur. All you may need to do is tell the person for whom you wrote the script to give you call if a problem arises so that you can make a quick modification to the script to fix it.

Just keep this thought in mind: Most users will have no idea what a typical VBScript error message means or what to do if they receive one. It's important to, at a minimum, provide clear instructions on how to use your VBScripts and what to do if an error does occur.

Ignoring Errors

Another option that you might want to consider when developing your scripts is to tell VBScript to ignore any errors that occur. In some cases, this will work just fine. For example, suppose you wrote a VBScript that was supposed to connect to a number of networked computers and copy over a file located in a certain folder at regular intervals throughout the day. As problems sometimes occur on networks, it may be acceptable to ignore situations in which the script is unable to connect to a particular network drive, especially if you know that the script will run again later and get another chance to copy the missing file.

This approach, while effective, should be used with caution. There are few situations in which skipping an error will not result in the generation of another error later in a script. For example, if your script was supposed to perform another operation on each file copied from the network drive, then depending on how you wrote the script, the part of the script that performs this next step might generate an error.

To tell VBScript to ignore errors within your script, add the following statement, exactly as shown, to the beginning of your script:

```
On Error Resume Next
```

Even if you add this statement to your scripts, certain errors will still be reported. The key to using this statement is that it must be placed in your script before any statements in which you think an error is likely to occur. You can later cancel the effects of this statement using the following statement:

```
On Error GoTo 0
```

For example, the following statement will produce an error message because the keyword `WScript` is misspelled:

```
WScrip.Echo "Hello world!"
```

Adding `On Error Resume Next` before the statement prevents the error from appearing and allows the rest of the script to continue:

```
On Error Resume Next  
WScrip.Echo "Hello world!"
```

Now look at two more statements:

```
On Error Resume Next  
WScrip.Echo "Hello world!"  
On Error Goto 0  
WScrip.Echo "Goodbye world!"
```

The `On Error Goto 0` statement nullifies the effects of the `On Error Resume Next` statements for all statements that follow. Therefore, the first error is ignored, but the second error is reported and the script halts its execution.

Like with variables, VBScript allows you to localize the effects of the `On Error Resume Next` statement to the procedure level. In other words, if this statement is placed within a procedure, then it will be in effect only for as long as the procedure executes, and will be nullified when the procedure (that is, the function or subroutine) finishes executing. Combining the `On Error Resume Next` statement with procedures enables you to significantly limit the effects of this powerful statement.

Creating Error Handlers

The third option that you have for dealing with errors in your VBScripts is to create error handlers. To effectively use error handlers, you must be able to anticipate locations within your scripts where errors are likely to occur and then develop the appropriate programming logic to deal with, or handle, these errors.

You can handle errors in different ways. For example, you can create error handlers that do the following:

- Reword cryptic VBScript errors.
- Provide the user with instructions.
- Give the user another try.
- Apologize for the error.
- Ask the user to report the error.
- Take a corrective action.
- Log the occurrence of the error.

Definition

An error handler is an error-triggered routine that alters the execution environment's default handling of an error condition.

To set up an error handler, you need to know how to work with the `Err` object. The `Err` object provides a number of properties and methods that allow your scripts to access error information and clear error conditions. To access information about an error, you need to reference the following three properties:

- **Number.** This retrieves the last error number.
- **Description.** This retrieves the last error message.
- **Source.** This retrieves the name of the object that raised (or caused) the error.

You can modify the contents of any of these three properties, which enables you to reassign a custom error number and message and even modify source information. For example, in a particularly complex script, you might want to create and document your own custom set of error messages.

The first step in creating an error handler is to add the `On Error Resume Next` statement to your VBScript. You can then add the error-handling statements like this:

```
On Error Resume Next
```

```
NonExistentFunction()
```

```
If Err > 0 then
```

```
    Err.Number = 9999
```

```
    Err.Description = "This script is still a work in progress."
```

```
    MsgBox "Error: " & Err.Number & " - " & Err.Description
```

```
    Err.Clear
```

```
End if
```

Save these statements as a script and execute them. Because the script does not contain a procedure named `NonExistentFunction()`, an error will be generated. However, instead of displaying a VBScript run-time error message, the error-handling routine creates and displays the error message shown in Figure 9.6.

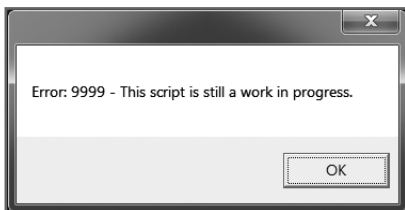


Figure 9.6 A custom error message generated by a VBScript error handler, as seen on a computer running Windows 7. © 2014 Cengage Learning.

The `Err` object also provides two very useful methods. One of these methods is the `Clear()` method. This method clears out or removes the previous error, ensuring that the next time a script checks for an error, it will not get a false status (that is, it won't see an already-handled error).

To use the `Clear()` method, place it at the end of your error-handling routine, as demonstrated in the previous example. VBScript automatically executes the `Clear()` method on several occasions, including the following:

- Whenever the `On Error Resume Next` statement executes
- Whenever an `Exit Sub` statement executes
- Whenever an `Exit Function` statement executes

The second `Err` object method is the `Raise()` method. This method allows you to generate error messages to test your error-handling routines. Without this method, the only way that you could test your error-handling routines would be to deliberately introduce an error situation into your code. This method is easy to use, as demonstrated by the following:

```
Err.Raise(500)
```

For example, if you save the previous statement as a script and run it on a computer running Windows 7, you will see the error message shown in Figure 9.7.

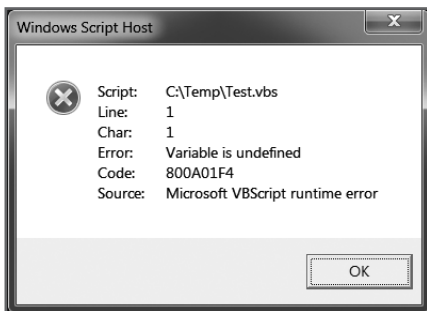


Figure 9.7 Using the `Err` object's `Raise()` method to generate a test error.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

To use the `Raise()` method, add it, along with the error number indicating the error that you want to generate, just before the error-handling procedure that you want to test in your script. After you have validated that the error handler is working as expected, remove the `Raise()` statement from your VBScript.

Reporting Errors

The best solution for errors is to prevent them from occurring in the first place. However, that's not always possible. The next best solution is to devise a way of dealing with errors, whether it be handling them or simply ignoring them. Another option is to report errors by recording them to a log file for later review. This enables you to come back and check to see whether any errors have occurred. This is important because many times users do not report errors when they occur, allowing the errors to go on forever. By logging error messages, you create an audit log that you can come back to and review from time to time to identify and fix any errors that may have occurred.

When logging error messages, you have two options:

- Creating your own custom log file
- Recording error messages in the Windows application event log

Creating a Custom Log File

To create a custom log file, you must instantiate the `FileSystemObject` object in your VBScript and then use its `OpenTextFile()` method to open the log file so that your script can write to it, as demonstrated in the following example:

```
On Error Resume Next

Err.Raise(7)

Set objFsoObject = WScript.CreateObject("Scripting.FileSystemObject")
If (objFsoObject.FileExists("C:\ScriptLog.txt")) Then
    Set objLogFile = objFsoObject.OpenTextFile("C:\ScriptLog.txt", 8)
Else
    Set objLogFile = objFsoObject.OpenTextFile("C:\ScriptLog.txt", 2, "True")
End If

objLogFile.WriteLine "Test.vbs Error: " & Err.Number & ", Description = " & _
    Err.Description & " , Source = " & Err.Source

objLogFile.Close()
```

In this example, the `On Error Resume Next` statement is used to allow the script to recover from errors and the `Err.Raise(7)` statement is used to simulate an “out of memory” error. The rest of the script logs the error in a file called `ScriptLog.txt`, located on the computer’s C: drive. If the file does not exist, it is created. Error messages are appended to the bottom of the file each time they are written, allowing a running history of information to accumulate. For more information about how to work with the `FileSystemObject` object and its methods and properties, refer to Chapter 8, “Storing and Retrieving Data.”

You can adapt the previous example as the basis for developing an error-logging routine in your VBScripts. Simply copy and paste all but the first two lines into a function and call it whenever errors occur. Just make sure that you call the function before clearing the error. Alternatively, you can modify the example to use variable substitution and pass the function the error number and description as arguments.

Trap

Be sure you always close any file that you open before allowing your script to terminate. If you don’t, you may have problems with the file the next time you want to open it because its end-of-file marker may be missing.

Recording an Error Message in the Application Event Log

An alternative to creating custom log files for your scripts is to record error messages in the Windows application event log. This is achieved using the `WshShell` object's `LogEvent()` method:

```
On Error Resume Next
```

```
Err.Raise(7)
```

```
Set objWshShl = WScript.CreateObject("WScript.Shell")
objWshShl.LogEvent 1, "Test.vbs Error: " & Err.Number & ", Description = " & _
    Err.Description & " , Source = " & Err.Source
```

In this example, an “out of memory” error has again been simulated, only this time, the error has been written to the Windows application event log using the `WshShell` object's `LogEvent()` method. Only two arguments were processed. The first is a number indicating the type of event being logged. Table 9.1 lists the different types of events that are supported by Windows. The second argument was the message to be recorded. Figure 9.8 shows how the message will appear when viewed from the Event Viewer on a computer running Windows 7.

TABLE 9.1 EVENT LOG ERROR INDICATORS

Value	Description
0	Indicates a successful event
1	Indicates an error event
2	Indicates a warning event
4	Indicates an informational event
8	Indicates a successful audit event
16	Indicates a failed audit event

© Jerry Lee Ford, Jr. All Rights Reserved.

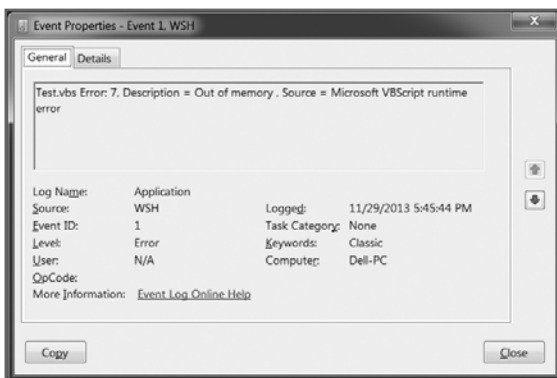


Figure 9.8 Writing error messages to the Windows application event log using the `WshShell` object's `LogEvent()` method.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

Back to the Hangman Game

Now that you've reviewed the basic steps involved in dealing with VBScript errors, let's return to the Hangman game and begin its development. I'm going to cover the development of this game from a different angle than in previous chapters. By now, you should have a pretty good idea of how things work, and you should be able to read and understand the scripts that you'll see throughout the remainder of this book. (Just in case, I'll leave plenty of comments in the code to help you along.) This time, I'll provide a much higher-level explanation of what is going on and offer suggestions for ways to test and develop this script one step at a time. I'll also point out techniques that you can use to test and track the results of functions within the script so that you can validate their operation without having to first complete the entire script.

Designing the Game

The overall design of the Hangman game is fairly complex. To simplify things, I'll begin the game-development process by designing a flowchart, shown in Figure 9.9, that breaks the game down into distinct units, each of which is responsible for performing a unique task.

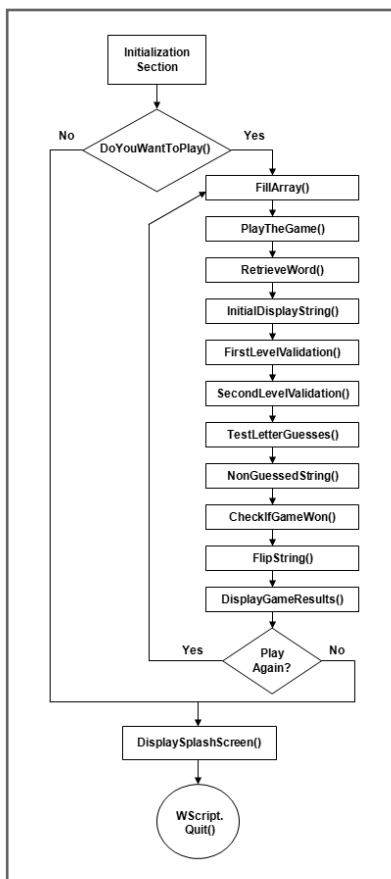


Figure 9.9 A flowchart providing a high-level design for the Hangman game.

In addition to the initialization section and the main processing section, this script is made up of 13 separate procedures. Therefore, you will develop this game in 15 steps, as follows:

1. Create a new script, adding your VBScript template and defining the variables, constants, and arrays that are used by this script.
2. Develop the controlling logic for the main processing section.
3. Use the `DoYouWantToPlay()` function to create an introductory game splash screen and determine whether the user wants to play.
4. Assign a list of game words to an array using the `FillArray()` function.
5. Create a loop in the `PlayTheGame()` function that controls the actual flow of the game, collecting player guesses and calling other functions as required.
6. Retrieve a randomly selected game word using the `RetrieveWord()` function.
7. Display space-separated underscore characters representing each letter in the game word using the `InitialDisplayString()` function.
8. Use the `FirstLevelValidation()` function to validate the player's input to make sure the player is providing valid guesses.
9. Use the `SecondLevelValidation()` function to test whether the player has already tried guessing a letter before accepting it as input.
10. Use the `TestLetterGuess()` function to check to see whether the player made an incorrect guess.
11. Use the `NonGuessedString()` function to create a temporary string blanking out the letters correctly guessed by the player.
12. Use the `CheckIfGameWon()` function to check to see whether the player has guessed all the letters that make up the mystery word.
13. Use the `FlipString()` function to spin through the script created in step 11, and reverse the display of each character of the string (that is, now only show the correctly guessed letters).
14. Tell the player whether he won or lost using the `DisplayGameResults()` function.
15. Display information about the game as it finishes using the `SplashScreen()` function.

Setting Up the Script Template and Initialization Section

This portion of the script, shown next, should look pretty familiar to you by now, and does not require much explanation. As you can see from the code, this section consists of the script template and the definition of the script's constant, variables, and array.

```

*****
'Script Name: Hangman.vbs
'Author:      Jerry Ford
'Created:     02/30/02
'Description: This script demonstrates how to create a game of Hangman
'              using VBScript and the WSH.
*****

'Initialization Section

Option Explicit

Const cTitlebarMsg = "VBScript HANGMAN"

Dim strChoice, strGameWord, intNoMisses, intNoRight, strSplashImage
Dim intPlayOrNot, strMsgText, intPlayAgain, strWrongGuesses
Dim strRightGuesses, blnWordGuessed, intLetterCounter
Dim strTempStringOne, strTempStringTwo, strWordLetter, strDisplayString
Dim intFlipCounter, intRandomNo, strProcessGuess, blnGameStatus
Dim strCheckAnswer

Dim astrWordList(9) 'Define an array that can hold 10 game words

```

Putting Together the Logic for the Main Processing Section

Like the other scripts you have seen in this book, the logic located in the script's main processing section is very straightforward. It calls upon procedures that determine whether the user wants to play, loads the game words into an array, starts the game, and ultimately ends the game by displaying a splash screen and executing the `WScript.Quit()` statement.

```

'Main Processing Section

intPlayOrNot = DoYouWantToPlay()
If intPlayOrNot = 6 Then 'User elected to play the game
    FillArray()
    PlayTheGame()
End If

SplashScreen()
WScript.Quit()

```

Trick

At this point in the script, you have enough code in place to run your first test and see whether there are any syntax errors. For now, I recommend that you go ahead and define a procedure for each of the preceding functions, placing a `MsgBox()` function that simply displays the name of the function inside each one. Save and execute the script and make sure the pop-up dialog boxes all appear when they should. You can leave the functions as they are until you are ready to complete them.

Trick

Using the `WScript.Quit()` method, as I did in this section, is not required. Script execution would have ceased after the display of the splash screen anyway. I added this statement for the sake of clarity, and to prevent any statements that I might have inadvertently left outside of a function in the procedure section from accidentally being executed.

Building the DoYouWantToPlay() Function

You've seen functions very similar to this one in previous chapters. All the `DoYouWantToPlay()` function does is display a clever graphic and ask the user if he wants to play a game of Hangman.

```
Function DoYouWantToPlay()
```

```
'Display the splash screen and ask the user if he wants to play
strSplashImage = Space(88) & "*****" & vbCrLf & _
    "W E L C O M E T O " & Space(55) & "*" & Space(13) & "*" & _
    vbCrLf & Space(88) & "0" & Space(13) & "*" & vbCrLf & _
    "V B S c r i p t H A N G M A N !" & Space(35) & "--| |--" & _
    Space(10) & "*" & vbCrLf & Space(87) & "/" & Space(1) & "\" & _
    Space(12) & "*" & vbCrLf & Space(103) & "*" & vbCrLf & Space(103) & _
    "*" & vbCrLf & space(99) & " ***** " & vbCrLf & _
    "Would you like to play a game?" & vbCrLf & " "
```

```
DoYouWantToPlay = MsgBox(strSplashImage, 36, cTitlebarMsg)
```

```
End Function
```

This is a good place to pause and perform another test of your script to ensure that this function looks and works like it should. This test allows you to evaluate the operation of all the controlling logic in the main processing section.

Building the FillArray() Function

The `FillArray()` function, shown next, simply loads a list of words into an array. Later, another procedure will randomly select a game word from the array.

```
Function FillArray()  
    'Add the words to the array  
    astrWordList(0) = "AUTOMOBILE"  
    astrWordList(1) = "NETWORKING"  
    astrWordList(2) = "PRACTICAL"  
    astrWordList(3) = "CONGRESS"  
    astrWordList(4) = "COMMANDER"  
    astrWordList(5) = "STAPLER"  
    astrWordList(6) = "ENTERPRISE"  
    astrWordList(7) = "ESCALATION"  
    astrWordList(8) = "HAPPINESS"  
    astrWordList(9) = "WEDNESDAY"  
End Function
```

You can't perform much of a test on this function at this point, but you can always save and run the script again to see whether you have any syntax problems. You should create a temporary script, copy this function into it, and then create a For...Next loop that processes and displays the contents of the array to ensure that the function is loading as expected. Next, delete the For...Next loop and add the following statements to the beginning of the temporary script:

```
Dim astrWordList(9) 'Define an array that can hold 10 game words  
FillArray()
```

Save this script again. A little later, I'll show you how to modify and use this temporary script to perform another test.

Building the PlayTheGame() Function

The PlayTheGame() function, shown next, controls the play of the Hangman game. When I developed this function, I wrote a few lines, stopped and tested it, and then wrote some more. Specifically, each time I added a call to an external function I stopped, wrote the function that I called, and then did a test to be sure that everything worked before continuing. However, it would take me too long to guide you through every step along the way. Instead, I'll leave it up to you to follow this basic process, and will instead focus on the development of the other functions that make up the script, most of which are called from within the PlayTheGame() function.

```
Function PlayTheGame()  
  
    'Initialize variables displayed by the game's initial pop-up dialog box  
    intNoMisses = 0  
    intNoRight = 0  
    strWrongGuesses = ""  
    strRightGuesses = ""
```

```
'Get the game a mystery word
strGameWord = RetrieveWord()

'Call function that formats the initial pop-up dialog box's display string
strDisplayString = InitialDisplayString()

strTempStringOne = strGameWord

'Let the player start guessing
Do Until intNoMisses = 6

    'Collect the player's guess
    strChoice = InputBox(vbCrLf & vbTab & strDisplayString & vbCrLf & _
        vbCrLf & vbCrLf & "No. of Misses: " & intNoMisses & _
        " " & vbTab & "Incorrect:" & strWrongGuesses & vbCrLf & _
        & vbCrLf & vbCrLf & _
        "Type a letter and click on OK." , cTitleBarMsg)

    'Determine if the player has quit
    If strChoice = "" Then
        Exit Function
    End If

    strProcessGuess = FirstLevelValidation()

    'The player wants to quit the game
    If strProcessGuess = "ExitFunction" Then
        Exit Function
    End If

    'The player typed invalid input
    If strProcessGuess <> "SkipRest" Then

        strProcessGuess = SecondLevelValidation()

    Select Case strProcessGuess
        Case "DuplicateWrongAnswer"
            MsgBox "Invalid: You've already guessed this incorrect letter."
        Case "DuplicateRightAnswer"
            MsgBox "Invalid: You've already guessed this correct letter."
```


Case Else

```
strCheckAnswer = TestLetterGuess()  
If strCheckAnswer <> "IncorrectAnswer" Then  
  
    'Reset the value of variable used to build a string containing  
    'the interim stage of the word as currently guessed by player  
    strTempStringTwo = ""  
  
    NonGuessedString()  
  
    'Check to see if the player has guessed the word  
    blnGameStatus = CheckIfGameWon()  
    If blnGameStatus = "True" Then  
        blnWordGuessed = "True"  
        Exit Do  
    End If  
  
    'Set the value of the temporary string equal to the string  
    'created by the Previous For...Next loop  
    strTempStringOne = strTempStringTwo  
  
    'Clear out the value of the strDisplayString variable  
    strDisplayString = ""  
  
    FlipString()  
  
    End If  
End Select  
End If  
Loop  
  
DisplayGameResults()
```

End Function

Building the RetrieveWord() Function

This function is designed to retrieve a randomly selected word to be used by the game. RetrieveWord() first selects a random number between 1 and 10, and then uses that number to retrieve a game word from the WordList() array. This function randomly retrieves a word from an array.

```
Function RetrieveWord()  
    Randomize  
    intRandomNo = FormatNumber(Int(10 * Rnd))  
    RetrieveWord = astrWordList(intRandomNo)  
End Function
```

This is a good place to perform another test. This time, open the temporary script that you created a little earlier in the “Building the FillArray() Function” section and cut and paste it into the statements located in the previous function. Paste the three statements into the temporary file, making them lines 3 through 5 in the script. Next, add the following statement as line 6:

```
MsgBox RetrieveWord
```

Save and run the script. Each time you execute the temporary script, a different randomly selected word should be displayed. If this is not the case, then something is wrong. Locate and fix any errors that may occur until the temporary script works as expected. Then, cut and paste any corrected script statements back into your Hangman script and move on to the next section.

Building the InitialDisplayString() Function

This function is used to display a series of underscore characters representing each letter that makes up the mystery game word:

```
Function InitialDisplayString()  
    'Create a loop that processes each letter of the word  
    For intLetterCounter = 1 to Len(strGameWord)  
        'Use underscore characters to display a string representing each  
        'letter  
        InitialDisplayString = InitialDisplayString & "_ "  
    Next  
End Function
```

You can run a quick test of this function by creating a new temporary VBScript, cutting and pasting the statements from within this function into the temporary script, and modifying it.

```
For intLetterCounter = 1 to Len("DOG")  
    'Use underscore characters to display a string representing each letter  
    InitialDisplayString = InitialDisplayString & "_ "  
Next  
  
MsgBox InitialDisplayString
```

When you run the script, you should see three underscore characters separated by blank spaces, indicating the length of the word. If anything is wrong, fix it and then copy the corrected statement(s) back into the Hangman script.

Building the FirstLevelValidation() Function

The FirstLevelValidation() function, shown next, ensures that the player is providing valid input. It checks to make sure that the player typed something, that the player did not type more than one character, and that a number or a symbol was not provided as input.

```
'Validate the player's input
Function FirstLevelValidation()

    'See if the player clicked Cancel or failed to enter any input
    If strChoice = "" Then
        FirstLevelValidation = "ExitFunction"
        Exit Function
    End If

    'Make sure the player only typed one letter
    If Len(strChoice) > 1 Then
        MsgBox "Invalid: You must only enter one letter at a time!"
        FirstLevelValidation = "SkipRest"
    Else
        'Make sure the player did not type a number by accident
        If IsNumeric(strChoice) = "True" Then
            MsgBox "Invalid: Only letters can be accepted as valid input!"
            FirstLevelValidation = "SkipRest"
        Else
            FirstLevelValidation = "Continue"
        End If
    End If

End Function
```

Building the SecondLevelValidation() Function

Like the previous function, the SecondLevelValidation() function, shown here, performs additional tests on the player's guess to make sure that the player is not trying to guess the same letter twice.

```
Function SecondLevelValidation()
    'Check to see if this letter is already on the list of incorrect guesses
    If Instr(1, strWrongGuesses, UCase(strChoice), 1) <> 0 Then
        SecondLevelValidation = "DuplicateWrongAnswer"
    Else
        'Check to see if this letter is already on the list of correct guesses
```

```

    If Instr(1, strRightGuesses, UCase(strChoice), 1) <> 0 Then
        SecondLevelValidation = "DuplicateRightAnswer"
    End If
End If
End Function

```

Building the TestLetterGuess() Function

The TestLetterGuess() function, shown here, checks to see whether the letter is part of the word and keeps track of missed guesses. If the total number of missed guesses equals 6, then this function assigns a value of False to the blnWordGuessed variable. This variable is a flag that is later checked to see whether the player has lost the game.

```

Function TestLetterGuess()
    If Instr(1, UCase(strGameWord), UCase(strChoice), 1) = 0 Then
        'Add the letter to the list of incorrectly guessed letters
        strWrongGuesses = strWrongGuesses & " " & UCase(strChoice)
        'Increment the number of guesses that the player has made by 1
        intNoMisses = intNoMisses + 1
        'If the player has missed six guesses then he has used up all chances
        If intNoMisses = 6 Then
            blnWordGuessed = "False"
        End If
        TestLetterGuess = "IncorrectGuess"
    Else
        TestLetterGuess = "CorrectGuess"
    End If
End Function

```

Building the NonGuessedString() Function

As I was creating the game, I wanted an easy way of seeing what game word had been randomly selected and of tracking which letters had yet to be guessed. The NonGuessedString() function, shown next, builds a string that, if it were displayed, would show all the letters that make up the word, minus the letters that the player has correctly guessed. This function gave me a tool for displaying how the game was keeping track of the game word.

```

Function NonGuessedString()
    'Loop through the temporary string
    For intLetterCounter = 1 to Len(strTempStringOne)
        'Examine each letter in the word one at a time
        strWordLetter = Mid(strTempStringOne, intLetterCounter, 1)
    Next intLetterCounter
End Function

```

```

'Otherwise add an underscore character indicating a nonmatching guess
If UCase(strWordLetter) <> UCase(strChoice) Then
    strTempStringTwo = strTempStringTwo & strWordLetter
Else
    'The letter matches player's guess. Add it to the temporary string
    intNoRight = intNoRight + 1
    strRightGuesses = strRightGuesses & " " & UCase(strChoice)
    strTempStringTwo = strTempStringTwo & "_"
End If
Next
End Function

```

After I developed this function, I added the following statement as the last statement in the function:

```
MsgBox " **** = " & strTempStringTwo
```

This way, each time the function ran, I was able to see the contents of the string. For example, if the game word is “DOG” and the player has missed his first guess, this string would be displayed in a pop-up dialog box as “D O G.” If the player then guessed the letter O, then the string would display as “D_G” the next time this function ran. This function allowed me to visually track the progress of the string as the game ran and manipulated its contents.

Building the CheckIfGameWon() Function

The CheckIfGameWon() function checks to see whether the number of correctly guessed letters is equal to the length of the word. If this is the case, then the player has guessed all the letters that make up the word and won the game.

```

Function CheckIfGameWon()
    'Check and see if the player has guessed all the letters that make up
    'the word. If so, set indicator variable and exit the Do...Until loop
    If intNoRight = Len(strGameWord) Then
        CheckIfGameWon = "True"
    End If
End Function

```

Again, a well-placed MsgBox() in this function can be used to track the value of the CheckIfGameWon variable.

Building the FlipString() Function

The problem with the string produced by the NonGuessedString() function was that it displayed a string in exactly the opposite format that I wanted to ultimately display. In other words, if the game word was “DOG” and the player had correctly guessed the letter O, then I wanted the game to display the word as

“_O_” and not as “D_G.” So I developed the FlipString() function. It loops through each character of the string created by the NonGuessedString() function and reverses the display of character data.

```
Function FlipString()
    'Spin through and reverse the letters in the strTempStringTwo variable
    'in order to switch letters to underscore characters and underscore
    'characters to the appropriate letters
    For intFlipCounter = 1 to Len(strTempStringTwo)
        'Examine each letter in the word one at a time
        strWordLetter = Mid(strTempStringTwo, intFlipCounter, 1)
        'Replace each letter with the underscore character
        If strWordLetter <> "_" Then
            strDisplayString = strDisplayString & "_ "
        Else
            'Replace each underscore with its appropriate letter
            strDisplayString = strDisplayString & _
                Right(Left(strGameWord,intFlipCounter),1) & " "
        End If
    Next
End Function
```

Here again, a well-placed statement that contains the MsgBox() function can be used to display the activity of this function as it attempts to spin through and reverse the display of the letters that make up the game word.

Building the DisplayGameResults() Function

The DisplayGameResults() function, shown here, determines whether the player won or lost the game. It is also responsible for displaying the results of the game and for determining whether the player wants to play again. If the user elects to play another game, the strings that are used to track the status of the game word are blanked out and the PlayTheGame() function is called. Otherwise, the function ends and processing control is passed back to the end of the current iteration of the PlayTheGame() function. This then returns control to the main processing section where the SplashScreen() function is called.

```
'Determine if the player won or lost and display game results
Function DisplayGameResults()

    'Select message based on whether or not the player figured out the word
    If blnWordGuessed = "True" Then
        strMsgText = "Congratulations, You Win!"
    Else
        strMsgText = "Sorry, You Lose."
    End If
```

```
'Display the results of the game
intPlayAgain = MsgBox(vbCrLf & "The word was: " & _
    UCase(strGameWord) & vbCrLf & vbCrLf & vbCrLf & strMsgText & _
    vbCrLf & vbCrLf & vbCrLf & _
    "Would you like to play again?" , 4, cTitleBarMsg)

'Find out if the player wants to play another game
If intPlayAgain = 6 Then
    'If the answer is yes reset the following variables & start a new game
    strDisplayString = ""
    strTempStringTwo = ""
    PlayTheGame()
End If

End Function
```

Building the SplashScreen() Function

The `SplashScreen()` function is the last function in the script. As you have seen in other games in this book, this function displays some information about the game and its creator. After this function is processed, the main processing section executes the `WScript.Quit()` method, terminating the game's execution.

```
'This function displays the game splash screen
Function SplashScreen()
    MsgBox "Thank you for playing VBScript Hangman [cw] Jerry Ford 2014." & _
        vbCrLf & vbCrLf & "Please play again soon!", , cTitlebarMsg
End Function
```

The Final Result

By now you should have all the pieces and parts of the Hangman script assembled and ready for execution. Save your work and give it a shot. After you have everything working correctly, you can remove or comment out any of the extra statements that use the `MsgBox()` function to track the game's intermediate results.

After you've thoroughly tested the script, give it to somebody else to test. Ask your tester to play the game according to the rules. Then ask him to play it by *not* following the rules. Ask your tester to keep track of any problems that he experiences and to record any error messages that might appear. If an error does appear, get the player to write down exactly what steps he took so you can generate the error yourself and begin debugging it.

Summary

In this chapter, you learned how to add programming logic to your scripts to help deal with errors. This included everything from rewriting error messages to making them more user friendly to ignoring errors to creating error-handling routines that enable your scripts to recover from certain types of errors. I also provided advice that can help you prevent errors from occurring in the first place, or at least minimize their frequency. Finally, I reviewed the different ways of reporting errors that cannot otherwise be handled. On top of all this, you learned how to create the Hangman game and how to test it at various stages of development.

Challenges

1. Make the Hangman game more fun and interesting by expanding the pool of game words.
2. Improve the Hangman program by keeping track of the number of games played during a session and displaying a summary of the overall number of times the player won and lost.
3. Add logic to the Hangman game that allows you to track its use. For example, prompt the player for his name, and then write a message to either a log file or the Windows application event log each time the player plays the game.

This page intentionally left blank

10

Using the Windows Registry to Configure Script Settings

So far, all the scripts you've worked with in this book collected configuration information and input from three places: the user, within the script itself, or INI files. In this chapter, I'll show you another option for externalizing script settings by storing and retrieving configuration data using the Windows Registry. As a bonus, in this chapter's game project, I'll also demonstrate how to retrieve input data from files. Specifically, you will learn the following:

- How to review the overall organization and design of the Windows Registry
- How to programmatically create, modify, and delete Registry keys and values
- How to read data stored in external files and process it as input

Project Preview: Part 2 of the Hangman Game

In this chapter, you will enhance the Hangman game you began developing in Chapter 9, "Handling Script Errors." You'll begin by creating a new setup script that uses the Windows Registry to store the location of the folder where new external text files are stored. These text files contains lists of words that the new version of the Hangman game will use to stump the player. You'll then modify the Hangman script by removing the array that stores game words within the script and tweaking the script so that it retrieves words from the external text files. You will also modify the game to allow the player to select the category of words to play in. For example, you might want to create different text files containing words for categories such as "Foods" or "Places."

Figures 10.1 through 10.5 demonstrate the overall flow of the game from beginning to end on a computer running Windows 7.

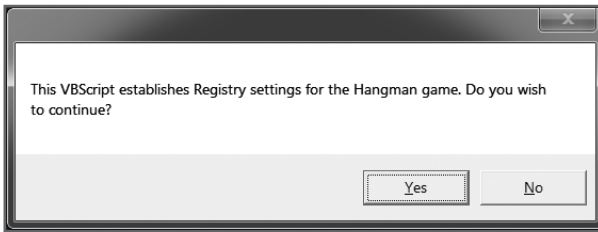


Figure 10.1 First, you need to run the Hangman setup script one time to establish the game's new Registry setting.
© 2014 Cengage Learning.



Figure 10.2 The Hangman game begins exactly as it did before, by inviting the user to play a game. © 2014 Cengage Learning.

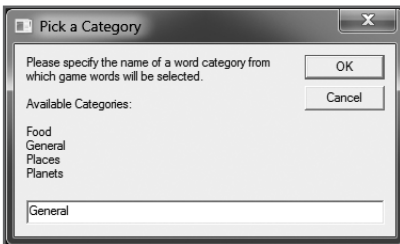


Figure 10.3 Now a list of word categories is dynamically generated, allowing the player to select a category of words to play in. © 2014 Cengage Learning.

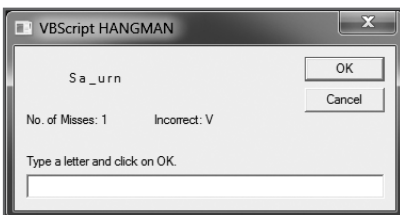


Figure 10.4 The flow of the game runs exactly as it did before, by randomly selecting a word from whichever word category the player selected. © 2014 Cengage Learning.



Figure 10.5 When the game ends, the results are displayed and the player is invited to guess a new word.
© 2014 Cengage Learning.

By the time you've finished writing the new Hangman setup script and modifying the original Hangman game, you'll have a basic understanding of the Windows Registry and what it means to access and modify its contents. You will also know how to retrieve data stored in external files to use it as input in your VBScripts.

Introducing the Windows Registry

Since the introduction of Windows 95, the Registry has been the central repository for configuration information on all Microsoft operating systems. The *Registry* is a type of built-in database that acts as a central repository for configuration settings. Windows uses it to store information that affects every component of the computer, including the following

- Windows operating system configuration settings
- Software configuration settings
- User configuration settings
- Windows services configuration settings
- Hardware configuration settings
- Software device driver configuration settings

As you can see, the Registry is used to store information regarding just about every aspect of the computer and its operation. It only makes sense, then, that by making changes to the Registry, you can configure the appearance, behavior, and operation of just about anything that affects the computer. For example, you could directly change the appearance and behavior of the Windows desktop or screensaver by making the appropriate changes to the Registry.

The Registry is so critical to the operation of Windows computers that you actually interact with it just about every day, perhaps without ever even realizing it. For example, just about every time you open the Windows Control Panel and make a change using one of its utilities or applets, you change a configuration setting stored in the Registry. In the case of the Control Panel applets, Microsoft has just made things easy for you by creating a collection of specialized graphical interfaces, each of which is designed to help you change the way the computer is configured. Alternatively, you can use the Windows Regedit Registry editor utility that comes with Windows to view and make changes to the Registry.

Because the Registry is a very reliable repository for storing and retrieving information, many application developers take advantage of it by storing their application's settings there. In a similar fashion, you can migrate settings from your VBScripts into the Registry.

You can also create VBScripts that can manipulate Registry contents to affect virtually every aspect of your computer's operation.

Definition

The Windows *Registry* is a built-in database that the operating system uses to store configuration information about itself, as well as the computer's software, hardware, and applications.

How Is the Registry Organized?

The Registry is organized as a collection of five root or parent keys, which are defined in Table 10.1. All the data in the Registry is stored in a tree-like fashion under one of these keys.

TABLE 10.1 REGISTRY ROOT KEYS

Key	Short Name	Description
HKEY_CLASSES_ROOT	HKCR	Stores information about Windows file associations
HKEY_CURRENT_USER	HKCU	Stores information about the currently logged-on user
HKEY_LOCAL_MACHINE	HKLM	Stores global computer settings
HKEY_USERS	—	Stores information about all users of the computer
HKEY_CURRENT_CONFIG	—	Stores information regarding the computer's current configuration

© Jerry Lee Ford, Jr. All Rights Reserved.

Although the Registry is logically organized into five root keys, physically it consists of many files. Files belonging to the Registry can be found in the `%systemroot%\system32\config` folder. These files include the following:

- DEFAULT
- SYSTEM
- SECURITY
- SAM
- SOFTWARE

Trick

`%systemroot%` is an environment variable created and maintained by the operating system. This variable identifies the location of the folder where Windows stores system files and folders. By default, this is `C:\Windows`.

Information managed by the Windows Registry also consists of data stored about each user of the computer. This data is stored in user profiles, which are located in either the Documents and Settings or the Users folder (depending on which version of Windows you are using) belonging to each user of the computer.

Even though the Registry has five root keys, as a VBScript programmer, chances are you'll only need to work with three of them. To help make things easier on you, Microsoft has created a short-name reference for each of these three keys. You can see these short names listed in the second column of Table 10.1.

You can, however, still interact with the remaining two keys by specifying their full names (HKEY_CURRENT_CONFIG and HKEY_USERS).

Understanding How Data Is Stored in the Registry

Data stored in the Windows Registry is organized into a hierarchy. This hierarchy consists of keys and values. A *key* is a container that holds values or other keys. *Values* are used to store actual data. All data stored in the Windows Registry has the following format:

Key : *key_type* : *value*

Key specifies the name of a Registry key. For example, to reference the Control Panel subkey, you would specify HKCU\Control Panel\. To reference the Desktop subkey, which is located under the Control Panel subkey, you would specify HKCU\Control Panel\Desktop\. Note that in both examples, the name of the last subkey is followed by the \ character. This character identifies that what is being referenced is a key and not a value.

value specifies the container used to store actual data. To reference a value, instead of the key that stores it, you must add the name of the value without the closing \ character. For example, to reference the ScreenSaveActive value stored in the Desktop subkey, you would specify HKCU\Control Panel\Desktop\ScreenSaveActive.

key_type identifies the type of data that has been stored. The Registry is capable of storing many types of data, as shown in Table 10.2.

Definition

A Registry *key* is a container that stores other Registry keys and values. You can think of a key as being akin to a folder in the Windows file system.

Definition

Within the context of the Windows Registry, a *value* represents the name of an element to which data is assigned. Therefore, a Registry value acts in many ways like a file, which is a container for storing data in a Windows file system.

TABLE 10.2 DATA TYPES SUPPORTED BY THE WINDOWS REGISTRY

Data Type	Description
REG_BINARY	Stores a binary value
REG_DWORD	Stores a hexadecimal DWORD value
REG_EXPAND_SZ	Stores an expandable string
REG_MULTI_SZ	Stores multiple strings
REG_SZ	Stores a string

Every value within the Registry falls into one of two types: named or unnamed. The most common type of value is named. Named values have been assigned an explicit name. This allows you to retrieve the data stored in the value by specifying its name. Unnamed values, as you might guess, do not have a name assigned to them. One unnamed value is stored under every key. This value represents the key's default value. In other words, it's the value that would be retrieved if you did not specify a specific value by name. Windows graphically identifies unnamed values by displaying a label of "Default." Figure 10.6 shows this on a computer running Windows 8.1.

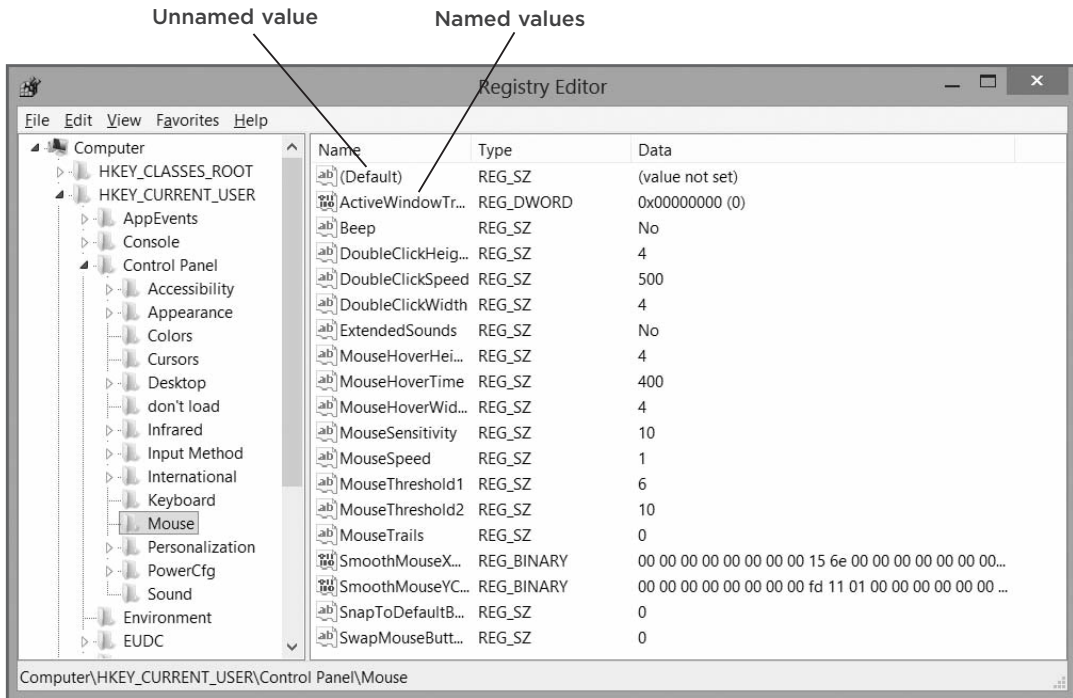


Figure 10.6 Unnamed values are assigned a label of "Default."

© 2014 Microsoft Corporation. Used with permission from Microsoft.

Accessing Registry Keys and Values

You can manually view the contents of the Windows Registry using the Regedit utility supplied with every version of Windows. (On Windows 7, type `Regedit` in the Start menu's Start Search text field and press Enter. On Windows 8.1, type `Regedit` at the Windows Start screen and press Enter.) For example, Figure 10.7 provides a high-level view of the Registry on Windows 8.1. As you can see, the five root keys are visible, and one of the root keys has been partially expanded to reveals its tree-like structure.

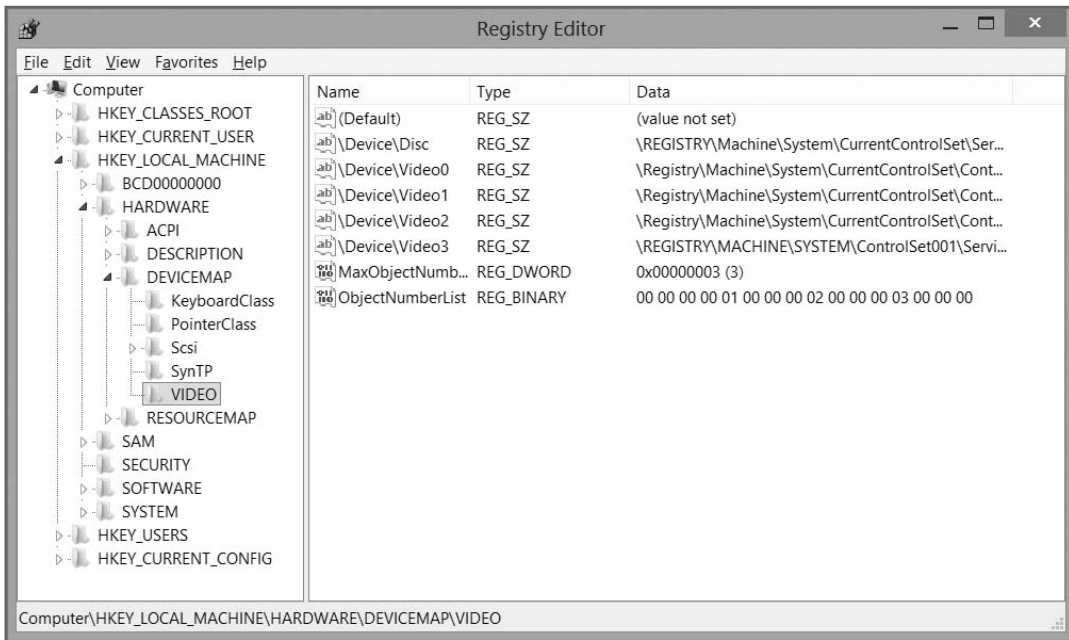


Figure 10.7 Examining the Windows Registry using the Regedit utility.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

Trap

One of the easiest ways to mess things up on a computer is to modify the Windows Registry without knowing what you're doing. The Windows Registry stores extremely critical system information. Incorrectly configuring keys and values stored in the Registry can have a disastrous effect on the computer, and could potentially prevent Windows from starting. Unless you're absolutely sure how a change will affect the Registry, don't make the change.

Creating a Key and Value to Store Script Settings

The WSH `WshShell` object supplies three methods that provide VBScript with the capability to access, modify, and delete Registry keys and values. These methods, which are demonstrated in the sections that follow, are as follows:

- **RegWrite()**. This provides the ability to create and modify a Registry key or value.
- **RegRead()**. This provides the ability to retrieve a Registry key or value.
- **RegDelete()**. This provides the ability to delete a Registry key or value.

Creating or Modifying Registry Keys and Values

The first step in creating a new Registry key and value is to instantiate the `WshShell` object within your VBScript. Then, using the `WshShell` object's `RegWrite()` method, all you have to do is provide the name of a new key or value and its location within one of the five Registry root keys. For example, the following statements create a new key called `GameKey` under the `HKEY_Current_User` root key, and then create a value called `HomeFolder` and assign it a string of `"C:\VBScript\Games"`.

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
objWshShell.RegWrite "HKCU\GameKey\HomeFolder", "C:\VBScript\Games"
```

You can later modify the Registry value by simply changing its assignment like this:

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
objWshShell.RegWrite "HKCU\GameKey\HomeFolder", "C:\MyGames\VBScript"
```

A single Registry key can be used to store any number of values. For example, the following statements establish a second value named `FileType` under the `GameKey` key and assign it a string of `".txt"`:

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
objWshShell.RegWrite "HKCU\GameKey\FileType", ".txt"
```

Accessing Information Stored in the Registry

After a Registry key and one or more values have been established, you can read them using the `WshShell` object's `RegRead()` method. For example, the following statements read and then display the value stored in the previous example:

```
Set objWshObject = WScript.CreateObject("WScript.Shell")
strResults = objWshObject.RegRead("HKCU\GameKey\FileType")
MsgBox strResults
```

Deleting Keys and Values

Now let's delete one of the two Registry values that we've just created using the `WshShell` object's `RegDelete()` method, as follows:

```
Set objWshObject = WScript.CreateObject("WScript.Shell")
objWshObject.RegDelete "HKCU\GameKey\FileType"
```

In a similar fashion, you can delete the `GameKey` key, thus deleting all the values that it stores, like this:

```
Set objWshObject = WScript.CreateObject("WScript.Shell")
objWshObject.RegDelete "HKCU\GameKey\"
```

Take note of the `\` character that follows `GameKey` in the previous statement. This character tells the `RegDelete()` method that the specified element is a Registry key and not a value.

Retrieving System Information Stored in the Registry

Now that you know the basics of reading, writing, modifying, and deleting Registry keys and values, look at the following example. In this example, the ProcessorInfo.vbs script shows how to retrieve information about the processor (that is, the CPU) of the computer on which the script is run.

```

'*****
'Script Name: ProcessorInfo.vbs
'Author: Jerry Ford
'Created: 02/13/14
'Description: This script collects CPU information about the computer that
'it is running on.
'*****

'Initialization Section
Option Explicit

Dim objWshShl, intResponse, strCpuSpeed, strCpuVendor, strCpuID

'Set up an instance of the WshShell object
Set objWshShl = WScript.CreateObject("WScript.Shell")

'Main Processing Section

'Prompt for permission to continue
intResponse = MsgBox("This VBScript gathers information about your " & _
    "processor from the Windows Registry." & vbCrLf & vbCrLf & _
    "Do you wish to continue?", 4)

'Call the function that collects CPU information
If intResponse = 6 Then
    GetProcessorInfo()
End If

WScript.Quit()

'Procedure Section

Function GetProcessorInfo()
    'Get the processor speed

```

```
strCpuSpeed = objWshShl.RegRead _
    ("HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0~MHz")

'Get the manufacturer name
strCpuVendor = objWshShl.RegRead _
    ("HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0\VendorIdentifier")

'Get processor ID information
strCpuID = objWshShl.RegRead _
    ("HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0\Identifier")

MsgBox "Speed: " & strCpuSpeed & vbCrLf & "Manufacturer: " & _
    strCpuVendor & vbCrLf & "ID: " & strCpuID
End Function
```

The script's initialization section defines its variables and instantiates the `WshShell` object. The main processing section prompts the user for confirmation before continuing and then calls the `GetProcessorInfo()` function before executing the `WScript.Quit()` method, thus terminating the script's execution.

The `GetProcessorInfo()` function performs three Registry read operations using the `WshShell` object's `RegRead()` method. Each read operation retrieves a different piece of information about the computer's processor. The function then uses the VBScript `MsgBox()` function to display a text string of the information collected about the computer's processor.

For additional examples of how to use VBScript to interact with the Windows Registry, see the "Desktop Administration" section in Appendix A, "WSH Administrative Scripting." You'll find two scripts that demonstrate how to perform desktop administration by manipulating Registry settings to configure the Windows desktop and screensaver.

Back to Part 2 of the Hangman Game

Now that you've had a review of the Windows Registry, including its overall structure and design, let's modify the Hangman game to work with the Registry. You might want to take a few minutes to review the design of the Hangman script, shown at the end of Chapter 9.

Because you already have the basic Hangman script written, all you have to do to complete this chapter's project is to focus on creating the new Hangman setup script and on modifying the parts of the original Hangman script affected by the changes. You tackle this project in two stages: creating a setup script that establishes Registry settings and updating the Hangman script to retrieve the Registry settings each time the game executes.

Creating the Setup Script

First, you'll create a VBScript called `HangmanSetup.vbs`. This script will create and store a Registry key called `Hangman` in the `HKEY_CURRENT_USER` root key (referred to in the script as `HKCU`). Within this key, a value called `ListLocation` will be created and assigned a string identifying the location where you plan to store your Hangman word files. The `HangmanSetup.vbs` script will be developed in three steps.

1. Create a new script, adding your VBScript template and defining the variables and objects used by this VBScript.
2. Set up the controlling logic in the Main Processing Section, first prompting for confirmation before continuing, and then finally calling the procedure that creates the Registry key and value.
3. Set up the Procedure Section by adding the `SetHangmanKeyAndValue()` function, which performs the actual modification of the Registry.

Defining Variables and Objects

By now, this step should be very familiar to you. Begin by copying over your VBScript template and filling in information about the new script.

```
*****
'Script Name: HangmanSetup.vbs
'Author: Jerry Ford
'Created: 02/22/14
'Description: This script configures Registry entries for the Hangman.vbs
'game.
*****

'Initialization Section

Option Explicit
```

Next, define the variables and objects required by the script. As you can see here, this is a very simple script, with only a few items that need to be defined:

```
Dim objWshShl, intResponse
Set objWshShl = WScript.CreateObject("WScript.Shell")
```

The first variable represents the `WshShell` object, and the second variable stores the user's response when asked whether he wants to make the Registry change.

Get Confirmation First

The Main Processing Section prompts the user for confirmation and then tests the results returned by the `InputBox()` function before proceeding. If the value returned is equal to 6, then the user elected to continue. Otherwise, the `WScript` object's `Quit()` method terminates script execution.

```
'Main Processing Section

'Ask for confirmation before proceeding
intResponse = MsgBox("This VBScript establishes Registry settings " & _
    "for the Hangman game. Do you wish to continue?", 4)

If intResponse = 6 Then
    SetHangmanKeyAndValue()
End If

WScript.Quit()
```

Modify the Registry

The final step in creating this script is to define a function that creates the new Registry key and value. As you saw earlier in this chapter, this operation is accomplished using the `WshShell` object's `RegWrite()` method.

Trap

When deciding what Registry key and value to create in situations like this, it's critical that you take steps to ensure that you don't accidentally overwrite an already existing key or value of the same name. Otherwise, you might accidentally disable another application or even a Windows component. In the case of this script, it's virtually certain that the key and value I defined will not be in use. However, if there is any doubt, you can add logic to your VBScripts that first check to determine whether the key and value already exist before proceeding.

```
'Procedure Section

Function SetHangmanKeyAndValue()
    objWshShl.RegWrite "HKCU\VBGames\Hangman>ListLocation", "c:\Hangman"
End Function
```

Assembling the Entire Setup Script

Now let's put the three sections of this script together; then run the script and click on Yes when prompted for confirmation. If you want to, you can use the `Regedit` utility to go behind your script and make sure that it created the new Registry key and value as expected.

You only need to run this script one time to set up a computer to play the Hangman game. However, you need to modify and rerun this script if you decide to change the location of the folder in which you plan to store your Hangman word files.

Updating the Hangman Game

In this second part of the project's development, you will modify the original Hangman script so that it retrieves from the Registry the location of the folder that stores the Hangman text files that contain the lists of words that players will be challenged to guess. You'll also add logic that enables the script to open and read the contents of the text files. To accomplish this goal, the original Hangman script needs to be modified in five steps.

1. Open the Hangman script and modify its initialization section to include additional variables and object references required to support the script's new functionality.
2. Delete the `FillArray()` function, which was responsible for retrieving a randomly selected word from an internal array, from the script.
3. Modify the `RetrieveWord()` function to call two new functions, `GetWordFileLocation()` and `SelectAWordCategory()`. Add logic that processes the text file specified by the player to randomly select a game word.
4. Create the `GetWordFileLocation()` function, which retrieves the location of the folder where the text files are stored from the Windows Registry.
5. Create the `SelectAWordCategory()` function, which presents the player with a list of word categories based on the text files that it finds in the folder.

Trick

You should make a copy of your current Hangman script and modify the copy instead of the original script. That way, if something goes wrong, you'll still have your original working version of the game to play.

Updating the Initialization Section

You need to make several changes to the Hangman script's initialization section. These include defining new variables used by new portions of the script. These variables appear in boldface in the following section:

```
'Initialization Section

Option Explicit

Const cTitlebarMsg = "VBScript HANGMAN"

Dim strChoice, strGameWord, intNoMisses, intNoRight, strSplashimage
Dim intPlayOrNot, strMsgText, intPlayAgain, strWrongGuesses
Dim strRightGuesses, blnWordGuessed, intLetterCounter
Dim strTempStringOne, strTempStringTwo, strWordLetter, strDisplayString
Dim strFlipCounter, intRandomNo, strProcessGuess, blnGameStatus
```

```
Dim strCheckAnswer, objWshShl, strGameFolder, objFsoObject, objGameFiles
Dim strSelection, strFileString, strCharactersToRemove
Dim blnValidResponse, strSelectCategory, strInputFile, strWordFile
Dim intNoWordsInFile, intLinesInFile, strWordList
```

In addition, you need to delete the following statement because the array that held the game words used in the original version of the script is no longer supported:

```
Dim astrWordList(9) 'Define an array that can hold 10 game words
```

Finally, you need to instantiate both the `FileSystemObject` object and the `WshShell` object like this:

```
'Set up an instance of the FileSystemObject object
Set objFsoObject = CreateObject("Scripting.FileSystemObject")
```

```
'Set up an instance of the WshShell object
Set objWshShl = WScript.CreateObject("WScript.Shell")
```

Methods and properties belonging to the `FileSystemObject` object are required to read and process the words stored in the game's text files. In addition, the `WshShell` object's `RegRead()` method is needed to retrieve the location of the folder where the game's text files are stored.

Removing Obsolete Statements

The next thing to do is copy and paste each of the words defined in the `FillArray()` function array into a blank Notepad file, each on its own separate line. Save the file in a folder called `Hangman` on your computer's C: drive and name the file `General.txt` (that is, save it as `C:\Hangman\General.txt`). This text file will be used to retrieve game words for the new and improved version of the game.

Once you have created and saved the `General.txt` file, delete the `FillArray()` statement shown here from the script:

```
FillArray()
```

This statement currently precedes the call to the `PlayTheGame()` function.

Then, because it is no longer needed, delete the `FillArray()` function definition that's located after the `DoYouWantToPlay()` function definition:

```
FillArray()
```

```
Function FillArray()
    'Add the words to the array
    astrWordList(0) = "AUTOMOBILE"
    astrWordList(1) = "NETWORKING"
    astrWordList(2) = "PRACTICAL"
```

```
astrWordList(3) = "CONGRESS"  
astrWordList(4) = "COMMANDER"  
astrWordList(5) = "STAPLER"  
astrWordList(6) = "ENTERPRISE"  
astrWordList(7) = "ESCALATION"  
astrWordList(8) = "HAPPINESS"  
astrWordList(9) = "WEDNESDAY"  
End Function
```

While you're at it, you might want to create one or two other text files, add a list of suitable game words to them, give them names that describe their contents, and then save them in the Hangman folder. That way, the player will have more than one category of words to choose from when playing the game.

Modifying the RetrieveWord() Function

You should begin modifying the RetrieveWord() function by first deleting all its statements and then adding the statements shown next. As you can see, I have added a number of comments to this code to explain its construction in detail.

```
'This function retrieves a randomly selected word from a word file  
Function RetrieveWord()  
  
    'Locate the folder where collections of game words are stored  
    strGameFolder = GetstrWordFileLocation()  
  
    'Get the player to select a word category  
    strSelectCategory = SelectAWordCategory(strGameFolder)  
  
    'Create the complete path and file name for the selected text file  
    strInputFile = strGameFolder & "\" & strSelectCategory  
  
    'Open the file for reading  
    Set strWordFile = objFsoObject.OpenTextFile(strInputFile, 1)  
  
    'Set this variable to zero. It represents the number of words in the file  
    intNoWordsInFile = 0  
  
    'Count the number of words in the file  
    Do while False = strWordFile.AtEndOfStream  
        'Read a line  
        strWordFile.ReadLine()  
        'Keep count of the number of words (or lines) read  
        intNoWordsInFile = intNoWordsInFile + 1
```



```
'If the loop iterates more than 50 times something is wrong
If intNoWordsInFile > 50 Then
    Exit Do
End If
Loop

'Close the file when done counting the number of words (or lines)
strWordFile.Close

'Pick a random number between 1 and the number of words in the file
Randomize
intRandomNo = FormatNumber(Int((intNoWordsInFile + 1) * Rnd),0)

'Open the file for reading
Set strWordFile = objFsoObject.OpenTextFile(strInputFile, 1)

'Skip the reading of all words prior to the randomly selected word
For intLinesInFile = 1 to intRandomNo - 1
    'Read the randomly selected word
    strWordFile.SkipLine()
Next

'Return the randomly selected word to the calling statement
RetrieveWord = strWordFile.ReadLine()

'Close the file when done
strWordFile.Close

End Function
```

Trap

Make sure any text files you create to store game words have at least one word in them. An empty text file with no words in it will result in an error.

Create the GetWordFileLocation() Function

The RetrieveWord() function calls upon the GetWordFileLocation() function, shown here, to retrieve the location of the folder where the Hangman game's text files are stored (that is, the function retrieves the information stored in the Windows Registry by the HangmanSetup.vbs script).

```
'This function retrieves the location of folder where text files are stored
Function GetstrWordFileLocation()
    'Get the folder name and path from its assigned Registry value
    GetstrWordFileLocation = _
        objWshShl.RegRead("HKCU\VBGames\Hangman\ListLocation")
End Function
```

Create the SelectAWordCategory() Function

The RetrieveWord() function also calls upon the SelectAWordCategory() function, shown next, to prompt the player to select a word category from which the game's mystery word should be randomly selected. This function takes one argument, TargetFolder, which is the location of the folder where the text files are stored. The function then displays a list of word categories based on the text files stored in the folder and prompts the player to select one. If the player fails to make a selection, the function automatically specifies the General category as the default. Again, I've added plenty of comments to the function to document its construction.

```
'This function returns a word category
Function SelectAWordCategory(TargetFolder)

    'Specify the location of the folder that stores the text files
    Set strGameFolder = objFsoObject.GetFolder(TargetFolder)
    'Get a list of files stored in the folder
    Set objGameFiles = strGameFolder.Files

    strSelection = ""

    'Loop through the list of text files
    For Each strWordList In objGameFiles
        'Build a master string containing a list of all the text files
        strFileString = strFileString & strWordList.Name

        'Remove the .txt portion of each file's file name.
        strCharactersToRemove = Len(strWordList.Name) - 4

        'Build a display string showing the category names of each text file
        strSelection = strSelection & _
            Left(strWordList.Name, strCharactersToRemove) & vbCrLf
    Next

    blnValidResponse = "False"
```

```
'Loop until a valid category strSelection has been made
Do Until blnValidResponse = "True"
  'Prompt the player to select a word category
  strChoice = InputBox("Please specify the name of a word category " & _
    "from which game words will be selected." & vbCrLf & vbCrLf & _
    "Available Categories:" & vbCrLf & vbCrLf & _
    strSelection, "Pick a Category" , "General")

  'If input is not in master string the player must try again
  If InStr(UCase(strFileString), UCase(strChoice)) = 0 Then
    MsgBox "Sorry but this is not a valid category. Please try again."
  Else
    blnValidResponse = "True"
  End If
Loop

'If the player typed nothing then specify a default word category
If Len(strChoice) = 0 Then
  strChoice = "General"
End If

'Add the .txt portion of the file name back
SelectAWordCategory = strChoice & ".txt"

End Function
```

Viewing the Completed Hangman Script

That's it! Assuming you did not skip any steps or make any typos when updating the Hangman game, you're new and improved version of the Hangman script should be ready for testing. Don't forget to test it thoroughly and to have someone else test it as well.

Summary

In this chapter, you learned how to write scripts that programmatically interact with the Windows Registry. This included reading, writing, and modifying Registry keys and values. These new programming techniques provide you with tools for externalizing script configuration information, allowing you to change your script configuration settings without having to make direct modifications to your scripts, and without having to worry about making mistakes while you do it. In addition, I showed you how to use text files as another source of data input for your VBScripts.

Challenges

1. Create a new collection of text files to increase the number of categories available to the player.
2. Add an error-handling routine to the HangmanSetup.vbs script and use it to report any problems that may occur when the script attempts to perform the `RegWrite()` method.
3. Modify HangmanSetup.vbs to display a pop-up dialog box that asks the user to agree to abide by any terms that you choose to specify to play the game. Store a value indicating whether the user has accepted the terms in the Registry. Check this value each time the Hangman game is started, allowing the user to play only if the terms have been accepted. Prompt the user to accept the terms again if they have not been accepted yet.
4. Create and store a variable in the Registry and modify Hangman.vbs to increment it every time the game is started. Use this value to track the number of games played. Check this value each time the game starts to determine whether it exceeds a value of 20. Then, if the user has not yet accepted your terms, prevent the game from running and force the user to accept your terms to play.

This page intentionally left blank

11

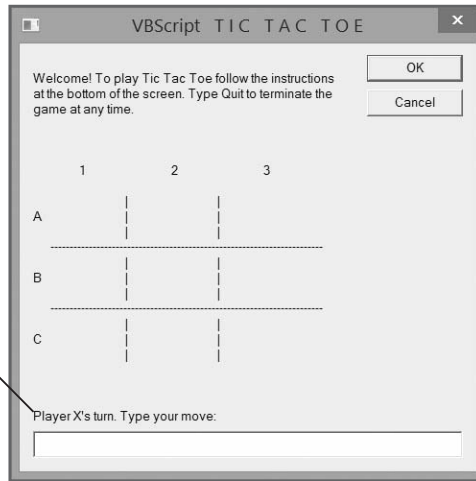
Working with Built-in VBScript Objects

To get any real work done, VBScript depends on access to objects and their associated properties and methods. So far, you have learned how to work with objects provided by the WSH and the VBScript run-time object model. Besides these collections of objects, your VBScripts have access to a small collection of built-in or core objects. Using these built-in VBScript objects, you can create scripts that react to errors, create their own custom objects, and perform a host of complex parsing operations when dissecting the contents of strings. Besides discussing VBScript's built-in objects, this chapter also assists you in creating a Tic Tac Toe game. Specifically, you will learn about the following:

- VBScript's built-in objects and collections
- How to define your own custom objects
- How to associate properties and methods with custom objects
- How to trigger events associated with custom objects
- How to perform advanced string parsing operations

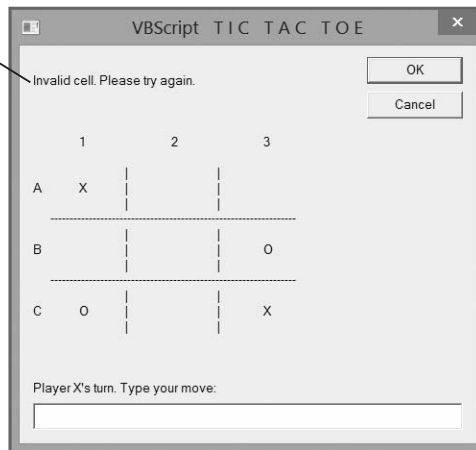
Project Preview: The Tic Tac Toe Game

In this chapter, you will develop a Tic Tac Toe game. Through the development of this game, you will learn how to create and control a two-player game. To do this, you will have to develop the logic that controls who goes next while simultaneously making sure that every player's move is valid. Figures 11.1 through 11.5 demonstrate the overall flow of the game from beginning to end, as seen on a computer running Windows 8.1.



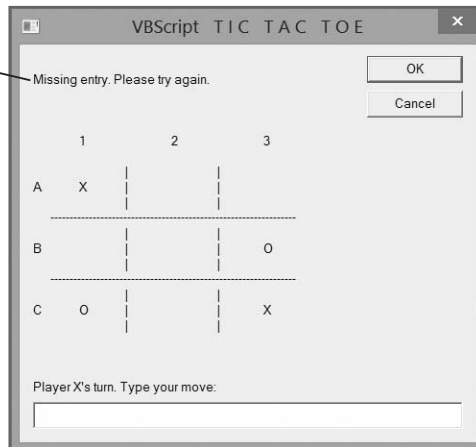
The game keeps track of each player's turn

Figure 11.1 The game begins by displaying a blank game board and prompting the first player to make a move. © 2014 Cengage Learning.



The game validates all player input to ensure that only valid moves are accepted

Figure 11.2 The game automatically updates the game board after each player's move. © 2014 Cengage Learning.



The game prevents players from accidentally missing a turn by clicking a button without first providing input

Figure 11.3 Messages that provide players with additional instruction when needed are posted at the top of the game board.

© 2014 Cengage Learning.

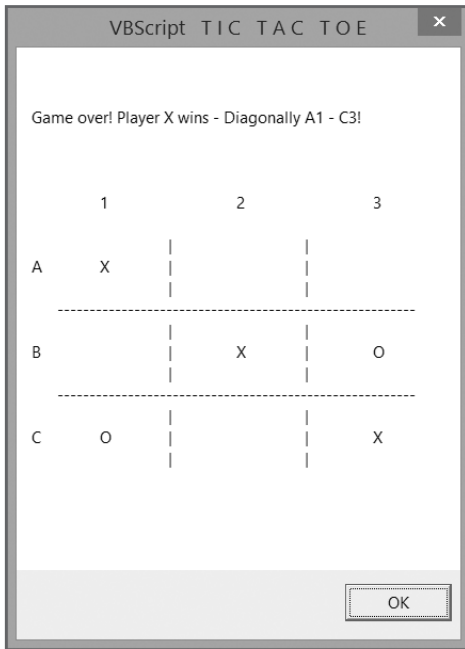


Figure 11.4 The results of each game are posted at the top of the game board. © 2014 Cengage Learning.

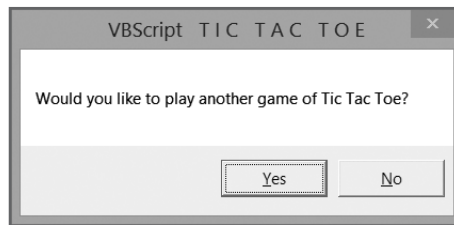


Figure 11.5 At the end of each game, players are prompted to play again. © 2014 Cengage Learning.

By the end of this chapter, you will have learned a great deal about how to work with VBScript's built-in collection of objects. You will also have developed your first multi-player VBScript game.

Leveraging VBScript's Built-in Collection of Objects

VBScript provides a small collection of built-in objects. The VBScript interpreter provides access to these objects. Therefore, they are available to any VBScript regardless of the execution host running it. These objects, though not numerous, provide VBScript with a powerful arsenal of capabilities, including the following:

- Creating customized objects complete with their own properties and methods
- Intercepting and dealing with run-time errors that your VBScripts may encounter
- Performing complex regular expression pattern matching

Table 11.1 displays a list of VBScript's built-in objects and provides a description of each object, as well as a complete listing of all the properties, methods, and events associated with the objects.

TABLE 11.1 VBSCRIPT'S COLLECTION OF BUILT-IN OBJECTS

Object	Description	Properties	Methods	Events
Class	Used to create new custom objects	None	None	Initialize, Terminate
Err	Used to retrieve information about run-time errors.	Description, HelpContext, HelpFile, Number, Source	Clear, Raise	None
Match	Used to access read-only properties associated with regular expression match strings.	FirstIndex, Length, Value	None	None
Matches	Represents a collection of regular expression Match objects.	None	None	None
RegExp	Provides the ability to work with regular expressions.	Global, IgnoreCase, Pattern	Execute, Replace, Test	None
SubMatches Collection	Used to access read-only values associated with regular expression submatch strings.	None	None	None

© Jerry Lee Ford, Jr. All Rights Reserved.

Built-in Object Properties

As you can see in Table 11.1, VBScript's built-in objects have a number of associated properties. A description of each of these properties is provided in Table 11.2.

Built-in Object Methods

Of all the VBScript built-in objects, only the Err object and the RegExp objects have methods associated with them. Methods associated with the Err object generate and clear errors, as outlined here:

- **Clear()**. This clears out Err object property settings.
- **Raise()**. This provides the ability to simulate a run-time error.

Methods associated with the RegExp object provide the ability to search strings and to replace portions of strings as outlined here:

- **Execute()**. This performs a regular expression search.
- **Replace()**. This replaces specified text during a regular expression search.
- **Test()**. This returns a Boolean value indicating whether a matching pattern is located within a string.

TABLE 11.2 BUILT-IN VBSCRIPT OBJECT PROPERTIES

Property	Description
Description	Returns error messages associated with the Err object
FirstIndex	Returns the starting character location of a substring within a string
Global	Returns a Boolean value
HelpContext	Returns the context ID associated with Help file topic
HelpFile	Retrieves the path of the specified Help file
IgnoreCase	Returns a value of True or False depending on whether a pattern search is case-sensitive
Length	Retrieves the number of characters associated with a search string match
Number	Retrieves an error number
Pattern	Returns a regular expression pattern from a search operation
Source	Returns the object name responsible for generating an error
Value	Retrieves a value from a search string match

© Jerry Lee Ford, Jr. All Rights Reserved.

Creating Custom Objects

VBScript enables you to store data in constants, variables, and arrays. VBScript supports a wide variation of variable subtypes, such as date, string, and integer. However, VBScript does not provide for strict enforcement of variable subtypes, meaning that you can store any type of value in any variable and then change the value type and value later on without raising any errors. Although all this flexibility is great, it also makes it easy to introduce errors. That's why it's best to use strict discipline when working with variables to ensure that you don't allow your scripts to mix data types. VBScript's support for arrays provides for the storage and retrieval of more complex data structures. But again, there is nothing built into VBScript to prevent you from mixing and matching data types within your arrays.

By providing you with access to the `Class` object, VBScript gives you the ability to create complex data structures in the form of custom objects. You can then define properties and methods for your custom objects. Once created, you can access custom objects just like you do any other objects. Custom objects help to improve data consistency because they give you the ability to establish validation procedures that ensure data consistency and enforce strict control over object manipulation.

Defining a Custom Object

You can create a custom object using the `Class...End Class` statement. The `Class` object provides a template for the creation of new objects. Once defined, custom objects must be instantiated just like any other object. The syntax of the `Class...End Class` statement follows:

```
Class ClassName
    Statements
End Class
```

ClassName is used to specify the name assigned to the new object. *Statements* are variables, properties, and methods that you define within the object. You define object properties for objects by adding any of the following statements within the `Class...End Class` statement:

- **Property Get.** This enables the retrieval of a value assigned to a private variable.
- **Property Let.** This enables the modification of a value assigned to a private variable.
- **Property Set.** This enables the modification of a value assigned to a public variable.

Defining Object Properties and Methods

Within the `Class...End Class` statement, variables, properties, and methods can be defined as either private or public using the `Private` and `Public` keywords. Labeling a variable, property, or method as private restricts access to only within the class. Labeling a variable, property, or method as public makes it accessible throughout a script.

When not specified, it is assumed that variables, properties, and methods are public. However, it is generally not a good idea to allow variables to be defined with a public scope. Making variables public removes the capability to strictly control their value within an object. Instead, it's better to make object variables private and then allow them to be accessed using the `Property Get` and `Property Let` statements.

To best demonstrate how all this works, let's look at an example. Here a new custom object is defined and assigned the name of `SuperHero`.

```
Class SuperHero
    Private strName

    Public Property Let Name(strIdentity)
        strName = strIdentity
    End property

    Function DisplayName()
        MsgBox "Our new hero's name is " & strName & "!"
    End Function
End Class
```

The first statement defines the object and assigns its name. The next statement defines a private variable named `strName`. The three statements that follow define an object property and make it writable by the rest of the script. The next three statements define a method for the object called `DisplayName()`. The last statement ends the definition of the `SuperHero` object.

To exercise your new object definition, create a new script and add the preceding statements to the script's procedure section. Then add the following statements to the initialization section. These statements define a variable and then use the variable to instantiate a new `SuperHero` object.

```
Dim objFirstHero
Set objFirstHero = New SuperHero
```

Once instantiated, you can assign a value to the object's `Name` property by adding the following statement to the script's main processing section:

```
objFirstHero.Name = "Captain Adventure"
```

You then can execute the object's `DisplayName()` method by adding the following statement to the main processing section:

```
objFirstHero.DisplayName()
```

Once assembled, the previous example displays the output shown in Figure 11.6 when executed on a computer running Windows 7.



Figure 11.6 Creating and instantiating a new `SuperHero` object.

© 2014 Cengage Learning.

Creating Event Procedures

Custom VBScript objects automatically support two events. These events execute as follows:

- **Class_Initialize.** This executes whenever a new instance of an object is instantiated.
- **Class_Terminate.** This executes whenever an instance of an object is destroyed.

The defining of these procedures is optional. When defined, the `Class_Initialize` procedure performs tasks such as the definition of variable default values. Similarly, the `Class_Terminate` procedure performs any cleanup that may be required after an object is no longer needed. For example, the following statements define an initialization procedure for the `SuperHero` object from the previous example.

```
Private Sub Class_Initialize
    MsgBox "In a blast of smoke and lightning another new super " & _
        "hero is born!"
End Sub
```

These statements must be added inside the Class...End Class statements. Once defined, they will automatically execute any time a new instance of the SuperHero object is established.

Trick

If your script instantiates an object that it does not need anymore, it can destroy that object instance as shown here.

```
Set objFirstHero = Nothing
```

In this example, the object instance is set equal to Nothing. This disassociates the specified object variable from an object, releasing any memory allocated to it.

The following example further demonstrates how to define a custom object complete with multiple properties and its own method and event definition:

```
'*****
'Script Name: NewObjectDemo.vbs
'Author:      Jerry Ford
'Created:     02/20/14
'Description: This script demonstrates how to create a custom object
'             with its own properties, methods, and events
'*****

'Initialization Section

Option Explicit

Dim objFirstHero  'Object variable representing the first super hero
Dim objSecondHero 'Object variable representing the second super hero

'Main Processing Section

ProcessFirstHero()
ProcessSecondHero()
WScript.Quit()

'Procedure Section
```

```
Function ProcessFirstHero()
```

```
    Set objFirstHero = New SuperHero 'Instantiate a new SuperHero object
```

```
    objFirstHero.Name = "Captain Adventure" 'Assign value to Name property
```

```
    objFirstHero.Power = "Laser Vision" 'Assign value to Power property
```

```
    objFirstHero.Weakness = "Dog Whistle" 'Assign value to Weakness property
```

```
    objFirstHero.Identity = "Bruce Tracy" 'Assign value to Identity property
```

```
    objFirstHero.DisplayIdentity() 'Execute the SuperHero object's method
```

```
End Function
```

```
Function ProcessSecondHero()
```

```
    Set objSecondHero = New SuperHero
```

```
    objSecondHero.Name = "Captain Marvelous" 'Assign value to Name property
```

```
    objSecondHero.Power = "Lightning Speed" 'Assign value to Power property
```

```
    objSecondHero.Weakness = "Blue Jello" 'Assign value to Weakness property
```

```
    objSecondHero.Identity = "Rob Denton" 'Assign value to Identity property
```

```
    objsecondHero.DisplayIdentity() 'Execute the SuperHero object's method
```

```
End Function
```

```
Class SuperHero
```

```
    Private strName, strPower, strWeakness, strIdentity 'Define variables  
                                                    'used by this class
```

```
    Public Property Let Name(strIdentity) 'Define the Name property
```

```
        strName = strIdentity
```

```
End property
```

```
    Public Property Let Power(strSuperPower) 'Define the Power property
```

```
        strPower = strSuperPower
```

```
End property
```

```
    Public Property Let Weakness(strHurtBy) 'Define the Weakness property
```

```
        strWeakness = strHurtBy
```

```
End property
```

```
    Public Property Let Identity(strSecretIdentity) 'Define the Identity
```

```
        strIdentity = strSecretIdentity 'property
```

```
End property
```

```

Function DisplayIdentity() 'This function defines the SuperHero object's
                          'DisplayIdentity() method
    MsgBox strName & vbCrLf & vbCrLf & _
        "Hero Power: " & vbTab & strPower & vbCrLf & _
        "Hero Weakness: " & vbTab & strWeakness & vbCrLf & _
        "Hero Identity: " & vbTab & strIdentity
End Function

Private Sub Class_Initialize 'This event automatically executes when
                             'the SuperHero object is instantiated
    MsgBox "In a blast of smoke and lightning another new super " & _
        "hero is born!"
End Sub
End Class

```

As the script runs, pop-up dialog boxes will be displayed. Figure 11.7 and Figure 11.8 demonstrate the execution of the `Class_Initialize` event and the object's `DisplayIdentity()` method on a computer running Windows 7.

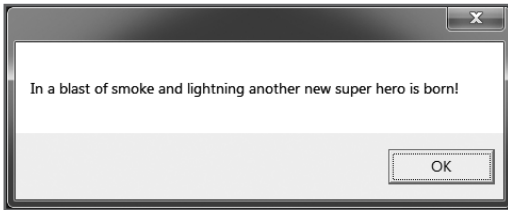


Figure 11.7 The `Class_Initialize` event occurs every time a new instance of the `SuperHero` object is established. © 2014 Cengage Learning.



Figure 11.8 The `SuperHero` object's `DisplayIdentity()` method displays the value of all properties assigned to an instance of an object. © 2014 Cengage Learning.

Working with the Err Object

The `Err` object provides access, via its properties, to information about run-time errors. For example, using the `Err` object's `Description` property, you can retrieve a string containing an error's description. Using the `Err` object's `Number` property, you can retrieve the error number associated with an error. And using the `Err` object's `Source` property, you can retrieve the name of the resource that reported the error.

The `Err` object also provides access to two methods:

- **Clear()**. The `Clear()` method clears out the properties belonging to the `Err` object. This is handy in situations when you can develop an effective error-handling routine that enables your script to recover from an error and keep running.
- **Raise()**. The `Raise()` method is equally useful, giving you the ability to simulate run-time errors so you can test your script's error-handling procedures.

For additional information and examples on how to work with the `Err` object, refer to Chapter 9, "Handling Script Errors."

Working with Regular Expressions

All remaining VBScript built-in objects and collections deal with regular expressions. A *regular expression* is a pattern consisting of characters and metacharacters. Regular expressions are used as a means of searching and replacing patterns within strings.

The first step in preparing your VBScripts to work with regular expressions is to instantiate the `RegExp` object. This object provides access to the remaining built-in VBScript objects. The `RegExp` object is instantiated as follows:

```
Dim objRegExp
Set objRegExp = New RegExp
```

The `RegExp` object provides access to the following properties:

- **Pattern**. This identifies the pattern to be matched.
- **IgnoreCase**. This contains a value of `True` or `False` depending on whether a case-sensitive search is performed.
- **Global**. This is an optional Boolean value used to indicate whether all occurrences of the specified pattern are to be replaced.

The `RegExp` object provides access to several methods, including the following:

- **Replace()**. This replaces matching string patterns.
- **Test()**. This performs a pattern search, generating a Boolean value based on whether a match is found.
- **Execute()**. This provides the ability to generate a `Matches` collection.

Replacing Matching Patterns

Using the RegExp object's `Replace()` method, you can replace matching patterns within a string. The syntax for this method is as follows:

```
RegExp.Replace(String1, String2)
```

String1 identifies the string to search and *String2* identifies the replacement string. For an illustration of how to work with the `Replace()` method, look at the following example:

```
Dim objRegExp
Set objRegExp = New RegExp

objRegExp.Pattern = "planet"
```

```
MsgBox objRegExp.Replace("A long time ago on a far away planet", "world")
```

In this example, a variable name `objRegExp` is defined and then used to instantiate a reference to the RegExp object. Next, a value of `planet` is assigned to the RegExp object's `Pattern` property to define a search pattern. Finally, the `Replace()` method is used to force the replacement of the word `planet` with the word `world`. Figure 11.9 shows the output generated when this example is run on a computer running Windows 8.1.

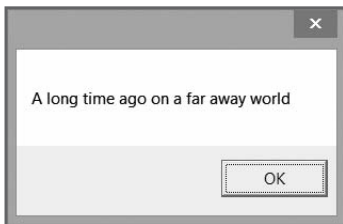


Figure 11.9 Using regular expression matching, you can substitute a portion of any string.

© 2014 Cengage Learning.

By default, the `Replace()` method replaces only the first occurrence of a match within the specified search string. However, by setting the value of the RegExp object's `Global` property to `True`, you can force the replacement of all matching patterns. To see this in action, modify the previous example as follows.

```
Dim objRegExp
Set objRegExp = New RegExp

objRegExp.Pattern = "planet"
objRegExp.Global = "True"

MsgBox objRegExp.Replace("A long time ago
on a far away planet", "world")
```

Definition

A *metacharacter* is a special character used to provide information about other characters. In the case of regular expressions, metacharacters specify how a matching pattern is to be processed.

VBScript's support for regular expressions includes the capability to define a host of complex pattern matches through the use of metacharacters. Table 11.3 lists all the metacharacters supported by VBScript.

TABLE 11.3 VBSCRIPT REGULAR EXPRESSION METACHARACTERS

Character	Description
\	Sets the next character as a special character, a back reference, a literal, or an octal escape
^	Matches the beginning of the input string
\$	Matches the end of the input string
*	Matches the preceding expression (zero or more times)
+	Matches the preceding expression (one or more times)
?	Matches the preceding expression (zero or one time)
{ <i>n</i> }	Matches exactly <i>n</i> times
{ <i>n</i> ,}	Matches a minimum of <i>n</i> times
{ <i>n</i> , <i>m</i> }	Matches a minimum of <i>n</i> times and a maximum of <i>m</i> times
.	Matches any individual character except the newline character
(<i>pattern</i>)	Matches a pattern and allows the matched substring to be retrieved from the Matches collection
<i>x y</i>	Matches <i>x</i> or <i>y</i>
[<i>xyz</i>]	Matches any of the specified characters
[^ <i>xyz</i>]	Matches any character except those specified
[<i>a-z</i>]	Matches characters specified in the range
[^ <i>a-z</i>]	Matches characters except for those specified in the range
\b	Matches on a word boundary
\B	Matches on a non-word boundary
\cx	Matches the control character specified as <i>x</i>
\d	Matches a single digit number
\D	Matches any single non-numeric character
\f	Matches the form-feed character
\n	Matches the new-line character
\r	Matches the carriage-return character
\s	Matches any white-space character (for example, space, tab, form-feed)
\S	Matches any non-white-space character
\t	Matches the tab character

TABLE 11.3 VBSCRIPT REGULAR EXPRESSION METACHARACTERS (CONTINUED)

Character	Description
<code>\v</code>	Matches the vertical tab character
<code>\w</code>	Matches any word character
<code>\W</code>	Matches any non-word character
<code>\xn</code>	Matches <i>n</i> , where <i>n</i> is a two-digit hexadecimal escape value
<code>\num</code>	Matches <i>num</i> , where <i>num</i> is a positive integer in a backward reference to captured matches
<code>\n</code>	Specifies an octal escape value or a back reference
<code>\nml</code>	Matches octal escape value <i>nml</i> where <i>n</i> is an octal digit in the range of 0-3 and <i>m</i> and <i>l</i> are octal digits in the range of 0-7
<code>\un</code>	Matches <i>n</i> , where <i>n</i> is a four-digit hexadecimal Unicode character

© Jerry Lee Ford, Jr. All Rights Reserved.

To better understand how to take advantage of metacharacters, take a look at the following example:

```
Dim objRegExp
Set objRegExp = New RegExp

objRegExp.Pattern = "[\d]"
objRegExp.Global = "True"
```

```
MsgBox objRegExp.Replace("1 years ago on a far away planet", "1000")
```

In this example, specifying the `\d` metacharacter as the value assigned to the RegExp object's `Pattern` property results in a replacement operation where any single numeric character is identified as a match. Figure 11.10 shows the output generated when you run this example. As you can see in this example, executed on a computer running Windows 8.1, the number 1 is replaced by the number 1000.



Figure 11.10 Using metacharacters enables you to perform complex substitutions.

© 2014 Cengage Learning.

Testing for Matching Patterns

The RegExp object's Test() method performs a pattern match without actually performing a replacement operation. The syntax for the Test() method is as follows:

```
RegExp.Test(string)
```

The following statements demonstrate how to use this method. In this example, the script displays one of two messages, depending on whether the string assigned to the Pattern property is found within the search string.

```
Dim objRegExp
Set objRegExp = New RegExp

objRegExp.Pattern = "planet"

If objRegExp.Test("A long time ago on a far away planet") = "True" Then
    MsgBox "The word " & objRegExp.Pattern & " was found!"
Else
    MsgBox "The word " & objRegExp.Pattern & " was not found!"
End If
```

Creating Matches Collections

Using the RegExp object's Execute() method, you can generate a Matches collection as a result of a regular expression search. The syntax of the Execute() method is as follows:

```
RegExp.Execute(string)
```

Once generated, the Matches collection is read-only. It is made up of individual Match objects. Each Match object has its own set of properties, which include the following:

- **FirstIndex.** This retrieves the starting character positions of a match within a string.
- **Length.** This returns the length of a match found within a string.
- **Value.** This retrieves the text of the match found within a string.

Once a Matches collection has been generated, you can process all the members of the collection using a loop, as demonstrated by the next example:

```
Dim objRegExp, objMatchCollection, objMatch, strStory, strDisplayMsg
Set objRegExp = New RegExp

objRegExp.Pattern = "bear"
objRegExp.Global = "True"
```

```
strStory = "Once upon a time there were three little bears. There " & _  
"was a mama bear, a papa bear, and a baby bear. There was a cousin bear too!"  
Set objMatchCollection = objRegExp.Execute(strStory)
```

```
For Each objMatch in objMatchCollection  
    strDisplayMsg = strDisplayMsg & "An instance of " & _  
        objRegExp.Pattern & " found at position " & objMatch.FirstIndex & _  
        vbCrLf  
Next
```

```
MsgBox strDisplayMsg
```

In this example, a For Each...Next loop was set up to process each Match object in the collection. The Match object's FirstIndex property was used to retrieve the starting position of each matching pattern in the search string, which was then used to generate the output shown in Figure 11.11, as seen on a computer running Windows 8.1.

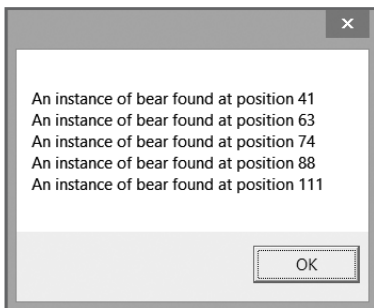


Figure 11.11 Using the RegExp object's Execute() method to generate and process the contents of a Matches collection.

© 2014 Cengage Learning.

Back to the Tic Tac Toe Game

The heart of the Tic Tac Toe game lies in the design of its game board, which is divided into three sections. Status and error messages are displayed at the top of the board to assist players when mistakes are made playing the game.

In the middle of the game's screen is an image of a traditional Tic Tac Toe board. To the right and top of the Tic Tac Toe board are letters and numbers, the coordinates of each cell that make up the game board. Players play the game by using these letters and numbers to specify what cell they want to select as their next move. Embedded within the board are variables representing each cell on the board. The values assigned to these variables are set to either X or O based on the moves made by each player as the game progresses.

The bottom of the game board is made up of an input text field that enables players to enter their moves. In addition, instruction is provided just above this text field in the form of a text message, which is used to keep track of player turns.

Designing the Game

Besides its initialization and main processing sections, the game is made up of nine functions. Each function performs a specific task. Here is a list of the script's functions along with a brief description of their associated tasks:

- **SetVariableDefaults()**. This establishes default values for various script variables.
- **ClearGameBoard()**. This resets each cell on the Tic Tac Toe game board so that it appears blank or empty.
- **ManageGamePlay()**. This controls the overall execution of the game, calling on other functions as necessary.
- **DisplayBoard()**. This displays the game board along with instructions, error messages, and any moves already made by each player.
- **DisplayGameResults()**. This displays the final results of each game, identifying who won or whether the game resulted in a tie.
- **ValidateInput()**. This ensures that players are only allowed to enter valid cell coordinates when taking their turns.
- **MarkPlayerSelection()**. This associates input provided by players with the appropriate cell coordinates on the game board.
- **SeeIfWon()**. This checks the game board to determine whether a player has won or whether the game has ended in a tie.
- **DisplaySplashScreen()**. This displays information about the script and its author.

Setting Up the Script's Template and Initialization Section

The Tic Tac Toe game begins by defining the constants and variables used by the game. Because the game uses a large number of variables, I embedded comments to the right of each variable to identify its purpose.

```

'*****
'Script Name:   TicTacToe.vbs
'Author:       Jerry Ford
'Created:      02/25/14
'Description:  This script is a VBScript implementation of the
'              Tic Tac Toe game
'*****

'Initialization Section

Option Explicit

Const cTitleBarMsg = "VBScript   T I C   T A C   T O E"

```

```

Dim A1, A2, A3, B1, B2, B3, C1, C2, C3      'Variables representing sections
                                           'of the Tic Tac Toe game board

Dim blnGameOver          'Boolean variable that determines when to end game
Dim blnPlayerTypedQuit  'Variable to track whether a player typed Quit
Dim blnStopGame         'Variable used in main processing section to
                       'determine when to stop the game
Dim blnValidCell        'Boolean variable that determines whether a player
                       'specified a valid cell
Dim intNoMoves          'Variable to keep track of the number of plays
Dim intPlayAgain        'Variable holds player response when asked to play
                       'again
Dim strNotificationMsg  'Variable to display messages to player
Dim strPlayer           'Variable to identify whose turn it is
Dim strWinner           'Variable to determine whether the game is won
Dim strPlayerInput      'Variable to hold the player's cell selection
Dim strDirection        'Variable identifies how the player won the game

```

```
blnStopGame = "False"
```

Developing the Logic for the Main Processing Section

The game's main processing section is made up of a Do...Until loop and a series of procedure calls. The loop is set up to execute until the players decide to stop playing the game, as tracked using a Boolean variable named `blnStopGame`.

```

Do Until blnStopGame = "True" 'Keep playing until players decide to stop

    SetVariableDefaults()
    ClearGameBoard()
    ManageGamePlay()

    If blnPlayerTypedQuit = "True" Then 'One of the players typed Quit
        blnStopGame = "True"
    Else 'The game is over. Ask the players whether they'd like to play again
        intPlayAgain = MsgBox("Would you like to play another game of " & _
            "Tic Tac Toe?", 4, cTitleBarMsg)

        If intPlayAgain = 7 Then 'A player clicked on No. B break out of loop
            blnStopGame = "True"
        End If
    End If

```

```
End If
```

```
Loop
```

```
DisplaySplashScreen()
```

Building the SetVariableDefaults() Function

The SetVariableDefaults() function, shown here, is straightforward. It is responsible for setting default variable values.

```
Function SetVariableDefaults() 'Establish default variable settings
    blnGameOver = "False"
    blnPlayerTypedQuit = "False"
    blnValidCell = "False"
    intNoMoves = 0
    strNotificationMsg = "Welcome! To play Tic Tac Toe follow the " & _
        "instructions at the bottom of the screen. Type Quit to terminate " & _
        "the game at any time."
    strPlayer = "X"
    strWinner = "None"
    strDirection = ""
End Function
```

Building the ClearGameBoard() Function

The ClearGameBoard() function that follows is executed to clear the contents of the game board to prepare it for a new game. As you can see, the game board is cleared by assigning a blank space to each cell of the board, thus removing any Xes or Os that may be present from a previous game.

```
Function ClearGameBoard() 'Reset the game board
    A1 = " "
    A2 = " "
    A3 = " "
    B1 = " "
    B2 = " "
    B3 = " "
    C1 = " "
    C2 = " "
    C3 = " "
End Function
```


Building the ManageGamePlay() Function

The ManageGamePlay() function that follows is controlled by a Do...Until loop that executes until the value assigned to a variable named blnGameOver is set equal to True. Within the loop, the function begins by performing a series of checks to determine whether the game has already been won. The first check looks to see whether player X has won. If this is the case, the value assigned to strNotificationMsg is set to Game over! Player X wins and the value assigned to a variable named strDirection is used to identify how the game was won. The display string assigned to strNotificationMsg is later used by the DisplayGameResults() function. The next check looks to see whether player O won. The last check looks to see whether the game has ended in a tie.

```
Function ManageGamePlay() 'Manage the overall execution of the game
    Do Until blnGameOver = "True"

        'Start by checking to see if the game has already been completed
        If strWinner = "X" Then
            strNotificationMsg = "Game over! Player X wins " & strDirection
            DisplayGameResults()
            blnGameOver = "True"
        End If

        If strWinner = "O" Then
            strNotificationMsg = "Game over! Player O wins " & strDirection
            DisplayGameResults()
            blnGameOver = "True"
        End If

        If strWinner = "Nobody" Then
            strNotificationMsg = "Game over. It's a tie!"
            DisplayGameResults()
            blnGameOver = "True"
        End If

        If blnGameOver <> "True" Then 'If game is not over display the board
            DisplayBoard()           'in order to collect next player's input
            ValidateInput()         'Validate the input

            If UCase(strPlayerInput) = "QUIT" Then 'See if a player typed Quit
                blnPlayerTypedQuit = "True"
                blnValidCell = "False"
                blnGameOver = "True"
            End If
        End If
    End Do
End Function
```

```
End If
End If

'Count the number of valid cell selections
If blnValidCell = "True" Then
    intNoMoves = intNoMoves + 1
    MarkPlayerSelection()
End If

'If all nine cells have been filled in we have a tie
If intNoMoves = 9 Then
    SeeIfWon()
    If strWinner = "None" Then
        strWinner = "Nobody"
    End If
Else
    SeeIfWon()
End If

'Time to switch player turns
If blnValidCell = "True" Then
    If strPlayer = "X" Then
        strPlayer = "O"
    Else
        strPlayer = "X"
    End If
End If

Loop
End Function
```

If the game is not over yet—that is, if `blnGameOver <> "True"`—then the game board is displayed and a player is prompted to make a move. The player's move is then validated. If the player typed `Quit`, then the script sets a number of controlling variables to indicate that the game is about to be terminated. Otherwise, the value assigned to `intNoMoves` is incremented by 1 to keep track of the game's progress and the `MarkPlayerSelection()` function is called to associate the player's move with a specific cell on the game board. Next, the value assigned to `intNoMoves` is examined. If it is equal to 9, then the game is over and the script calls on `SeeIfWon()` to ascertain whether there was a winner. If nine moves were made and no winner is identified, the game is declared a tie and the value assigned to `strWinner` is set equal to `Nobody`.

If nine moves have not been made yet, the script still executes the `SeeIfWon()` function to see whether player X or player O has managed to line up three cells in a row. Finally, the last set of statements inside the loop controls player turns by switching the value assigned to `strPlayer` to either X or O.

Building the `DisplayBoard()` Function

The `DisplayBoard()` function consists of text designed to provide a graphic-like display. Embedded inside the game board displayed by this function are nine variables named A1–A3, B1–B3, and C1–C3, each representing a different cell on the game board.

```
Function DisplayBoard() 'Display the game board
    strPlayerInput = UCase(InputBox(vbCrLf & strNotificationMsg & _
        vbCrLf & vbCrLf & vbCrLf & vbCrLf & _
        vbTab & "1" & vbTab & vbTab & "2" & vbTab & vbTab & "3" & vbCrLf & _
        vbCrLf & vbTab & vbTab & "|" & vbTab & vbTab & "|" & vbTab & _
        vbCrLf & "A" & vbTab & A1 & vbTab & "|" & vbTab & A2 & vbTab & _
        "|" & vbTab & A3 & vbCrLf & vbTab & vbTab & "|" & vbTab & vbTab & _
        "|" & vbTab & vbCrLf & "      -----" & _
        "-----" & vbCrLf & vbTab & _
        vbTab & "|" & vbTab & vbTab & "|" & vbTab & vbCrLf & "B" & vbTab & _
        B1 & vbTab & "|" & vbTab & B2 & vbTab & "|" & vbTab & B3 & _
        vbCrLf & vbTab & vbTab & "|" & vbTab & vbTab & "|" & vbTab & _
        vbCrLf & "      -----" & _
        "-----" & vbCrLf & vbTab & vbTab & "|" & _
        vbTab & vbTab & "|" & vbTab & vbCrLf & "C" & vbTab & C1 & vbTab & _
        "|" & vbTab & C2 & vbTab & "|" & vbTab & C3 & vbCrLf & vbTab & _
        vbTab & "|" & vbTab & vbTab & "|" & vbTab & vbCrLf & vbCrLf & _
        vbCrLf & vbCrLf & "Player " & strPlayer & _
        "'s turn. Type your move:", cTitleBarMsg))
End Function
```

Whenever this function is called, it displays the game board, including the values assigned to each of its nine embedded variables (as either Xes or Os). This gives the game the capability to dynamically display each move made as the game progresses.

Building the `DisplayGameResults()` Function

The `DisplayGameResults()` function that follows is responsible for displaying the final results of the game. The dialog box generated by this function is not much different from the dialog box created by the `DisplayBoard()` function, except that this function uses the `MsgBox()` function in place of the `InputBox()` function. The `MsgBox()` function is more appropriate for this function because it can be used to display a dialog box with a single OK button, whereas using the `InputBox()` function would have resulted in the unnecessary display on a text input field at the bottom of the dialog box.

```

Function DisplayGameResults() 'Game is over. Display the results.
  MsgBox vbCrLf & _
    strNotificationMsg & _
    vbCrLf & vbCrLf & vbCrLf & vbCrLf & _
    vbTab & "1" & vbTab & vbTab & "2" & vbTab & vbTab & "3" & vbCrLf & _
    vbCrLf & vbTab & vbTab & "|" & vbTab & vbTab & "|" & vbTab & _
    vbCrLf & "A" & vbTab & A1 & vbTab & "|" & vbTab & A2 & vbTab & _
    "|" & vbTab & A3 & vbCrLf & vbTab & vbTab & "|" & vbTab & vbTab & _
    "|" & vbTab & vbCrLf & _
    "      -----" & _
    vbCrLf & vbTab & vbTab & "|" & vbTab & vbTab & _
    "|" & vbTab & vbCrLf & "B" & vbTab & B1 & vbTab & "|" & vbTab & _
    B2 & vbTab & "|" & vbTab & B3 & vbCrLf & vbTab & vbTab & "|" & _
    vbTab & vbTab & "|" & vbTab & vbCrLf & "      -----" & _
    "-----" & vbCrLf & _
    vbTab & vbTab & "|" & vbTab & vbTab & "|" & vbTab & vbCrLf & _
    "C" & vbTab & C1 & vbTab & "|" & vbTab & C2 & vbTab & "|" & vbTab & _
    C3 & vbCrLf & vbTab & vbTab & "|" & vbTab & vbTab & "|" & vbTab & _
    vbCrLf & vbCrLf & vbCrLf & vbCrLf, , cTitleBarMsg
End Function

```

Building the ValidateInput() Function

The `ValidateInput()` function, shown here, uses a `Select Case` statement to process the input provided by players to determine whether it is valid. Input is valid only if it is provided in the form of a valid cell range. Next, the function checks to make sure the player did not accidentally click OK before entering a move. The last validation test performed by this function checks to make sure that the cell specified by the player has not already been selected. This is done by checking to see whether the value assigned to the cell is anything other than a blank space. If it is, then regardless of whether a value of X or O has been assigned, the cell is not available.

```

Function ValidateInput() 'Run tests to determine if player input is valid

  Select Case strPlayerInput 'Ensure a valid cell was specified
    Case "A1"
      blnValidCell = "True"
    Case "A2"
      blnValidCell = "True"
    Case "A3"
      blnValidCell = "True"

```

```
Case "B1"
    blnValidCell = "True"
Case "B2"
    blnValidCell = "True"
Case "B3"
    blnValidCell = "True"
Case "C1"
    blnValidCell = "True"
Case "C2"
    blnValidCell = "True"
Case "C3"
    blnValidCell = "True"
Case Else
    blnValidCell = "False"
    strNotificationMsg = "Invalid cell. Please try again."
End Select

If strPlayerInput = "" Then 'Player must type something
    strNotificationMsg = "Missing entry. Please try again."
    blnValidCell = "False"
End If

'Check each cell to make sure that it has not already been selected
If strPlayerInput = "A1" Then
    If A1 <> " " Then
        blnValidCell = "False"
        strNotificationMsg = "Invalid entry. Cell already selected. " & _
            "Please try again."
    End If
End If

If strPlayerInput = "A2" Then
    If A2 <> " " Then
        blnValidCell = "False"
        strNotificationMsg = "Invalid entry. Cell already selected. " & _
            "Please try again."
    End If
End If
```

```
If strPlayerInput = "A3" Then
  If A3 <> " " Then
    blnValidCell = "False"
    strNotificationMsg = "Invalid entry. Cell already selected. " & _
      "Please try again."
  End If
End If
```

```
If strPlayerInput = "B1" Then
  If B1 <> " " Then
    blnValidCell = "False"
    strNotificationMsg = "Invalid entry. Cell already selected. " & _
      "Please try again."
  End If
End If
```

```
If strPlayerInput = "B2" Then
  If B2 <> " " Then
    blnValidCell = "False"
    strNotificationMsg = "Invalid entry. Cell already selected. " & _
      "Please try again."
  End If
End If
```

```
If strPlayerInput = "B3" Then
  If B3 <> " " Then
    blnValidCell = "False"
    strNotificationMsg = "Invalid entry. Cell already selected. " & _
      "Please try again."
  End If
End If
```

```
If strPlayerInput = "C1" Then
  If C1 <> " " Then
    blnValidCell = "False"
    strNotificationMsg = "Invalid entry. Cell already selected. " & _
      "Please try again."
  End If
End If
```

```
If strPlayerInput = "C2" Then
  If C2 <> " " Then
    blnValidCell = "False"
    strNotificationMsg = "Invalid entry. Cell already selected. " & _
      "Please try again."
  End If
End If
```

```
If strPlayerInput = "C3" Then
  If C3 <> " " Then
    blnValidCell = "False"
    strNotificationMsg = "Invalid entry. Cell already selected. " & _
      "Please try again."
  End If
End If
```

```
End Function
```

Building the MarkPlayerSelection() Function

The MarkPlayerSelection() function that follows is responsible for associating the player's move with the appropriate cell on the game board. It does this by assigning the value stored in strPlayer to the specified cell. Remember, the value assigned to strPlayer is either an X or an O, depending on whose turn it is.

```
Function MarkPlayerSelection() 'Mark an X or O in the appropriate cell
  If strPlayerInput = "A1" Then
    A1 = strPlayer
  End If
  If strPlayerInput = "A2" Then
    A2 = strPlayer
  End If
  If strPlayerInput = "A3" Then
    A3 = strPlayer
  End If
  If strPlayerInput = "B1" Then
    B1 = strPlayer
  End If
  If strPlayerInput = "B2" Then
    B2 = strPlayer
  End If
```

```
If strPlayerInput = "B3" Then
    B3 = strPlayer
End If
If strPlayerInput = "C1" Then
    C1 = strPlayer
End If
If strPlayerInput = "C2" Then
    C2 = strPlayer
End If
If strPlayerInput = "C3" Then
    C3 = strPlayer
End If
End Function
```

Building the SeeIfWon() Function

The SeeIfWon() function, shown here, performs a series of eight tests to see whether the game has been won by one of the players. These tests include checking all three cells in each row and in each column to see whether the same player has selected them. The function also checks diagonally to see whether there is a winner.

```
Function SeeIfWon()

    'Check across the first row
    If A1 = strPlayer Then
        If A2 = strPlayer Then
            If A3 = strPlayer Then
                strWinner = strPlayer
                strDirection = "- First row across!"
            End If
        End If
    End If

    'Check across the second row
    If B1 = strPlayer Then
        If B2 = strPlayer Then
            If B3 = strPlayer Then
                strWinner = strPlayer
                strDirection = "- Second row across!"
            End If
        End If
    End If

End Function
```



```
'Check across the third row
If C1 = strPlayer Then
  If C2 = strPlayer Then
    If C3 = strPlayer Then
      strWinner = strPlayer
      strDirection = "- Third row across!"
    End If
  End If
End If

'Check the first column
If A1 = strPlayer Then
  If B1 = strPlayer Then
    If C1 = strPlayer Then
      strWinner = strPlayer
      strDirection = "- First column down!"
    End If
  End If
End If

'Check the second column
If A2 = strPlayer Then
  If B2 = strPlayer Then
    If C2 = strPlayer Then
      strWinner = strPlayer
      strDirection = "- Second column down!"
    End If
  End If
End If

'Check the third column
If A3 = strPlayer Then
  If B3 = strPlayer Then
    If C3 = strPlayer Then
      strWinner = strPlayer
      strDirection = "- Third column down!"
    End If
  End If
End If
```

```
'Check diagonally
If A1 = strPlayer Then
  If B2 = strPlayer Then
    If C3 = strPlayer Then
      strWinner = strPlayer
      strDirection = "- Diagonally A1 - C3!"
    End If
  End If
End If

'Check the diagonally
If A3 = strPlayer Then
  If B2 = strPlayer Then
    If C1 = strPlayer Then
      strWinner = strPlayer
      strDirection = "- Diagonally C1 - A3!"
    End If
  End If
End If

End Function
```

Building the DisplaySplashScreen() Function

The script's final function, `DisplaySplashScreen()`, is shown here. This function displays information about the script and its author and then terminates the script's execution by using the `WScript` object's `Quit()` method.

```
Function DisplaySplashScreen() 'Display splash screen and terminate game
  MsgBox "Thank you for playing Tic Tac Toe" & _
    "© Jerry Ford 2014." & vbCrLf & vbCrLf & "Please play again " & _
    "soon!", 4144, cTitleBarMsg
  WScript.Quit()
End Function
```

The Final Result

That's it. You have all the necessary pieces to assemble the game. Once you have keyed everything, run through the Tic Tac Toe game a few times to be sure you haven't accidentally made a few typos when keying in the script. After you have everything working just right, go out and get a friend to play with and show off what you have learned.

Summary

In this chapter, you learned how to work with built-in VBScript objects. This included learning how to create custom objects with their own unique sets of properties and methods. You also learned how to trigger events associated with custom objects. On top of all this, you learned how to perform complex parsing operations by working with the RegExp object, and you created your first multi-player VBScript game.

Challenges

1. Enhance the Tic Tac Toe game by adding options that allow the players to get help.
2. If you have a website, considering modifying the game's closing splash screen to display its address.
3. Try making a computerized version of this game where a single player goes head to head against the computer.
4. Add logic that keeps track of the total number of games played and display this information, along with the total number of games won by each player, at the end of the final game.

12

Combining Different Scripting Languages

In this chapter, you will learn how to develop a new type of script, called a Windows Script File, which enables you to combine VBScript with one or more other WSH-supported scripting languages to create a single executable script. Doing so enables you to create scripts that can take advantage of the strengths of each individual scripting language. Specifically, in this chapter I'll demonstrate how to develop Windows Script Files that combine VBScript and JScript. Along the way, you'll be introduced to the Extensible Markup Language, or XML, which allows different scripting languages to be combined into Windows Script Files. Specifically, you will learn the following:

- How to combine VBScript with another scripting language to create Windows Script Files
- How XML is used to format Windows Script Files
- A sneak peek at the JScript scripting language
- How to execute Windows Script Files

Project Preview: The VBScript Game Console

In this chapter's project, you will learn how to create Windows Script Files that combine VBScript with a little bit of JScript to create a game console for all your VBScript games. Once started, the game console displays a dynamically created numbered list of your VBScript games and enables the user to choose which game to play by either typing the name of the game or typing its assigned number.

When started, the game console appears in the upper-left corner of the display area. As games are selected, they will appear in the middle of the screen. This keeps the game console handy without making it intrusive. Figures 12.1 through 12.5 demonstrate the overall operation of the game console from beginning to end when executed on a computer running Windows 7.

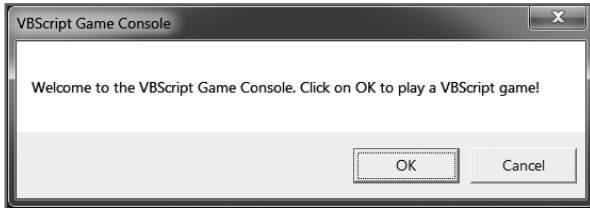


Figure 12.1 A JScript that displays the game console's initial splash screen is executed. © 2014 Cengage Learning.

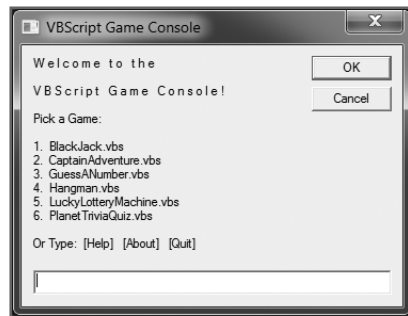


Figure 12.2 The core logic for the game console is provided by a VBScript, which is responsible for displaying and controlling the execution of your VBScript games.

© 2014 Cengage Learning.

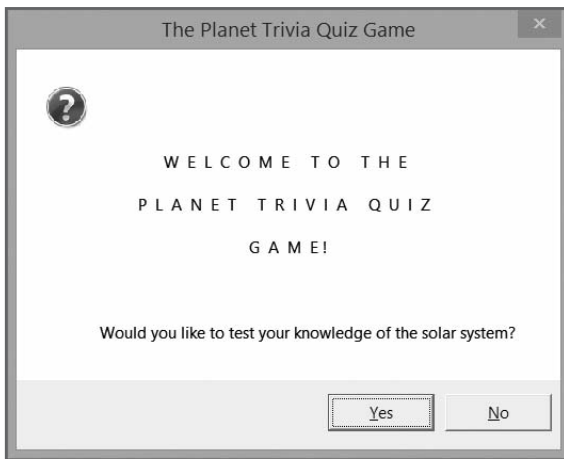


Figure 12.3 The game console remains tucked away in the corner while the player enjoys playing your VBScript game. © 2014 Cengage Learning.

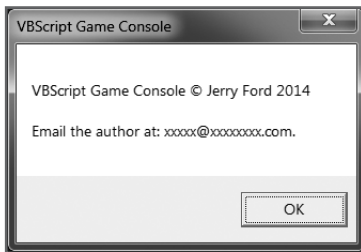


Figure 12.4 By selecting the About option, the user can get more information about the game console and its author. © 2014 Cengage Learning.

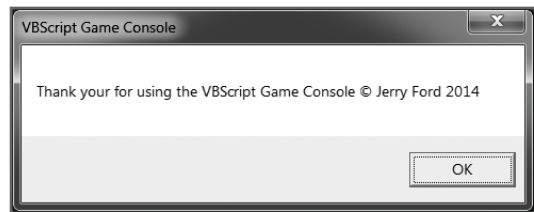


Figure 12.5 When the game console is finally closed, another JScript is run to display a closing splash screen.

© 2014 Cengage Learning.

Introducing Windows Script Files

One of the strengths of the WSH is that it supports a number of different scripting languages, including VBScript, JScript, Perl, Python, and REXX. Microsoft automatically equips the WSH with VBScript and JScript. Third-party software developers provide support for the other scripting languages. Besides executing scripts written in any of these scripting languages, the WSH enables you to put any combination of these languages into a single script file known as a Windows Script File.

Extensible Markup Language, or XML, provides the glue for combining different scripts into a Windows Script File. In this chapter, I'll cover some of the more commonly used WSH-supported XML statements. However, there simply is not enough space available in this book to completely cover every single XML element supported by the WSH.

Using XML, you specify the components that make up Windows Script Files. For example, you use XML to mark the locations within Windows Script Files where individual scripts (written in scripting languages such as VBScript or JScript) are embedded. Windows Script Files are saved as plain-text files with a .wsf file extension and can be created using any plain-text or script editor.

XML is case-sensitive and imposes a strict set of rules on the format of Windows Script Files. For example, within the context of the WSH, most XML tags occur in pairs with one opening and one closing tag. Failure to include a matching closing tag will result in an error.

Trick

WSH supports XML version 1.0. Version 1.0 supports both uppercase and lowercase spelling of tag elements. Lowercase spelling is preferred, and I recommend that you use it. That way, if lowercase spelling becomes a requirement in a future version of XML, you will not have to retrofit your Windows Scripts Files for them to continue to run.

Definition

A *Windows Script File* is a type of script that allows multiple scripts, written in any WSH-supported scripting language, to be combined to create a single script.

Definition

Extensible Markup Language, or XML, is a language similar in design and syntax to HTML. It is used within the context of the WSH to define the structure of Windows Script Files.

Examining WSH-Supported XML Tags

XML represents an extensive and powerful multipurpose markup language. XML is therefore often used in many environments, including the WSH. In this chapter, I'll introduce you to a number of commonly used XML tags, and I'll provide examples of how they're used to build Windows Script Files.

To begin, take a look at Table 12.1, which shows the XML tags that you'll see demonstrated in this chapter's examples.

TABLE 12.1 XML TAGS COMMONLY USED
IN WINDOWS SCRIPT FILES

Tag	Description
<code><?job ?></code>	Enables or disables error handling and debugging for a specified job
<code><?xml ?></code>	Specifies the Windows Script File's XML level
<code><comment> </comment></code>	Embeds comments within Windows Script Files
<code><script> </script></code>	Identifies the beginning and ending of a script within a Windows Script File
<code><job> </job></code>	Identifies the beginning and ending of a job inside a Windows Script File
<code><package> </package></code>	Enables multiple jobs to be defined within a single Windows Script File
<code><resource> </resource></code>	Defines static data (constants) that can be referenced by scripts within a Windows Script File

© Jerry Lee Ford, Jr. All Rights Reserved.

Using the `<?job ?>` Tag

The `<?job ?>` tag allows you to enable or disable error reporting and debugging within your Windows Script Files. The use of this tag is optional. Unlike most tags, the `<?job ?>` tag does not have a closing tag. The syntax for this tag is as follows:

```
<?job error="flag" debug="flag" ?>
```

Both `error` and `debug` are Boolean values. By default, both are set equal to `false`. Setting `error="true"` turns on error reporting, thus allowing syntax and run-time error messages to be reported. Setting `debug="true"` turns on debugging for Windows Script Files, allowing them to start the Windows script debugger.

Trick

To take advantage of the `<?job ?>` tag's `debug` capability, you'll need to install the Microsoft Windows script debugger utility. This utility is designed to assist programmers in debugging script errors. To learn more about this utility, check out [msdn.microsoft.com/en-us/library/ms875975\(v=exchg.65\).aspx](http://msdn.microsoft.com/en-us/library/ms875975(v=exchg.65).aspx).

The following example demonstrates how to use the `<?job ?>` tag within a Windows Script File:

```
<job>
  <?job error="true" debug="true"?>
  <script language="VBScript">
    MsgBox "Error handling and debugging have been enabled. "
  </script>
</job>
```

As you can see, both error reporting and script debugging have been enabled.

Using the `<?xml ?>` Tag

The `<?xml ?>` tag is used to specify the version of XML required to support a Windows Script File. As with the `<?job ?>` tag, the use of this tag is optional.

When used, the `<?xml ?>` tag must be the first tag defined as the first statement in the Windows Script File. The syntax for the `<?xml ?>` tag is as follows:

```
<?xml version="version" standalone="DTDFlag" ?>
```

`version` specifies the version of XML required to support the Windows Script File. The current version is 1.0. `standalone` is used to specify an external document type definition, which is a feature not currently supported by the WSH. You can include it if you want, but you'll have to specify it as having a value of Yes, and it will be ignored.

When used, the `<?xml ?>` tag enforces stricter interpretation of all XML statements. It also enforces case-sensitivity while requiring that all values be specified within either single or double quotes. Omitting this tag provides for a less restrictive syntax. Let's look at the following example, which demonstrates the placement of an `<?xml ?>` tag at the beginning of a small Windows Script File:

```
<?xml version="1.0" standalone="yes" ?>
<job>
  <?job error="true" debug="true"?>
  <script language="VBScript">
    MsgBox "Error handling and debugging have been enabled. "
  </script>
</job>
```

The `<comment>` and `</comment>` Tags

You can document the XML statements used within your Windows Script Files using the XML `<comment>` and `</comment>` tags. Using these tags, you can spread comments over multiple lines. The syntax for the `<comment>` and `</comment>` tags is as follows:

```
<comment> Comment Text </comment>
```

The following example demonstrates the use of the XML `<comment>` and `</comment>` tags.

```
<?xml version="1.0" standalone="yes" ?>
<job>
  <?job error="true" debug="true"?>
  <comment>The following VBScript displays an information message</comment>
  <script language="VBScript">
```



```
    MsgBox "Error handling and debugging have been enabled File."  
</script>  
</job>
```

The <job> and </job> Tags

To embed a script into a Windows Script File, you must first define a pair of root tags. The <job> and </job> tags provide one type of root tag pair.

All Windows Script Files are composed of at least one job. The beginning and ending of a job are identified by the <job> and </job> tags. The syntax for the tags is as follows:

```
<job [id=JobID]>  
    . . .  
</job>
```

When only one job is defined within a Windows Script File, the `id=JobID` parameter can be omitted from the opening <job> tag. However, if two or more jobs are defined within a single Windows Script File, each must be given a unique ID assignment. This assignment allows you to execute any job within the Windows Script File.

The following example shows a Windows Script File that contains a single job. The job itself is made up of two different scripts:

```
<?xml version="1.0" standalone="yes" ?>  
<job>  
    <?job error="true" debug="true"?>  
  
    <comment>The following VBScript displays an information message</comment>  
    <script language="VBScript">  
        MsgBox "VBScript has displayed this message."  
    </script>  
  
    <comment>The following JScript displays an information message</comment>  
    <script language="JScript">  
        WScript.Echo("JScript has displayed this message.");  
    </script>  
</job>
```

If you double-click the file, you'll see two pop-up dialog boxes appear. Figure 12.6 and Figure 12.7 show how these dialog boxes appear when this example is executed on a computer running Windows 8.1. The VBScript generates the first pop-up dialog box and the JScript generates the second. To place more than one job within a Windows Script File, you must use the <package> and </package> tags, which I'll explain next.

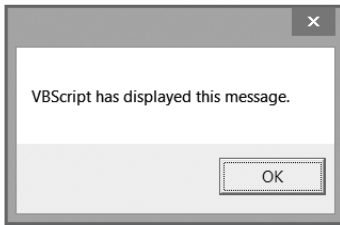


Figure 12.6 A pop-up dialog box displayed by the Windows Script File's VBScript. © 2014 Cengage Learning.

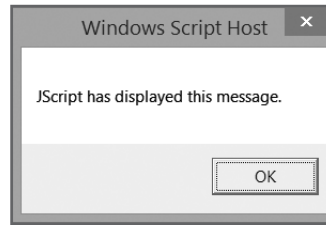


Figure 12.7 A pop-up dialog box displayed by the Windows Script File's JScript. © 2014 Cengage Learning.

The <package> and </package> Tags

To place more than one job within a Windows Script File, you must enclose the jobs within the <package> and </package> tags. The syntax for these tags is as follows:

```
<package>
. . .
</package>
```

To understand their use, look at the following example:

```
<?xml version="1.0" standalone="yes" ?>
```

<package>

```
<comment>The following job contains a VBScript and a JScript</comment>
<job id="job1">
  <?job error="true" debug="true"?>
  <comment>The following VBScript displays an information message</comment>
  <script language="VBScript">
    MsgBox "A VBScript has displayed this message."
  </script>
  <comment>The following JScript displays an information message</comment>
  <script language="VBScript">
    WScript.Echo "A JScript has displayed this message."
  </script>
</job>
```

```
<comment>The following job contains one VBScript</comment>
<job id="job2">
  <script language="VBScript">
    MsgBox "A second VBScript has displayed this message."
  </script>
</job>
```

</package>

In this Windows Script File, the `<package>` and `</package>` tags are used to define two jobs. The first job is assigned an ID of `job1`, and the second job has been assigned an ID of `job2`.

The `<resource>` and `</resource>` Tags

You have already learned about the advantages of defining constants within your VBScripts. However, these constants are available only within the script that defines them. Using the XML `<resource>` and `</resource>` tags, you can define constants within your Windows Script Files that can then be accessed by every script located within the same job. Therefore, when used, these tags must be placed within the `<job>` and `</job>` tags.

The syntax of the `<resource>` and `</resource>` tags is as follows:

```
<resource id="resourceID" . . .  
</resource>
```

`resource id` specifies the name of the constant whose value is then assigned when you type it between the opening and closing tags, as demonstrated in the following example:

```
<job>  
  <resource id="cTitleBarMsg">TestScript.wsh</resource>  
  <script language="VBScript">  
    Set objWshShl = WScript.CreateObject("WScript.Shell")  
    MsgBox "Greetings", , getResource("cTitleBarMsg")  
  </script>  
</job>
```

The `<script>` and `</script>` Tags

You've already seen the `<script>` and `</script>` tags in action a number of times in this chapter. They are used to mark the beginning and ending of individual scripts embedded within jobs in Windows Script Files. The syntax for these tags is as follows:

```
<script language="language" [src="externalscript"]>  
  . . .  
</script>
```

`language` specifies the scripting language used to create a script. The `src` argument is used to specify an optional reference to an external script. If used, the external script is called and executed just as if it were embedded within the Windows Script File.

In the real world, time is money. Saving time during script development means that you can get more done in less time. One way experienced programmers save time is by creating reusable code. You can use Windows Script Files to save development time by externalizing scripts that perform common tasks. That way, you can set up a reference to those scripts in any number of Windows Script Files without having to reinvent the wheel. In addition, you'll save yourself a lot of maintenance work because if you ever need to modify a commonly used external script, you'll only have to make the change to the script once. This is preferable to making the same change over and over again in any scripts where you embedded copies of the script.

Let's look at an example of how to create a Windows Script File that includes both an embedded VBScript and one that is externally referenced. As you can see, the following Windows Script File includes a reference to two VBScripts:

```
<job>
  <script language="VBScript">
    MsgBox "This message is being displayed by an embedded VBScript"
  </script>
  <script language="VBScript" src="TestScript.vbs" />
</job>
```

The embedded VBScript simply displays a text message stating that it has executed. Similarly, the external VBScript might consist of a single statement that uses the `MsgBox()` function to display a similar message:

```
MsgBox "This message is being displayed by an external VBScript"
```

If you create both of these scripts and then double-click the Windows Script File, you'll see that both scripts will execute and display their pop-up dialog box in sequence.

Executing Your Windows Script Files

As you know, you can run a Windows Script File by double-clicking it. When started this way, the Windows Script File runs the first job that has been defined. If more than one job has been defined within the Windows Script File, you can execute any job by running the script from the Windows command prompt and specifying the job's ID, as demonstrated next. Of course, every script contained within the job that is run will be executed.

Let's look at some examples of how to run Windows Script Files from the Windows command prompt. To run the first script in a Windows Script File that contains two jobs, just type the name of an execution host followed by the name of the Windows Script File:

```
cscript TestWsfScript.wsf
```

If the two jobs in the Windows Script Files have been assigned job IDs of `job1` and `job2`, you can selectively execute either job by specifying its ID:

```
wscript TestWsfScript.wsf //job:job2
```

Back to the VBScript Game Console

The VBScript game console project is actually a Windows Script File designed to display a list of VBScript games that is dynamically generated based on the contents of a game folder. Once started, the VBScript game console gives the user easy access to any VBScript games that you have stored in the game folder.

The VBScript game console is actually made up of three different scripts, two written in JScript and the other in VBScript. The rest of this chapter explains how the VBScript game console is built.

Designing the Game Console

The VBScript game console consists of three scripts written using two different WSH-supported scripting languages, VBScript and JScript. Because JScript is a full-featured scripting language in its own right, I won't be able to go into great detail about its syntax or structure. However, I've tried to organize this Windows Script File in such a way as to ensure that the JScript you see is not overly complex. Hopefully, given the comments I have added to each script, you will be able to understand what's happening in each JScript.

The VBScript game console will be created in four stages:

- **Stage 1:** Create a WSF and add the XML statements required to define the script's structure.
- **Stage 2:** Create the first JScript, which will be responsible for displaying the VBScript game console's initial splash screen and determining whether the user wants to open the console.
- **Stage 3:** Design the Windows Script File's VBScript, wherein the logic that controls the operation of the game console is stored.
- **Stage 4:** Create a second JScript, which is responsible for displaying the VBScript game console's closing splash screen.

Using XML to Outline the Script's Structure

The first stage in developing the VBScript game console involves two activities. First, create a new file and save it with a `.wsf` file extension, thus creating a new Windows Script File. Second, add the XML tags required to outline the overall structure of the Windows Script File, like this:

```
<package>  
  <comment>This .WSF file builds a VBScript game console</comment>  
  <job>  
    <resource id="cTitlebarMsg">VBScript Game Console</resource>
```

```

    <script language="JScript"> </script>
    <script language="VBScript"> </script>
    <script language="JScript"> </script>
    <script language="JScript"> </script>
  </job>
</package>

```

The `<package>` and `</package>` tags were not required because this Windows Script File contains only one job. I added them anyway, just in case I ever decide to expand the script by adding another job. For example, if script configurations are ever migrated to the Windows Registry, it might be helpful to define a second job to the Windows Script File specifying a setup script.

The `<comment>` and `</comment>` tags were added to help document the function of the Windows Script File. The opening `<job>` and closing `</job>` tags define the Windows Script File's only job. As only one job was defined, I did not bother to add the `<job>` tag's `id` attribute. Finally, three separate sets of `<script>` and `</script>` tags have been created, marking the location at which each script that makes up the Windows Script File will be placed.

Writing the First JScript

Because the focus of this book is on VBScript as a WSH scripting language and not on JScript, I'm not going to attempt to explain in detail the following JScript. This script is relatively simple, and you should be able to tell what's going on by looking at the comments embedded within the script itself.

```

/*****
//Script      Name: N/A
//Author:     Jerry Ford
//Created:    03/01/14
//Description: Display the WSF's initial splash screen
/*****

//Initialization Section

var objWshShl = WScript.CreateObject("WScript.Shell");
var strWelcome;
var strInstructions;
var intResults;
var intReply;
var strTitleBarMsg;

strWelcome = "Welcome to the VBScript Game Console. ";
strInstructions = "Click on OK to play a VBScript game!";

```

```
//Main Processing Section

//Verify that the user wants to open the VBScript game console
intReply = DisplayInitialSplashScreen();

//intReply will be set equal to 2 if the user clicks on Cancel
if (intReply == 2) {
    //Close the VBScript game console
    WScript.Quit();
}

//Procedure Section

//This procedure prompts the user for confirmation
function DisplayInitialSplashScreen() {
    strTitleBarMsg = getResource("cTitlebarMsg");

    //Display pop-up dialog box using the WshShell object's Popup() method
    intResults = objWshShl.Popup(strWelcome +
        strInstructions, 0, strTitleBarMsg, 1);

    //Return the result to the calling statement
    return intResults
}
```

Trick

One way to develop each of the three scripts used in this Windows Script File is to create each script as a standalone script and get them all working as expected, and then to cut and paste the scripts into the Windows Script File in the areas identified for each script by the XML tags.

As you can see, this JScript is broken down into the same three sections that I've been using to organize this book's VBScripts (that is, the initialization section, the main processing section, and the procedure section). Comments in JScript are created using the // characters, and I have added a number of them to the script to explain its operation. The script's only function, `DisplayInitialSplashScreen()`, is responsible for displaying the VBScript game console's initial splash screen, which it does using the `WshShell` object's `Popup()` method. JScript does not provide any functions that work similarly to the VBScript `MsgBox()` or `InputBox()` functions. Therefore, to display text in a pop-up dialog box using JScript, you must use either the `WshShell` object's `Popup()` method or the `WScript` object's `Echo()` method.

Developing the VBScript Game Console

The VBScript portion of the VBScript game console contains the bulk of the complexity and programming logic. The first step in developing this VBScript is to insert your VBScript template and fill it in, as follows:

```
*****
'Script Name: N/A
'Author:      Jerry Ford
'Created:     03/01/14
'Description: Display the VBScript game console interface
*****

'Initialization Section

Option Explicit
```

Defining the Elements in the Initialization Section

Next, let's define the variables, objects, and array used by the VBScript. In most of the VBScripts that you've seen in this book, I've included a constant that defines the title bar message to be displayed in the script's pop-up dialog boxes. However, this time I've omitted this constant in the VBScript because I have, instead, defined this value using the <reference> and </reference> tags at the beginning of the Windows Script File. This allows me to retrieve the constant and create a standard title bar message for every script defined in the Windows Script File.

```
Dim objFsoObject, objWshShl, strPlayOrNot, strConsoleStatus
Dim objGameFolder, objGames, strSelection, objWordList
Dim strFileString, intCount, strDisplayString, intNoFilesFound
Dim strTitleBarMsg, intResults, strGamePath

'Specify the location where the game script files are stored
strGamePath = "C:\VBScriptGames"

Dim ConsoleArray()

'Set up an instance of the FileSystemObject object
Set objFsoObject = CreateObject("Scripting.FileSystemObject")

'Set up an instance of the WshShell object
Set objWshShl = WScript.CreateObject("WScript.Shell")

'Retrieve the title bar message to the displayed in pop-up dialog boxes
strTitleBarMsg = getResource("cTitlebarMsg")
```


Building the Main Processing Section

The statements listed in the main processing section are straightforward. I began by first checking the value of `intResults`, which was set by the previous JScript. If `intResults` is equal to 2, then the player told the JScript to shut down the game console. However, after executing the `WScript` object's `Quit()` method, inside the JScript, the WSF script keeps running, executing the VBScript. Therefore, you'll need to include this additional check and execute the `WScript` object's `Quit()` method a second time to prevent the VBScript from displaying the game console.

I then used the `FileSystemObject` object's `GetFolderMethod()` to establish a reference to the location where the VBScript games to be displayed in the game console are stored. A `For Each` loop that spins through the list of files stored in this folder, keeping a record of the number of files counted, is executed.

Trap

As the VBScript is currently written, it expects to find only VBScript files stored in the game folder. Therefore, no steps have been taken to filter out other file types. If you plan to store different files in the game folder, you will need to add logic to the VBScript to prevent it from displaying those files as well.

Next, the VBScript's array is resized according to the number of files found. This array is used to store the names of each VBScript game and to associate each VBScript game with its assigned number as shown in the game console's dialog box. Finally, the `ConsoleLoop()` function is called. This function is responsible for the overall operation of the VBScript game console.

```
'Main Processing Section

If intResults = 2 Then
    WScript.Quit()
End If

'Specify the location of the folder where game script files are stored
Set objGameFolder = objFsoObject.GetFolder("C:\VBScriptGames")

'Get a list of files stored in the folder
Set objGames = objGameFolder.Files

'Look and count the number of game script files
For Each objWordList In objGames
    intNoFilesFound = intNoFilesFound + 1
Next
```

```
'Redefine the script's array based on number of game script files found
ReDim ConsoleArray(intNoFilesFound)

'Call the function that displays the VBScript game console
ConsoleLoop()
```

Creating the ConsoleLoop() Function

The VBScript game console is controlled by the `ConsoleLoop()` function. This function is responsible for assigning a number to each VBScript, for loading the VBScript's array, for interrogating user input, and for performing the appropriate action based on that input.

```
'This functions displays the VBScript game console, accepts user
'input, validates the input, and starts other VBScript games
Function ConsoleLoop()

    'This string contains a list of all the script game files discovered
    'in the target folder
    strSelection = ""

    'This counter will be used to track individual script game files and
    'will be kept in sync with array entries
    intCount = 0

    'Loop through the list of script game files
    For Each objWordList In objGames

        'Build a master string containing a list of all the script game files
        'But exclude the VBScriptGameConsole.wsf file from this list
        If objWordList.Name <> "VBScriptGameConsole.wsf" Then

            'Increment count each time through the loop
            intCount = intCount + 1

            strFileString = strFileString & " " & objWordList.Name

            'Build another list, adding number for later display
            strSelection = strSelection & intCount & ". " & _
                objWordList.Name & vbCrLf

            'Load the name of each script into the array
```

```
ConsoleArray(intCount) = objWordList.Name
```

```
End If
```

```
Next
```

```
'This variable is used to determine when to close the console  
strConsoleStatus = "Active"
```

```
'Create loop & keep it running until the user decides to close it  
Do Until strConsoleStatus = "Terminate"
```

```
  'Interrogate the user's input  
  strPlayOrNot = UCase(PickAGame())
```

```
  'If the user did not type anything or if he clicked on  
  'Cancel then exit the function let things come to an end  
  If strPlayOrNot = "" Then  
    Exit Function  
  End If
```

```
  'Define a Select Case statement and use it to test the various  
  'possible types of user input  
  Select Case UCase(strPlayOrNot)
```

```
    'If the user typed QUIT then exit the function let things  
    'come to an end  
    Case "QUIT"  
      Exit Function
```

```
    'If the user typed ABOUT call the function that displays  
    'additional information about the VBScript game console  
    Case "ABOUT"  
      AboutFunction()
```

```
    'If the user typed HELP call the function that provides  
    'additional help information  
    Case "HELP"  
      HelpFunction()
```

```
    'Otherwise call the function that runs the selected VBScript
```

```
Case Else
    ValidateAndRun()'

End Select
Loop

End Function
```

Creating the ValidateAndRun() Function

When called by the `ConsoleLoop()` function, the `ValidateAndRun()` function, shown here, validates user input by making sure the user has supplied either a valid game number or valid game name. If a valid number or name is not supplied, then the function calls the `InvalidChoice()` function, which displays a generic error message telling the user how to properly operate the VBScript game console. If a valid number or name is supplied, then the function calls the `RunScript()` function, which then executes the specified VBScript game.

```
'This function validates user input and if appropriate calls
'functions that display further instructions or run the selected
'VBScript
Function ValidateAndRun()

    'Check to see if the user provided a valid game number
    If IsNumeric(strPlayOrNot) <> 0 Then
        'Make sure that the user did not type a negative number
        If strPlayOrNot > 0 Then
            'Make sure that the user did not type an invalid number
            If CInt(strPlayOrNot) <= CInt(intCount) Then
                'If the number is valid then find the associated script
                strPlayOrNot = ConsoleArray(strPlayOrNot)
                'Call the procedure that will then run the selected script
                RunScript()
            Else
                'Call this procedure if the user has not typed a valid
                'script number
                InvalidChoice()
            End If
        Else
            InvalidChoice()
        End If
    End If
```

```
'Check to see instead if the user provided a valid game name
Else
  'Proceed only if the input typed by the user is a valid VBScript
  'game (e.g. its name appears in the previously built list of
  'VBScript game names
  If InStr(1, strSelection, strPlayOrNot, 1) > 1 Then
    'If the user didn't type the .vbs file extension, add it
    If InStr(1, strPlayOrNot, ".VBS", 1) = 0 Then
      strPlayOrNot = strPlayOrNot & ".vbs"
      'Recheck to make sure that the script name is still valid
      If InStr(1, strSelection, strPlayOrNot, 1) > 1 Then
        'Call the procedure that runs the selected script
        RunScript()
      Else
        'Call this procedure if the user has not typed a valid
        'script name
        InvalidChoice()
      End If
    Else
      'If the user specified the script's .vbs file extension and
      'it is found in the previously built list of VBScript game
      'names then go ahead and call the procedure that will run
      'the script
      If InStr(1, strSelection, strPlayOrNot, 1) > 1 Then
        RunScript()
      Else
        'Run this procedure if user fails to supply valid input
        InvalidChoice()
      End If
    End If
  Else
    'If user-supplied input is not found in the previously
    'built list of VBScript game names, call this procedure
    InvalidChoice()
  End If
End If

End Function
```

Creating the PickAGame() Function

The `PickAGame()` function, shown next, is charged with displaying the contents of the VBScript game console whenever it is called. It does this by first building a primary display string that consists of a list of all VBScript games that have been found, as well as instructions for getting help, information about the script and its author, and information about closing the game console.

The display string, which is aptly named `DisplayString`, is then plugged into a VBScript `InputBox()` function, thus displaying information about your VBScript games and providing the user with a means of selecting those games.

```
'This function displays the main VBScript game console and collects
'user input
Function PickAGame()
    strDisplayString = strSelection & vbCrLf & _
        "Or Type: [Help] [About] [Quit]" & vbCrLf

    PickAGame = InputBox("W e l c o m e   t o   t h e" & vbCrLf & _
        vbCrLf & "V B S c r i p t   G a m e   C o n s o l e !" & _
        vbCrLf & vbCrLf & "Pick a Game:" & vbCrLf & vbCrLf & _
        strDisplayString, strTitleBarMsg, "", 50, 50)
End Function
```

By default, all WSH and VBScript pop-up dialog boxes are displayed in the middle of the display area. However, in the previous example I specified values of 50 and 50 as the last two attributes of the `InputBox()` function. These two values specify the location where the pop-up dialog box should be displayed on the user's screen. In this case, the pop-up dialog box will be displayed in the upper-left corner of the screen. This keeps it handy without crowding the display area in the middle of the screen, where the VBScript games are displayed.

Creating the RunScript() Function

The `RunScript()` function, shown here, is very straightforward. When called, it uses the `WshShell` object's `Run()` method to execute the VBScript selected by the user, as specified in the variable called `PlayOrNot`.

```
'This function starts the VBScript selected by the user
Function RunScript()
    objWshShl.Run "WScript " & strGamePath & "\" & strPlayOrNot
End Function
```

Creating the InvalidChoice() Function

The `InvalidChoice()` function, shown next, is responsible for displaying a generic error message using the VBScript `MsgBox()` function whenever the user provides the VBScript game console with invalid

input. Examples of invalid input include numbers that have not been assigned to a VBScript listed in the console, such as -4 or 9999, as well as misspelled names of listed VBScript games.

```
'This function is called whenever the user provides invalid input
Function InvalidChoice()
    MsgBox "Sorry. Your selection was not valid. A valid " & _
        "selection consists of one of the following:" & vbCrLf & _
        vbCrLf & "* The number associated with one of the listed " & _
        "VBScript games" & vbCrLf & "* The name of a listed " & _
        "VBScript game" & vbCrLf & "* The name of a listed " & _
        "VBScript game plus its file extension" & vbCrLf & _
        "* Help - To view help information." & vbCrLf & _
        "* About - To view additional information about this game " & _
        "and its Author" & vbCrLf & "* Quit - To close the " & _
        "VBScript Game Console" & vbCrLf & vbCrLf & _
        "Please try again.", , strTitleBarMsg
End Function
```

Creating the HelpFunction() Function

The HelpFunction() function, shown next, uses the VBScript MsgBox() function to display additional help information about the VBScript game console. It is called anytime the user types help and clicks OK.

```
'This function displays help information in a pop-up dialog box
Function HelpFunction()
    MsgBox "Additional help information for the VBScript Game " & _
        "Console can be found at:" & vbCrLf & vbCrLf & _
        "www.xxxxxxxx.com.", , strTitleBarMsg
End Function
```

Creating the AboutFunction() Function

The final function in the VBScript, shown next, is responsible for displaying information about the VBScript game console and its author. It is called whenever the user types about in the VBScript game console and clicks OK. As you can see, the function consists of a single statement that uses the MsgBox() function. The information included here is really just a brief template; I leave it to you to finish adding whatever content you think is appropriate.

```
'Display information about the VBScript game console and its author
Function AboutFunction()
    MsgBox "VBScript Game Console © Jerry Ford 2014" & vbCrLf & _
        vbCrLf & "Email the author at: xxxxx@xxxxxxxx.com.", , strTitleBarMsg
End Function
```

Writing the Second JScript

The final script defined in this Windows Script File, shown next, is JScript. It displays the game's closing splash screen and is designed in the same basic manner as the first JScript, using the `WshShell` object's `Popup()` method to display its graphical pop-up dialog box.

```
//*****  
//Script Name: N/A  
//Author:      Jerry Ford  
//Created:     03/01/14  
//Description: Display the WSF's closing splash screen  
//*****  
  
//Initialization Section  
  
var objWshShl = WScript.CreateObject("WScript.Shell");  
var strMessage;  
var strAuthor;  
var dtmDate;  
var strTitleBarMsg;  
  
strMessage = "Thank you for using the VBScript Game Console © ";  
strAuthor = "Jerry Ford ";  
dtmDate = "2014";  
  
//Main Processing Section  
  
strTitleBarMsg = getResource("cTitlebarMsg");  
  
//Display a pop-up dialog box using the WshShell object's Popup() method  
objWshShl.Popup(strMessage + strAuthor + dtmDate, 0, strTitleBarMsg);
```

The Final Result

That's it. If you have not already done so, go ahead and assemble the entire Windows Script File. Before you run it for the first time, be sure you've first created a folder called `C:\VBScriptGames` and that you've copied both the VBScript game console script as well as a few other of your VBScript games into the folder.

Summary

In this chapter, you learned how to combine two or more different scripts into a single script using Windows Script Files. Windows Script Files are created using XML and a combination of different scripting languages. In addition, you got the opportunity to demonstrate your ability to work with Windows Script Files by developing the VBScript game console.

At this point, you should have a solid understanding of both VBScript and the WSH and should feel confident not just in your game-development capabilities but also your ability to apply the knowledge and skills you've learned here in real-world situations.

13

Working with the Windows Management Instrumentation

This book has demonstrated the power and flexibility of the WSH and VBScript. This included teaching you how to develop scripts that are capable of interacting with both the Windows file system and Registry and of executing system commands to perform all kinds of administrative tasks. All these types of tasks were performed using different WSH and VBScript objects and their associated properties and methods. In addition to administering system resources using WSH and VBScript objects, the WSH also enables you to interact with the Windows Management Instrumentation, or WMI, to query and interact with managed resources available on your computer and network. In this chapter, you will learn the following:

- About the WMI infrastructure
- The basic steps involved in retrieving WMI data and scripting WMI tasks
- How to work with the WMI Query Language
- How to develop VBScripts that execute WMI queries

Introducing the Windows Management Instrumentation

The Windows Management Instrumentation (WMI) is Microsoft's primary systems and applications management support technology for its Windows operating systems. The WMI provides the infrastructure through which Windows resources are accessed and managed. The WMI also provides a framework through which Windows resources are defined and exposed, making them available for monitoring and configuration, thus providing a means for administering system, application, and network resources.

Through interaction with the WMI, you can create and execute VBScripts that execute under the control of the WSH, which are capable of reporting on, interacting with, and administering many different resources, such as available disk space, the motherboard, and virtual memory. As will be demonstrated, the WMI provides the ability to access and manage a host of different resources, including the following:

- The file system
- Event logs
- Printers
- Disk drives
- The Registry
- Shares
- Active processes
- Services
- The scheduler
- Performance data
- Network services (DHCP, DNS, and SNMP)
- System hardware (motherboard, memory, etc.)

In addition to providing access to data for all of these resources, the WMI also supports the ability to set up the real-time monitoring of resources such as the generation of event log records or the modification of the Windows Registry or file system.

Microsoft originally released the WMI as part of Service Pack 4 for Windows NT back in 1998. It has since come pre-installed on all Windows computers starting with Windows 2000. The WMI is actually a Microsoft implementation of a Web-Based Enterprise Management (WBEM) standard. WMI also complies with the Common Information Model (CIM) standard. Both of these standards were developed by the Distributed Management Task Force, also known as the DMTF.

Hint

The DMTF is an industry consortium composed of more than 100 companies, like Microsoft, IBM, Intel, Dell, Hewlett-Packard, and Oracle. It is dedicated to the development of standards for systems manageability. You can learn more about it by visiting its website at www.dmtf.org/home.

Among the benefits of the WMI is the ability to locally and remotely manage computers running Microsoft operating systems. Thus, using a scripting language like VBScript, along with the WSH, you can programmatically interact with and manage just about every major component that operates on a Windows computer, both hardware and software. The WMI also provides access to network resources, including DNS and DHCP, enabling the administration of different network resources. The WMI also supports the remote administration of Windows computers across a network, provided that administrators executing the scripts have the appropriate access permissions to resources on remote systems.

WMI Infrastructure Overview

The WMI framework is composed of an architecture that is made up of three primary parts:

- **Consumers.** A script, application, or management utility that accesses and administers managed resources using the WMI infrastructure.
- **The WMI infrastructure.** Four components (WMI scripting library, CIM object manager, WMI providers, and Common Information Model) that provide the framework through which managed resources are defined and accessed.
- **Managed resources.** Physical or logical components (such as a disk drives and Windows services) that are exposed by the WMI, allowing data to be collected regarding the state of resources and for resources to be monitored and managed.

Figure 13.1 provides a depiction of how these different components interact and work together to support the operation of the WMI.

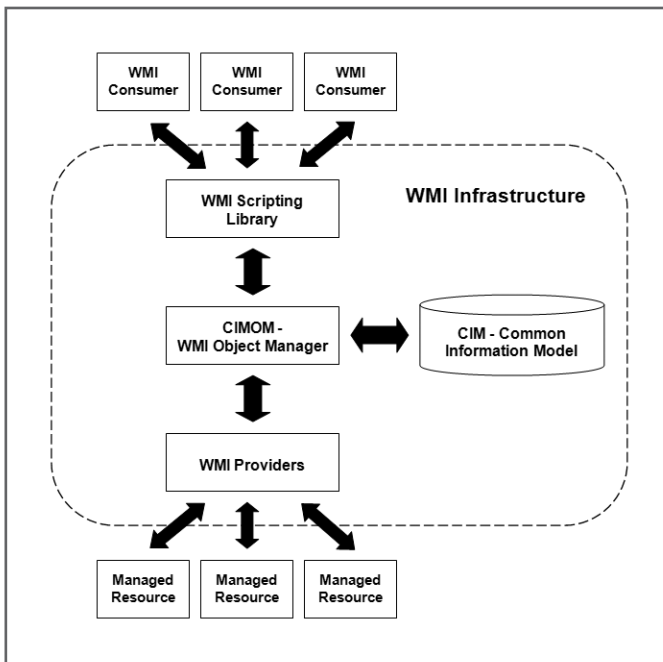


Figure 13.1 A depiction of the components that make up the WMI infrastructure. © 2014 Cengage Learning.

A high-level understanding of what each of the components shown in Figure 13.1 does is essential to understanding how to work with the WMI and is provided in the sections that follow.

Identifying WMI Consumers

The management resources exposed by the WMI are made available to WMI consumers. A *WMI consumer* is any script, program, or utility that accesses and administers managed resources via the WMI infrastructure. WMI consumers occur in many forms, including enterprise management applications like Microsoft Systems Management Server and VBScripts executed by the WSH. This chapter will demonstrate how to develop VBScripts, which as WMI consumers, access and administer Windows resources.

Examining the WMI Infrastructure

The WMI infrastructure consists of four major components. It is through these four components that managed resources are exposed, accessed, and managed. The components are as follows:

- **WMI scripting library.** This is a collection of objects that offer access to the WMI infrastructure, providing you with everything needed to programmatically retrieve WMI data and administer managed resources.
- **CIM object manager (CIMOM).** This is the WMI component responsible for managing the exchange of data between consumer applications and WMI providers.
- **Common Information Model (CIM).** This is a repository used to store and manage all WMI namespaces, which contain class definitions for every managed resource.
- **WMI providers.** These serve as an interface through which the WMI interacts with and manages resources.

The WMI Scripting Library

To develop WMI consumers using VBScript, you need to learn how scripts interact with the WMI scripting library. The WMI scripting library facilitates the scripting of WMI-managed resources by providing access to a set of objects that allow you access to the WMI infrastructure. The objects provide a consistent interface for accessing the WMI infrastructure.

As you will learn later in this chapter, you can access WMI resources from within VBScripts by connecting to the WMI service and then retrieving an instance of a WMI managed resource with which you can access properties and methods belonging to specific types of WMI classes. You can also execute queries that retrieve data based on your specified criteria.

The CIM Object Manager

The CIM object manager, or CIMOM, manages the exchange of data between providers and consumers. In addition to managing the retrieval of data from managed resources, the CIMOM also serves as the interface through which consumers access the WMI and submit requests.

The CIMOM provides a range of services on behalf of the WMI, including the following:

- **Query management.** CIMOM allows consumers to submit queries that collect WMI information using the WMI Query Language (WQL).
- **Routing management.** CIMOM automatically routes information requests to the appropriate provider to retrieve data for a specified managed resource.
- **Remote access management.** CIMOM facilitates remote access to network computers by establishing connections to the CIMOM running on other network computers.
- **Provider registration.** CIMOM allows providers to register with the WMI and to identify the types of data and tasks they can perform.
- **WMI security.** CIMOM restricts access to WMI resources by ensuring that the user has appropriate security permissions.

The CIMOM retrieves information from the Common Information Model (CIM). This enables the CIMOM to determine which provider to hand off requests to in order to fulfill WMI requests.

The Common Information Model

The WMI maintains information about the different types of resources through class definitions stored in the *Common Information Model*, or *CIM*. The CIM serves as a repository for information collected by WMI providers and is responsible for managing the definitions for all repository objects.

The CIM is an object-oriented repository that defines all the different objects that make up a managed system. The CIM organizes this information into classes. A *class* defines all the properties and methods that describe and control the interaction of managed resources. The CIM stores classes using a hierarchy, similar to the way Windows manages the Windows file system. Within this hierarchy, the term *namespace* is roughly equivalent to that of a file system's folder.

The CIM maintains namespaces in a hierarchy, where child classes inherit attributes from their parent classes. An example of a namespace is the `root\cimv2` namespace, within which a series of classes is stored that define managed resources for the computer and its operating system.

Definition

A *class* is a template used to define and create objects, which include properties that describe object attributes and methods for interacting with and controlling objects that are instantiated based on the class.

The CIM contains a class for every managed resource. For example, the `Win32_OperatingSystem` class represents the computer's operating system. This class defines an abundance of properties and methods. One such property is `BootDevice`, which identifies the disk drive used to load the Windows operating system when the computer is started. This class defines numerous methods, including the `Win32Shutdown` method, which can be used to shut down a computer using any of the options supported by Windows (log off, shutdown, reboot, etc.).

Definition

A *namespace* is a hierarchal container used to organize and maintain relationships between classes.

You will see examples of scripts that demonstrate how to work with and extract data using a number of different classes in this chapter. These classes include the following:

- Win32_Service
- Win32_BIOS
- Win32_BaseBoard
- Win32_NTLogEvent
- Win32_OperatingSystem

Hint

Within the CIM, classes are used to outline descriptions of managed resources. Because of the dynamic nature of computer systems, data about managed resources is not stored in the CIM. Instead it is collected when needed, ensuring that it is always up to date and accurate. To interact with specific managed resources, you must instantiate specific instances of objects, using the appropriate classes as the basis for instantiating the objects.

Working with WMI Providers

The WMI provides access to managed resources through providers. Providers serve as intermediaries between the WMI, or more specifically the CIMOM, and the different parts of the operating system, network, and applications. A different WMI provider exists for each type of managed resource.

WMI providers are responsible for collecting data from managed resources and for sending management instructions to them. Examples of providers include the Win32, Event Log, Registry, and Active Directory providers. The Win32 provider gives you access to information about the computer, including the file systems, disk drives, operating system, services, printers, shares, etc. The Event Log provider offers the ability to access, read, delete, copy, and monitor event logs. The Registry provider offers the ability to programmatically access, modify, and delete Registry keys and values. The Active Directory provider allows you to interact with and manage Active Directory objects.

The WMI is an extensible framework, enabling software developers to develop additional providers to expose different parts of their applications, thus allowing their management through WMI. Examples of such applications include the Internet Information Server and Systems Management Server applications.

Hint

To learn more about the different types of WMI providers now available, visit the “WMI Providers” page at [http://msdn.microsoft.com/en-us/library/aa394570\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394570(VS.85).aspx).

Identifying Managed Resources

A *managed resource* is any hardware or software component exposed by the WMI (for which a class has been defined). The WMI uses providers to communicate with managed resources. Managed resources include such things as the Windows file system, disk drives, event logs, printers, the Registry, the motherboard, processes, services, and shares.

Scripting the WMI

Thanks to the WMI scripting library, you can create scripts that interact with the WMI. These scripts can execute locally on your computer or remotely, across a network, on any computer for which you have administrative access. The scripts that you execute do not have to be pre-deployed on remote computers. The WMI will automatically create a temporary copy of any script that it is asked to execute to the remote computer, significantly simplifying the overall effort involved.

Definition

A *managed resource* is a hardware or software component that is exposed and therefore manageable by the WMI.

Developing WMI Scripts

As you are about to learn, most VBScripts that interact with the WMI do so by executing three basic steps:

1. Connecting to the WMI
2. Retrieving access to a WMI-managed resource
3. Retrieving results or executing class methods

To connect to the WMI, either on your own computer or remotely to a networked computer, you call upon the `GetObject` function, passing it the name of the WMI scripting library and of the computer on which the script is to execute, as demonstrated here:

```
set WMIservices = GetObject("winmgmts:\\DELL-PC")
```

The WMI moniker is "winmgmts:". The name of the computer is formed by typing the characters \\ followed by the computer's name. Specifying your computer's name allows you to execute your VBScript locally on your computer. By specifying the name of a remote computer, you can execute your script on a different network computer, provided you have permission to do so. As an alternative to specifying the name of your computer to run a script locally, you may instead execute the script locally by substituting "." in place of the computer's name. As such, you could rewrite the following statement

```
Set WMIservices = GetObject("winmgmts:\\DELL-PC")
```

as shown here:

```
Set WMIservices = GetObject("winmgmts:\\.")
```


Trick

A quick and easy way to determine a computer's name is to log on to it and type `hostname` at the command prompt, as demonstrated here:

```
C:\Users\Jerry>hostname
DELL-PC
```

Here, the computer's name is **DELL-PC**.

Once executed, the `GetObject` function returns a reference to an object named `SWbemServices`, which is one of the objects that the WMI makes available through its scripting library. The `SWbemServices` object has a method named `InstancesOf`, which you can use to retrieve data from a specified WMI class, as demonstrated here:

```
set WMIcollection = WMIServices.InstancesOf("Win32_OperatingSystem")
```

The `InstancesOf` method retrieves a collection named `SWbemObjectSet`, which is automatically populated with a list of all instances of the managed resources identified for the specified class name. Each item in the `SWbemObjectSet` collection represents an instance of the specified managed resource. Once the `SWbemObjectSet` collection has been generated, you can process its contents and use different `SWbemObject` properties and methods.

Retrieving Operating System Data

Okay, that's enough information about the WMI's architecture. It is time to begin putting your newfound understanding of the WMI to work. Let's begin by developing a script that demonstrates how to retrieve information about the operating system on which the script executes. This script, named `GetOSInfo.vbs`, is shown here:

```
'*****
'Script Name: GetOSInfo.vbs
'Author:      Jerry Ford
'Created:     03/06/14
'Description: This script demonstrates how to use WMI to retrieve
'              operating system information
'*****

'Initialization Section
Option Explicit

'Main Processing Section
DisplayData("DELL-PC") 'Set the computer on which the script will execute
WScript.Quit() 'Terminate script execution
```

```
'Procedure Section
```

```
'This function retrieves and displays operating system information
```

```
Function DisplayData(trgtSystem)
```

```
    DIM WMI Services, WMI Collection, WMI Object
```

```
    Set WMI Services = GetObject("winmgmts:\\\" & trgtSystem )
```

```
    Set WMI Collection = WMI Services.InstancesOf("Win32_OperatingSystem")
```

```
    For Each WMI Object In WMI Collection
```

```
        Wscript.Echo "Operating System: " & WMI Object.Caption
```

```
        Wscript.Echo "OS Version Number: " & WMI Object.Version
```

```
        Wscript.Echo "OS Manufacturer: " & WMI Object.Manufacturer
```

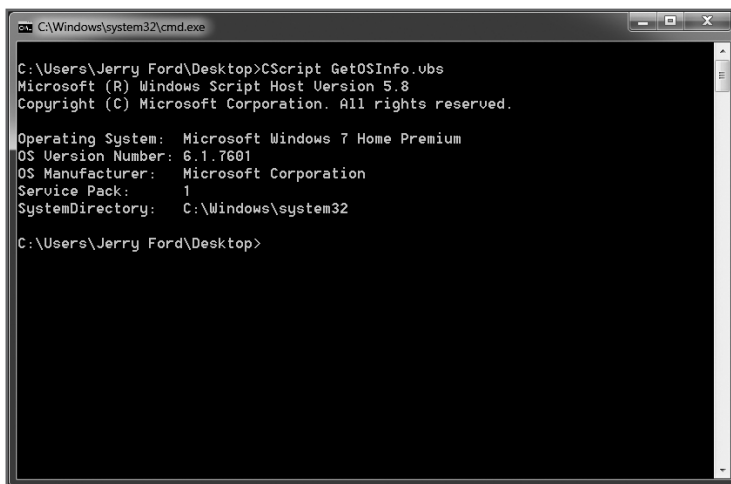
```
        Wscript.Echo "Service Pack: " & WMI Object.ServicePackMajorVersion
```

```
        Wscript.Echo "SystemDirectory: " & WMI Object.SystemDirectory
```

```
    Next
```

```
End Function
```

This script executes a function named `DisplayData()`, passing it the name of the computer upon which the script should execute. Once executed, the `DisplayData()` function uses the `GetObject` method to retrieve a reference to the `SWbemServices` object. Next, using the `InstancesOf` method and the `Win32_OperatingSystem` class, a collection representing the Windows operating system is returned. (Because the computer has a single operating system, the collection consists of a single entry.) This class provides access to dozens of properties that contain information about the operating system, a few of which are displayed when the script's loop executes. Figure 13.2 shows the output generated when I executed this script on a computer running Windows 7.



```
C:\Windows\system32\cmd.exe
C:\Users\Jerry Ford\Desktop>CScript GetOSInfo.vbs
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

Operating System: Microsoft Windows 7 Home Premium
OS Version Number: 6.1.7601
OS Manufacturer: Microsoft Corporation
Service Pack: 1
SystemDirectory: C:\Windows\system32

C:\Users\Jerry Ford\Desktop>
```

Figure 13.2 Using the WMI to retrieve and display information about the computer upon which the VBScript is executed.

Hint

The WMI maintains a WMI class for every managed resource. To learn more about the different types of WMI classes now available, visit the “WMI Classes” page at [http://msdn.microsoft.com/en-us/library/aa394554\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394554(VS.85).aspx).

Retrieving Information About Windows Services

As the following script will demonstrate, you can use this same basic set of steps to retrieve and display information about any managed resource. Consider, for example, the following script, which is almost identical to the script that was provided in the previous section. The only difference here is that instead of referencing the Win32_OperatingSystem class, this script references the Win32_Service class, along with properties provided by that class.

```

'*****
'Script Name: GetServicesInfo.vbs
'Author:      Jerry Ford
'Created:     03/06/14
'Description: This script demonstrates how to use WMI to retrieve
'             information about Windows services
'*****

'Initialization Section
Option Explicit

'Main Processing Section
DisplayData(".") 'Set the local system as the target
WScript.Quit() 'Terminate script execution

'Procedure Section

'This function displays the amount of physical memory on the target system
Function DisplayData(trgtSystem)
    DIM WMIServices, WMICollection, WMIObject

    Set WMIServices = GetObject("winmgmts:\\\" & trgtSystem)
    Set WMICollection = WMIServices.InstancesOf("Win32_Service")

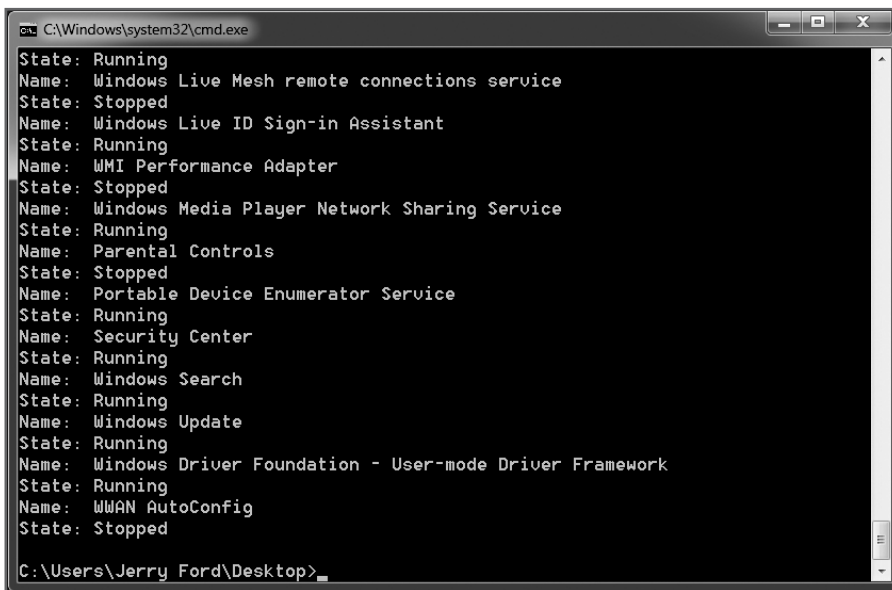
    For Each WMIObject In WMICollection
        WScript.Echo "Name:  " & WMIObject.DisplayName & vbCrLf & _
                    "State: " & WMIObject.State
    Next
End Function

```

Hint

Note the use of "." in place of a hard-coded computer name in this example. This makes it easier to copy the script to another computer and run it locally; you won't have to first remember to modify the script to reflect the computer's name.

The `Win32_Service` class represents the services running on a Windows computer. Its properties contain information about the current status of the service and its methods allow you to manage that service. Because the Windows operating system runs many services, the results that are displayed when the script executes will not all fit on the command console. Figure 13.3 shows the results returned when this script was executed on a computer running Windows 7.



```
C:\Windows\system32\cmd.exe
State: Running
Name: Windows Live Mesh remote connections service
State: Stopped
Name: Windows Live ID Sign-in Assistant
State: Running
Name: WMI Performance Adapter
State: Stopped
Name: Windows Media Player Network Sharing Service
State: Running
Name: Parental Controls
State: Stopped
Name: Portable Device Enumerator Service
State: Running
Name: Security Center
State: Running
Name: Windows Search
State: Running
Name: Windows Update
State: Running
Name: Windows Driver Foundation - User-mode Driver Framework
State: Running
Name: WAN AutoConfig
State: Stopped
C:\Users\Jerry Ford\Desktop>
```

Figure 13.3 Retrieving information about all the services currently running on the computer where the script was executed. © 2014 Microsoft Corporation. Used with permission from Microsoft.

Executing WMI Queries

In addition to collecting data by retrieving instances of specified managed resources, you can also use the WMI Query Language (WQL) to submit queries that, when processed, retrieve information for any matching WMI resources. The WQL is a retrieval-only language, derived from the standard SQL language. It consists of a small collection of statements, as outlined in Table 13.1.

TABLE 13.1 WQL KEYWORDS

Keyword	Description
and	Returns a result of <code>true</code> if the result returned by both of two combined Boolean expressions is equal to <code>true</code>
associators of	Returns a list of every instance of a given resource type
class	Refers to the class associated with a specified query object
from	Specifies the class to be used in the <code>select</code> statement
group clause	Instructs the WMI to create a single notification representing a group of events
having	Refines a <code>select</code> statement query by specifying the type of events to be processed during the group interval by the <code>within</code> keyword
is	Performs a comparison of query data
isa	Performs a comparison against the subclasses belonging to a specified class
keyonly	Reduces query overhead by populating query results with the keys of instances produced by <code>references of</code> and <code>associations of</code> queries
like	Refines query results by determining whether a string appears within the query results
not	Inverts the logic used when processing query results
null	A value that indicates that an object does not have an assigned value
or	Combines two conditions when formulating a query
references of	Returns all instances that refer to a specified source instance
select	Indicates specific properties to be used when formulating a query
true	A Boolean value representing a true result
where	Refines a query's scope by allowing the specification of required criteria
within	Sets the polling or grouping interval to be used by an event query
false	A Boolean value representing a false result

© Jerry Lee Ford, Jr. All Rights Reserved.

In addition to its language keywords, the WQL also supports a standard set of comparison operators that you can use in conjunction with the `where` keyword when developing `select` clauses. These operators are listed in Table 13.2.

The next couple of examples will demonstrate how to integrate WQL queries into your VBScripts.

Hint

The WQL does not permit the creation of cross-namespace queries, limiting you to query a single class at a time.

TABLE 13.2 WMI QUERY LANGUAGE OPERATORS

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

© Jerry Lee Ford, Jr. All Rights Reserved.

Retrieving Event Log Records

As an initial example of how to work with the WQL, let's look at a script that retrieves event log records from the Windows System event log. To write this script, shown here, you will need to use the WMI ExecQuery method in place of the InstancesOf method.

```

*****
'Script Name: GetEventLogInfo.vbs
'Author:      Jerry Ford
'Created:     03/06/14
'Description: This script demonstrates how to use WMI to retrieve
'              records from the Windows event log
*****

'Initialization Section
Option Explicit

'Main Processing Section
DisplayData(".") 'Set the local system as the target
WScript.Quit()  'Terminate script execution

'Procedure Section

'This function retrieves and displays operating system information
Function DisplayData(trgtSystem)
    DIM WMIServices, WMICollection, WMIObject

```

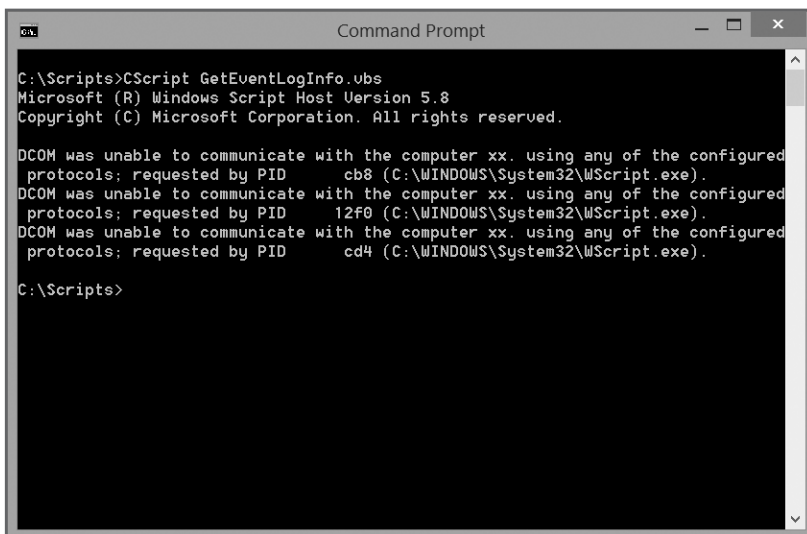
```
Set WMIservices = GetObject("winmgmts:\\\" & trgtSystem)
Set WMIcollection = _
    WMIservices.ExecQuery("Select * from Win32_NTLogEvent where " & _
        "LogFile = 'System' and EventType = 1", , 48)

For Each WMIObject In WMIcollection
    Wscript.Echo "Operating System: " & WMIObject.Message
Next
End Function
```

As this example demonstrates, when working with the `ExecQuery` method, you must pass it a query string that identifies the WMI class that you want to access. Here, the query string begins with the `Select` keyword followed by the `*` character, which instructs the WMI to return all matching instances from the `Win32_NTLogEvent` class. Next, the `where` keyword is used to refine the query. When used, the `where` keyword creates a clause made up of a class property, an operator, and a constant value. In the case of this script, two properties are specified, `LogFile` and `EventType`, and the equals operator is used. Figure 13.4 provides an example of the type of output that you might see when this example is executed on a computer running Windows 8.1. `System` is specified as the value for `LogFile`, and `1` is specified for `EventType`.

Hint

You can modify the script to work with different event logs by specifying a different log name, such as `Application`. By setting `EventType` to `1`, you instructed the query to retrieve error records. You could just as easily assign a value of `2` to retrieve warning records, `3` for information records, `4` for security audit success, or `5` for security audit failure records.



```
Command Prompt
C:\Scripts>CScript GetEventLogInfo.vbs
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

DCOM was unable to communicate with the computer xx. using any of the configured
protocols; requested by PID      cb8 (C:\WINDOWS\System32\WScript.exe).
DCOM was unable to communicate with the computer xx. using any of the configured
protocols; requested by PID      12f0 (C:\WINDOWS\System32\WScript.exe).
DCOM was unable to communicate with the computer xx. using any of the configured
protocols; requested by PID      cd4 (C:\WINDOWS\System32\WScript.exe).

C:\Scripts>
```

Figure 13.4 Using the WMI to retrieve error records from the System event log.

© 2014 Microsoft Corporation.
Used with permission from Microsoft.

Trap

Depending on how large your event log files are, this script may take a considerable amount of time to execute.

Retrieving BIOS Information

This next example demonstrates how to formulate a query that retrieves BIOS information, demonstrating WMI's ability to collect and process data that would otherwise be difficult to obtain.

```

'*****
'Script Name: GetBIOSInfo.vbs
'Author:      Jerry Ford
'Created:     03/06/14
'Description: This script demonstrates how to use WMI to retrieve
'              System BIOS information
'*****

'Initialization Section
Option Explicit

'Main Processing Section
DisplayData(".") 'Set the local system as the target
WScript.Quit()  'Terminate script execution

'Procedure Section

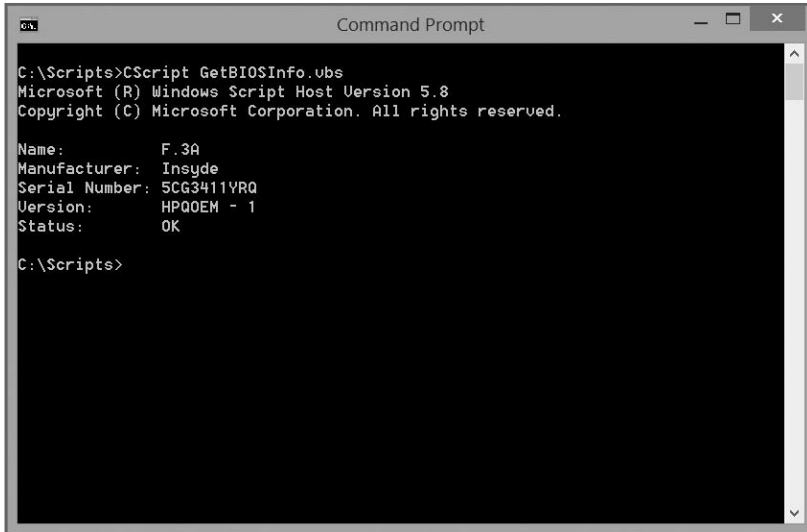
'This function retrieves and displays operating system information
Function DisplayData(trgtSystem)
    DIM WMIservices, WMICollection, WMIObject

    Set WMIservices = GetObject("winmgmts:\\." & trgtSystem)
    Set WMICollection = WMIservices.ExecQuery("Select * from Win32_BIOS", , 48)

    For Each WMIObject In WMICollection
        Wscript.Echo "Name:           " & WMIObject.Name
        Wscript.Echo "Manufacturer:  " & WMIObject.Manufacturer
        Wscript.Echo "Serial Number: " & WMIObject.SerialNumber
        Wscript.Echo "Version:       " & WMIObject.Version
        Wscript.Echo "Status:        " & WMIObject.Status
    Next
End Function

```


As you can see, this query retrieves all available records using the Win32_BIOS class and then displays a subset of the information that is available by referencing Win32_BIOS class properties. Figure 13.5 shows the output produced when this script was executed on a computer running Windows 8.1.



```

Command Prompt
C:\Scripts>CScript GetBIOSInfo.vbs
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

Name:          F.3A
Manufacturer:  Insyde
Serial Number: 5CG341YRQ
Version:       HPQOEM - 1
Status:        OK

C:\Scripts>

```

Figure 13.5 Formulating a WMI query to retrieve information about the computer's BIOS.

© 2014 Microsoft Corporation.
Used with permission from Microsoft.

Retrieving Motherboard Data

In addition to retrieving operating system and BIOS information, the WMI also puts motherboard information at your fingertips, as demonstrated in the following script:

```

'*****
'Script Name: GetMotherBoardInfo.vbs
'Author:      Jerry Ford
'Created:     03/06/14
'Description: This script demonstrates how to use WMI to retrieve
'              motherboard information
'*****

'Initialization Section
Option Explicit

'Main Processing Section
DisplayData(".") 'Set the local system as the target
WScript.Quit()  'Terminate script execution

'Procedure Section

```

```
'This function retrieves and displays operating system information
Function DisplayData(trgtSystem)
    DIM WMI_Services, WMI_Collection, WMI_Object

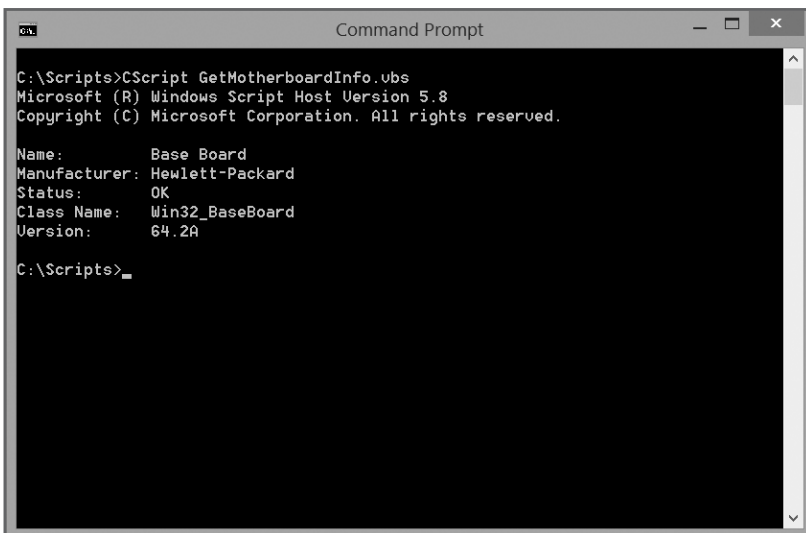
    Set WMI_Services = GetObject("winmgmts:\\\" & trgtSystem)
    Set WMI_Collection = _
        WMI_Services.ExecQuery("Select * from Win32_BaseBoard", , 48)

    For Each WMI_Object In WMI_Collection
        Wscript.Echo "Name:           " & WMI_Object.Name
        Wscript.Echo "Manufacturer: " & WMI_Object.Manufacturer
        Wscript.Echo "Status:       " & WMI_Object.Status
        Wscript.Echo "Class Name:   " & WMI_Object.ClassName
        Wscript.Echo "Version:      " & WMI_Object.Version
    Next
End Function
```

As you can see, this script's WMI query pulls data from the Win32_BaseBoard class. Figure 13.6 shows an example of the output produced when this script was executed on a computer running Windows 8.1.

Hint

Not all motherboards report the same data. Do not be surprised if you run this script on your computer and some of the specified properties are not reported.



```
Command Prompt
C:\Scripts>CScript GetMotherboardInfo.vbs
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

Name:           Base Board
Manufacturer:   Hewlett-Packard
Status:        OK
Class Name:     Win32_BaseBoard
Version:        64.2A

C:\Scripts>_
```

Figure 13.6 Using the WMI to retrieve information about your computer's motherboard.

© 2014 Microsoft Corporation.
Used with permission from Microsoft.

Using the WMI to Manipulate Managed Resources

So far, all of the examples that you have seen have demonstrated how to retrieve and display WMI information. However, as the following example demonstrates, in addition to retrieving WMI data, you can also use the WMI to interact with and administer managed resources, which in the case of the following example happen to be Windows services.

```

*****
'Script Name: WMIServiceCycler.vbs
'Author:      Jerry Ford
'Created:     03/06/14
'Description: This script demonstrates how to use VBScript to stop and
'              start Windows services.
*****

'Initialization Section
Option Explicit
Dim strServiceToManage

'Main Processing Section

'Prompt the user to specify the name of the service to cycle
strServiceToManage = InputBox("What service would you like to cycle?")

'Call the procedure that stops a service
StopService(strServiceToManage)

'Pause for 5 seconds
WScript.Sleep(5000)

'Call the procedure that starts a service
StartService(strServiceToManage)

'Terminate script execution
WScript.Quit()

'Procedure Section

'This subroutine stops a specified service
Function StopService(ServiceName)
    Dim wbemService, listOfServices, windowsService

```

```
Set wbemService = GetObject("winmgmts:\\.")
Set listOfServices = wbemService.InstancesOf("Win32_Service")

For Each windowsService In listOfServices
    If windowsService.Name = ServiceName Then
        MsgBox("Stopping " + windowsService.Name)
        windowsService.StopService
    End If
Next
End Function
```

```
'This subroutine starts a specified service
Function StartService(ServiceName)
    Dim wbemService, listOfServices, windowsService

    Set wbemService = GetObject("winmgmts:\\.")
    Set listOfServices = wbemService.InstancesOf("Win32_Service")

    For Each windowsService In listOfServices
        If windowsService.Name = ServiceName Then
            MsgBox("Starting " + windowsService.Name)
            windowsService.StartService
        End If
    Next
End Function
```

When executed, this script begins by displaying a pop-up dialog box that requests the name of the Windows service to be stopped and then restarted. Figure 13.7 shows the pop-up dialog box that is displayed when the script is first started on a computer running Windows 8.1.

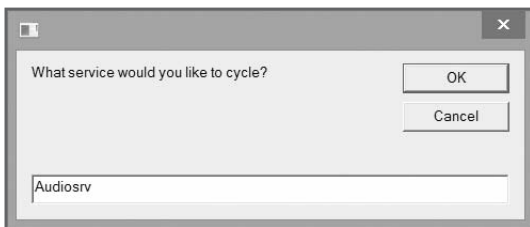


Figure 13.7 The script needs to know the name of the service that you want to cycle.

Next, a function named `StopService()` is called and passed the name of the service as an argument. The script then pauses its execution for five seconds before calling on a function named `StartService()`, passing it the name of the service so that it can be restarted. Both the `StopService()` and the `StartService()` functions start by connecting to the WMI and then use the `Win32_Service` class to generate a list of Windows services. When executed, the `StopService()` function stops the specified service, passed to it as an argument using the `StopService()` method. Similarly, the `StartService()` function uses the `StartService()` method to start the service back up again. Figure 13.8 shows the two pop-up dialog boxes that are displayed by these two functions when the script executes on a computer running Windows 8.1.

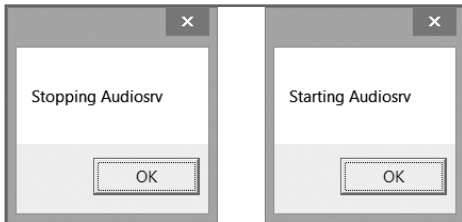


Figure 13.8 The script ensures that the administrator is kept aware of its actions.

© 2014 Cengage Learning.

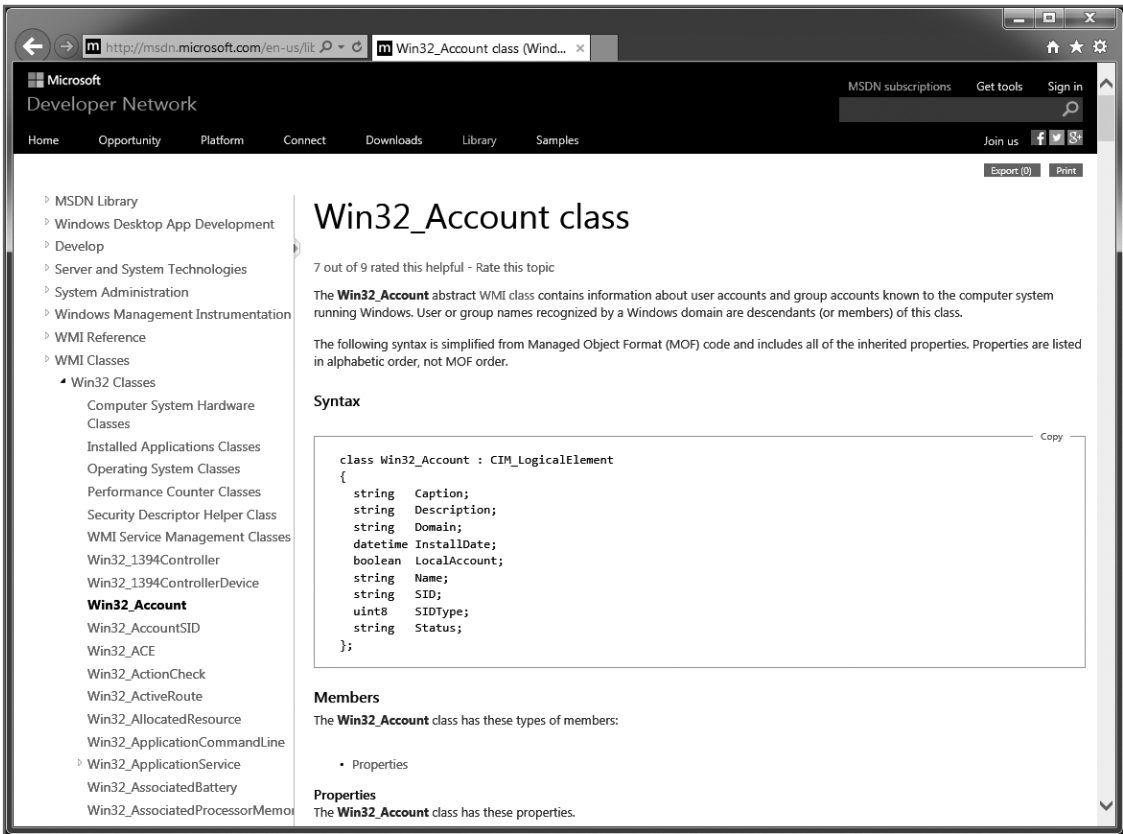
Locating CIM Information

The CIM is the repository used to store and organize all the classes managed by the WMI. As mentioned, the WMI organizes classes into a hierarchy of namespaces. The hardest thing about working with the WMI is arguably finding the class that you need to work with when developing an administrative script. One excellent source of documentation regarding WMI classes is maintained at the Microsoft TechNet website at [http://msdn.microsoft.com/en-us/library/aa394554\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394554(v=vs.85).aspx) (see Figure 13.9).

Here you will not only find a listing of class names but, by drilling down into specific classes, you'll get lists of all the properties and methods defined in specific classes. Not all classes are available on all Windows operating systems; by referring to this WMI documentation, you can save yourself a great deal of frustration and confusion when you write script that works correctly on one type of Windows operating system but not another.

Summary

In this chapter, you learned how the WMI is architected and how its different components work together to facilitate the administration of managed resources. This chapter demonstrated how to retrieve and process WMI data using different classes. You learned how to use the WMI Query Language to formulate queries and retrieve WMI data and then learned how to incorporate the WMI into your VBScripts. In addition to using the WMI to retrieve data provided by different managed resources, you also learned how to use the WMI to interact with and administer resources. Lastly, you learned where to go online to learn more about the different providers, classes, properties, and methods supported by the WMI.



The screenshot shows a web browser window displaying the Microsoft Developer Network (MSDN) page for the `Win32_Account` class. The browser's address bar shows the URL `http://msdn.microsoft.com/en-us/it...`. The page header includes the Microsoft logo, "Developer Network", and navigation links like "Home", "Opportunity", "Platform", "Connect", "Downloads", "Library", and "Samples". A search bar and social media links are also visible.

The main content area is titled "Win32_Account class". Below the title, there is a rating of "7 out of 9 rated this helpful - Rate this topic". The text explains that the `Win32_Account` abstract WMI class contains information about user accounts and group accounts known to the computer system running Windows. It notes that user or group names recognized by a Windows domain are descendants (or members) of this class.

The following syntax is simplified from Managed Object Format (MOF) code and includes all of the inherited properties. Properties are listed in alphabetic order, not MOF order.

Syntax

```
class Win32_Account : CIM_LogicalElement
{
    string Caption;
    string Description;
    string Domain;
    datetime InstallDate;
    boolean LocalAccount;
    string Name;
    string SID;
    uint8 SIDType;
    string Status;
};
```

Members

The `Win32_Account` class has these types of members:

- Properties

Properties

The `Win32_Account` class has these properties.

The left sidebar contains a navigation menu with categories like "MSDN Library", "Windows Desktop App Development", "Develop", "Server and System Technologies", "System Administration", "Windows Management Instrumentation", "WMI Reference", and "WMI Classes". Under "WMI Classes", there is a sub-section for "Win32 Classes" which lists various classes including `Win32_Account`.

Figure 13.9 Visit the Microsoft Developer Network for more information on WMI classes.

© 2014 Microsoft Corporation. Used with permission from Microsoft.

This page intentionally left blank

14

Adding a GUI to Your Scripts

As you have no doubt come to understand, VBScripts executed by the WSH run within the constraints of either the WScript.exe or CScript.exe execution host. As such, other than the ability to display text in a couple of pop-up dialog boxes, your scripts are invariably tied to the Windows command prompt. However, there is another Windows scripting technology capable of leveraging the WSH while at the same time enabling you to wrap your scripts up inside a graphical user interface (GUI). This technology is known as HTML Applications (HTAs). This chapter will provide you with an overview of HTAs and examples of their usage.

Specifically, you will learn the following:

- About the components that make up HTAs
- How to wrap your scripts up in GUIs using HTML
- How to create event driven scripts
- How to configure the appearance of your GUI applications using Cascading Style Sheets (CSS)

Project Preview: The HTA Rock, Paper, Scissors Game

As this book's final project, you will learn how to update the Rock, Paper, Scissors game that you worked on in Chapter 5, "Conditional Logic". You will accomplish this by converting the Rock, Paper, Scissors game into an HTA, adding the HTML tags and content needed to create the game's interface as well as a little CSS to improve the application's presentation. Figures 14.1 and 14.2 demonstrate the operation of both the WSH and the new HTA version of the game as seen on a computer running Windows 7.

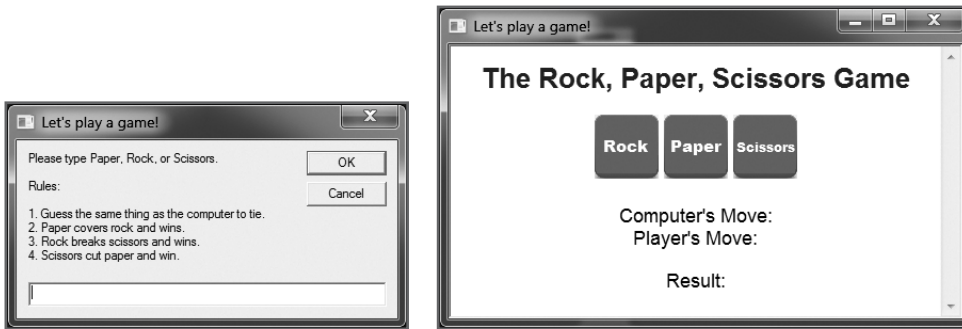


Figure 14.1 It is obvious from the start that the HTA version of the Rock, Paper, Scissors game looks and acts a lot more like a Windows application. © 2014 Cengage Learning.



Figure 14.2 The HTA version of the game is played by clicking on graphical button controls, speeding up game play and eliminating typos. © 2014 Cengage Learning.

Introducing HTML Applications (HTAs)

An *HTML Application (HTA)* is a Windows program made up of HTML and a scripting language. The scripting language, which in the case of this book is VBScript, must be an Internet Explorer–supported scripting language.

HTAs are full-featured Windows applications. HTAs look and execute like other Windows applications, including those developed using programming languages like Visual Basic, Java, and C++. An HTA executes as a trusted Windows application. As such, it executes outside of the constraints of Internet Explorer's security restrictions.

Definition

An *HTML Application (HTA)* is computer program that consists of HTML, Dynamic HTML, and Internet Explorer–supported scripting languages like VBScript, which has a GUI and executes without the limitations that Internet Explorer normally imposes on applications.

This allows it to do things that browser-based scripting otherwise prevents, like accessing the computer's file system and Registry. HTAs can also interact with WMI. HTAs enable administrators to wrap their scripts up inside a graphical user interface (GUI). HTAs are also used to prototype Windows applications, create desktop applications, develop wizards, and so on.

Even though they include HTML, HTAs execute in a different process than Internet Explorer. HTAs are executed by the `mshta.exe` execution host, which is analogous to the WSH `WScript.exe` and `CScript.exe` execution hosts. When you execute an HTA script, the `mshta.exe` execution host first instantiates Internet Explorer's rendering engine and the appropriate scripting engine (for example, `vbscript.dll`). Execution by the `mshta.exe` execution host allows scripts to access the WSH core object model. There are, however, some limitations. For example, an HTA cannot access the `WScript` object's methods, most notably `Quit`, `Echo`, and `Sleep`. These methods are available only when your scripts are executed by the `WScript.exe` or `CScript.exe` execution hosts. The `mshta.exe` execution host has no equivalents.

HTAs have access to the Internet Explorer DOM, which provides them with access to browser controls (for example, text boxes, option buttons, checkbox controls, etc.). Using these elements, you can construct the GUIs for your scripts. Another benefit of HTAs is that in addition to the Internet Explorer DOM, they have access to HTML, XHTML, and even CSS. You can therefore leverage these optional technologies in your HTAs.

Trap

To execute an HTA, Internet Explorer must be enabled on the computer. Other Web browsers are not supported. Therefore, any computer on which you wish to execute an HTA must have Internet Explorer enabled. Starting with Microsoft Vista, Internet Explorer is an optional application, so it may not always be available.

How Do HTAs Compare to HTML Pages?

Like WSH scripts, HTAs are text files. HTA files are organized like HTML files, however. If you are familiar with Web page development, then learning how to develop HTA files should come easily to you. Even if you have not done prior Web development, don't worry. This chapter will teach the basics and get you up and running quickly. The following example shows the construction of a basic HTML page:

```
<html>
  <head>
    <title>A Simple HTML Page</title>
  </head>
  <body>
    <h2>Hello World!</h2>
  </body>
</html>
```

If you save this HTML page and then load it using Internet Explorer, you'll see the output shown in Figure 14.3.

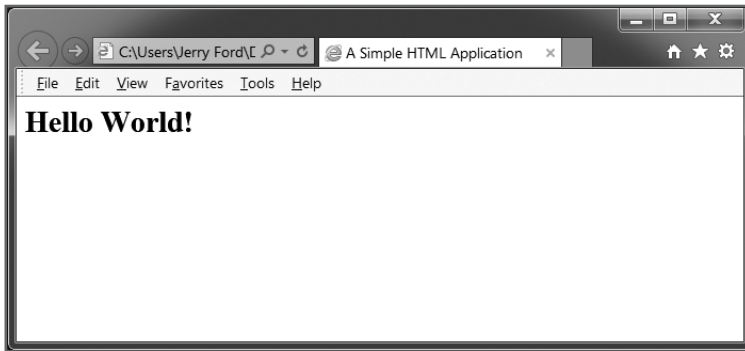


Figure 14.3 A simple HTML page that displays a text string, as seen on a computer running Windows 7. © 2014 Cengage Learning.

Like HTML pages, content in an HTA is marked up using HTML tags. Every HTML page begins and ends with the `<html>` `</html>` tags. Inside these tags are two more sets of high level tags: the `<head>` `</head>` tags and the `<body>` `</body>` tags. Other tags, text, graphics, and media are embedded within the `<head>` `</head>` and `<body>` `</body>` sections, providing the HTML page's content. For example, the preceding HTML page uses the `<title>` `</title>` tags to display a text string in the browser window's title bar. It also displays a text message of "Hello World!" on the window itself using the `<h2>` `</h2>` tags. The `<h2>` `</h2>` tags are level-two heading tags that display enclosed text in an enlarged font size.

Creating and Executing an HTA

HTAs are developed in much the same way as HTML pages, the major difference being that they are saved with an `.hta` file extension. In fact, you can create an HTA by simply creating and saving a text file with an `.hta` file extension. To test this out, use Notepad to create a new text file. Then type the following text in it and save the file with name of `HelloWorld.hta`:

```
Hello World!
```

The `.hta` file extension identifies the file as an HTA. As a result, Windows will automatically run the HTA file using the `mshta.exe` execution host. To execute this HTA, double-click on it. A window similar to the one shown in Figure 14.4 will be displayed.

A simple HTA is displayed using various defaults that specify how its GUI window looks and operates. These defaults are called out in Figure 14.4. However, you can configure application window features by adding the `<HTA:APPLICATION>` tag to the script file. This optional tag exposes a collection of window attributes that you can specify and configure, determining which window features are added to the application window. These attributes also control certain behaviors, such as whether more than one instance of the HTA can be run at a time.

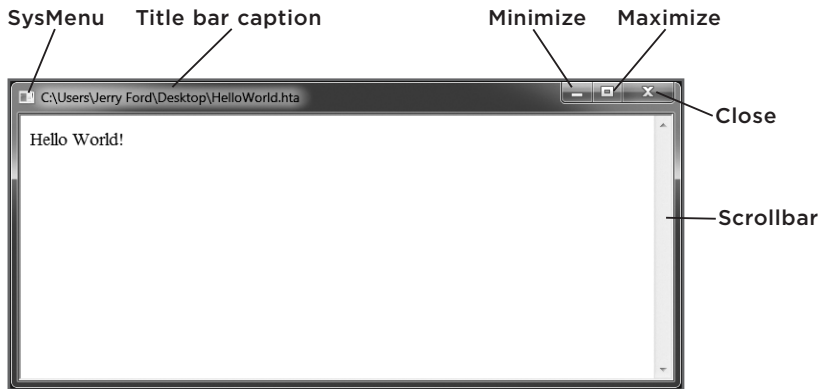


Figure 14.4 A simple HTA made up of a text string, as seen on a computer running Windows 7.

© 2014 Cengage Learning.

Constructing an HTA

To create an HTA that does something other than display plain text in a window, you must add some more HTML to the HTA file. Before you can do this, you must understand how HTAs are organized. While structured as HTML pages, there are really three main parts and an optional fourth part to an HTA that you must focus on. These parts include the following:

- **<HTA:APPLICATION> tag.** This provides control over the appearance of an HTA's application window.
- **<script> </script> tags.** These contain the VBScript statements, functions, and subroutines, and the WSH objects that make your HTAs work.
- **<body> </body> tags.** These contain the HTML tags (elements) that make up an HTA's interface.
- **<style> </style> tags (optional).** These contain Cascading Style Sheets (CSS), which are used to describe the way HTML content is presented.

Introducing the <HTA:APPLICATION> Tag

HTA files are organized like HTML files. The <HTA:APPLICATION> tag is placed in an HTA file's <head> </head> section. This tag is optional. If not included, a default HTA window is displayed without any customization. This tag exposes the HTA DOM, providing access to a collection of attributes that allow HTA to customize the application window within which it executes, specifying features like text box controls, buttons, radio buttons, checkbox controls, etc. The following example converts the preceding HTML page to an HTA (after adding the <HTA:APPLICATION> tag and saving it with an .hta file extension).

```
<html>
  <head>
    <title>My First HTA</title>
    <HTA:APPLICATION
```

```
ID="htaHelloWorldApp"
APPLICATIONNAME="Hello World"
SCROLL="yes"
SINGLEINSTANCE="yes">
>
</head>

<script language="VBScript">
</script>

<body>
  <h2>Hello World!</h2>
</body>
</html>
```

HTAs look and execute just like a Windows application. To run the previous example, all you have to do is double-click on it. Figure 14.5 shows how the HTA looks on a computer running Windows 7.



Figure 14.5 A simple HTA made up of a text string. © 2014 Cengage Learning.

Note that unlike the simple line HTA that was previously presented, the presentation of the text string “Hello World” is governed by its enclosure within the `<h2>` and `</h2>` tags.

An Overview of `<HTA:APPLICATION>` Tag Attributes and Properties

The inclusion of the `<HTA:APPLICATION>` tag in the `<head>` `</head>` section of an HTA provides you with the ability to customize the look and operation of the window within which the HTA executes. Table 14.1 provides a list of all the attributes and properties that are exposed by the `HTA:APPLICATION` object. By specifying and configuring these attributes in the `<HTA:APPLICATION>` tag, you can exercise detailed control over the look and operation of the application window within which an HTA executes.

**TABLE 14.1 HTML APPLICATION ATTRIBUTES
AND PROPERTIES REFERENCE**

Attribute	Property	Description
APPLICATIONNAME	applicationName	Assigns or retrieves the name of an HTA
BORDER	border	Assigns or retrieves the type of window border for an HTA
BORDERSTYLE	borderStyle	Assigns or retrieves the border style for an HTA
CAPTION	caption	Assigns or retrieves a Boolean value indicating if the HTA window should display a caption in its title bar
	commandLine	Retrieves arguments passed to the HTML Application when it was started
CONTEXTMENU	contextMenu	Assigns or retrieves a text string that is used to display a context menu when the user right-clicks
ICON	icon	Assigns or retrieves the path and name of an HTA's icon
INNERBORDER	innerBorder	Assigns or retrieves a text string specifying whether a 3D border is displayed
MAXIMIZEBUTTON	maximizeButton	Assigns or retrieves a text string indicating whether a Maximize button is displayed on the HTA's title bar
MINIMIZEBUTTON	minimizeButton	Assigns or retrieves a text string indicating whether a Minimize button is displayed on the HTA's title bar
NAVIGABLE	navigable	Assigns or retrieves a text string indicating if linked documents should be loaded into the HTA window or into a new browser window
SCROLL	scroll	Assigns or retrieves a text string indicating whether a scrollbar should be displayed
SCROLLFLAT	scrollFlat	Assigns or retrieves a text string indicating the type of scrollbar to display: flat or 3D
SELECTION	selection	Assigns or retrieves a text string indicating whether data content can be selected using the keyboard or mouse
SHOWINTASKBAR	showInTaskBar	Assigns or retrieves a text string indicating whether the Windows taskbar should display the HTA
SINGLEINSTANCE	singleInstance	Assigns or retrieves a text string specifying that only a single instance of the HTA can run at a time
SYSTEMENU	sysMenu	Assigns or retrieves a Boolean value that specifies whether a system menu is displayed in the HTA
VERSION	version	Assigns or retrieves the HTA version number
WINDOWSTATE	windowState	Assigns or retrieves an HTA window's initial size

Table 14.1 provides a high-level overview of all of the attributes and properties belonging to the <HTA:APPLICATION> tag. However, there is a lot more to these attributes and properties than there is room to cover in this chapter. For example, consider the BORDER attribute/border property, shown here:

```
<HTA:APPLICATION BORDER = "sType" ... >
```

sType is a text string specifying one of the following values:

- **thick**. This results in a thick window border that includes a size grip and sizing border, enabling the user to manually resize the window.
- **Dialog**. This results in a dialog box window border.
- **none**. This results in the display of a window without a border.
- **thin**. This results in a thin window border with no title bar.

To effectively work with the attributes and properties supported by the <HTA:APPLICATION> tag, you need to know their syntax and the range of values they support. Microsoft provides this information to you via the HTML Applications Reference, located at <http://msdn.microsoft.com/en-us/library/ms536473%28VS.85%29.aspx>.

Customizing the HTA Attributes and Properties

By specifying and configuring different HTA attributes and properties in the <HTA:APPLICATION> tag, you can exercise detailed control over the appearance and operation of the window in which an HTA executes. As an example of how to do this, consider the following:

```
<html>
  <head>
    <title>Window Without a Title Bar</title>
    <HTA:APPLICATION
      ID="htaNoTitleBar"
      APPLICATIONNAME="Window Without a Titlebar"
      SCROLL="auto"
      SINGLEINSTANCE="yes"
      CAPTION="no"
    >
  </head>

  <script language = "VBScript">
    Sub CloseWindow
      self.close
    End Sub
  </script>
```

```
<body onkeypress="CloseWindow">
  Press any key to close this window.
</body>
</html>
```

Here, the `<HTA:APPLICATION>` tag has been assigned an ID of `htaNoTitlebar` and an application name of `Window Without a Titlebar`. In addition, the window has been configured to display a scrollbar, and only one instance of the application can be executed at a time. Lastly, the value of `CAPTION` has been set to `no`, hiding the display of the window's title bar. Figure 14.6 shows how this example looks when executed on a computer running Windows 7.

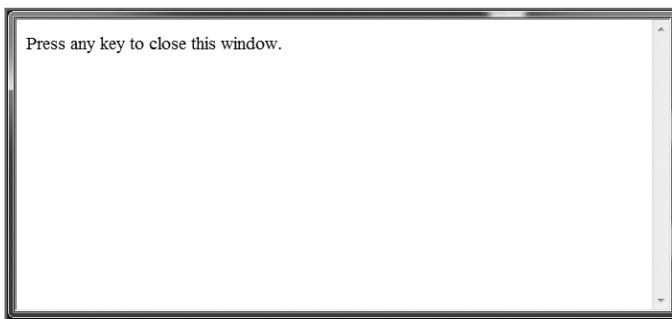


Figure 14.6 An HTA without a title bar cannot be moved and does not feature SysMenu, Minimize, Maximize, or Close buttons. © 2014 Cengage Learning.

An application window like the one shown in Figure 14.6 can be used to display a splash screen or other type of higher customized window. Because the HTA's title bar has been removed, the HTA must provide another means of terminating the application's execution. This will be achieved by assigning the name of the `CloseWindow` subroutine to the `onkeypress` event. This event occurs whenever the user presses any keyboard key. When this happens, the `CloseWindow` subroutine executes. It contains a single statement. `self` is a property of the window object that provides a convenient reference back to the window and `close()` is a window object method that closes the application window.

Trick

You can also terminate an HTA's execution using the Task Manager to locate and terminate the `mshta.exe` execution host process.

The `<script>` `</script>` Tags

Most HTA pages include one or more script files, which are embedded within `<script>` `</script>` tags. All these scripts are organized as named subroutines or functions. HTAs are event-driven applications, so these scripts are configured to execute when particular events occur. HTAs react to events that occur when, for example, an application initially starts or when the administrator clicks an interface button or other control. In short, HTAs start and wait for instructions on what to do based on user input and interaction.

Unlike WSH administrative scripts, which usually start and run to completion immediately upon execution, HTAs start by displaying a GUI and then pause to wait and respond to events that occur. Events occur for all sorts of reasons. When an HTA first starts, the application window's `onLoad` event occurs. In addition, interface controls generate an `onClick` event when selected or clicked. You can use these events to trigger the execution of VBScripts (functions and subroutines) by specifying the name of a script to be executed when an event occurs.

An example of an event that occurs with every HTA is the `onLoad` event, which occurs when an HTA's window is started. You can automatically execute program statements when an application starts by adding a `Window_onLoad` subroutine inside the `<script>` `</script>` tags with any code statements you wish to execute. The following example demonstrates how to work with this event:

```
<script language="VBScript">
  Sub Window_OnLoad
    self.resizeTo 600, 300
  End Sub

  Sub ShowResult
    MsgBox "Hello " & textBox.value
    textBox.value = ""
  End Sub
</script>
```

Here, two scripts have been embedded within the `<script>` `</script>` tags of an HTA. Each script has been saved as a subroutine. The first subroutine is named `Window_OnLoad` and the second subroutine is named `ShowResult`. As stated, the first script automatically executes when the HTA's window is loaded. Using this script, you can execute any programming logic that needs to occur before the user has the opportunity to interact with the HTA. This script consists of a single statement, which executes the window object's `resizeTo()` method. This resizes the width and length of the window, which in the example is 600 pixels wide by 300 pixels high.

The `ShowResult` subroutine can be called by another script or triggered by a control. In the example, `ShowResult` uses the VBScript `MsgBox()` function to display a text message.

The `<body>` `</body>` Tags

An HTA's GUI is laid out using HTML tags defined in the application's `<body>` `</body>` section. The following HTML tags build out a simple GUI made up of a label, a text control, and a button control. All these tags are embedded within the `<body>` `</body>` tags.

Hint

A complete review of HTML and all the tags that it supports is beyond the scope of this book. Instead, you will be introduced to commonly used tags that are required to build effective GUIs. To take a deeper dive into HTML, you may wish to read *HTML, XHTML, and CSS for the Absolute Beginner* (ISBN: 978-1-4354-5423-1).

```
<body>
  <p>
    <label for = "userName">Enter your name:</label>
    <input type="text" name="textBox" id = "userName" size="50"
      maxLength = "50">
  </p>
  <input type="button" value="Click me!" onClick="ShowResult">
</body>
```

In this example, the `<label>` `</label>` tags are used to display a text string at the top of the window. Immediately following the label is a text control. The text control is defined using an `<input>` tag with a type assignment of `text`. It is assigned a name of `textBox`, allowing it to be programmatically referenced. The text control is assigned a size that specifies its length (that is, how many characters it can display) and is assigned a maximum length (that is, the total number of characters it can contain). All of these tags are themselves enclosed within a pair of `<p>` and `</p>` tags, which are paragraph tags that add a bit of space in between interface controls. The last HTML tag in the `<body>` `</body>` section is a second `<input>` tag, assigned a type of `button`, which is used to add a button control to the window. Its `value` property is assigned a value of `Click me!`. This string is displayed on top of the button. Lastly, `ShowResult` is assigned to the button's `onClick` event. As a result, the subroutine is executed whenever the button is clicked.

The `<style>` `</style>` Tags

An HTA's `<style>` `</style>` tags are used to specify Cascading Style Sheet (CSS) rules governing the presentation of the application. CSS is a style sheet programming language that influences the presentation of HTML content. CSS provides numerous advantages, including separation of content and presentation.

CSS has a prioritization scheme that governs how style rules are applied when one or more rules match the same element. As a result, CSS rules cascade downward to document elements (HTML tags) in a predictable manner. CSS rules are used to control presentation aspects such as font type, size, and color, as well as background styles, borders, and the alignment of content.

Hint

A complete review of CSS is beyond the scope of this book. Instead, you will be introduced to a small number of commonly used CSS rules needed to complete the HTA presented in this chapter. To learn more about CSS, you may wish to read *HTML, XHTML, and CSS for the Absolute Beginner* (ISBN: 978-1-4354-5423-1).

CSS has a simple syntax based on English keywords that specify styles and their values. CSS style sheets are made up of lists of rules. As outlined in Figure 14.7, CSS rules are made up of selectors and declaration blocks.

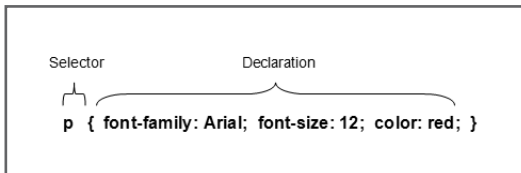


Figure 14.7 A CSS rule is made up of one or more selectors and a declaration block. © 2014 Cengage Learning.

CSS rules identify the element(s) they affect through the selector. *Selectors* specify the HTML elements to which rules apply. You can specify multiple selectors in a rule, provided you separate them by commas. A *declaration block* consists of one or more declarations that allow you to modify one or more aspects of an element's presentation. *Declarations* are placed inside an opening { character and a closing } character. They consist of one or more property/value pairs, as shown in Figure 14.8.

A *property* identifies a presentation aspect that the rule will modify. Property names are separated from their corresponding values using a colon. A *value* is a setting applied to a specified property for the selected elements. Property/value pairs are separated from other property/value pairs using a semi-colon.

Definition

A *declaration* is a statement that modifies one or more aspects of an element's presentation.

Definition

A *selector* identifies an HTML element to which a CSS rule is applied.

Definition

A *declaration block* is a CSS organization construct made up of one or more declarations.

Definition

A *property* specifies the presentation aspect that a CSS rule modifies.

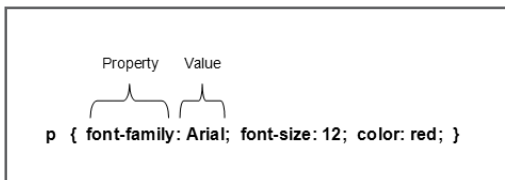


Figure 14.8 Declarations consist of property/value pairs.

Definition

A *value* is a setting assigned to a property for a selected element.

CSS Selectors

Selectors identify the effect of a CSS rule. In other words, it determines which elements are affected by the rule. Table 14.2 identifies a number of the most commonly used CSS selectors and explains their precedence.

TABLE 14.2 COMMONLY USED CSS SELECTORS

Name	Example	Description
Universal	<code>* {color: red;}</code>	A universal selector matches every element found in an HTA file. It is defined using the <code>*</code> character. When executed, the rule in the “Example” column displays all text in the HTA window in red.
Element	<code>p { color: green;}</code>	An element selector matches all instances of a specified element within an HTA window. In the example provided, any text enclosed by <code><p></code> <code></p></code> tags is colored green.
ID	<code>#score {color: blue;}</code>	An ID selector matches a single unique element as specified by its ID attribute. In the example provided, the element who’s ID is <code>score</code> is displayed in blue.

© Jerry Lee Ford, Jr. All Rights Reserved.

If multiple rules select the same element(s), they will attempt to modify the same property. If this occurs, the more exclusive rule’s declaration of that property will be applied. Because an ID selector is more exclusive than the universal selector, a rule matched by an element’s ID can override property values declared in a rule that matches all elements in the document.

Adding Style Rules to Your HTA

To integrate CSS rules into your HTAs, you embed them within the HTA’s `<style>` `</style>` tags. The CSS rules are then applied to matching elements found within the HTA. The following example demonstrates the construction of a small set of CSS rules. Note that in CSS, comments begin with the `/*` characters and end with the `*/` characters.

```
<style>
/*This rule formats all level 1 headings*/
h1 {
    color: purple;
    text-decoration: underline;
    text-align: center;
}
/*This rule formats all paragraphs*/
p {
    font-weight: bold;
```

```
    font-style: italic;
    color: blue;
}
/*This rule formats an element whose id = p1*/
#p1 {
    color: red;
}
</style>
```

The first CSS rule modifies the appearance of text embedded within any `<h1>` `</h1>` tags in the HTA, changing their text color to purple and making the text underlined and centered. The second CSS rule affects paragraph presentation, displaying their text as bold, italic, and blue. The third rule applies to a specific element whose ID is `p1`. CSS denotes IDs by pre-appending a `#` character to them.

The third rule demonstrates CSS specificity by modifying the presentation of one specific paragraph tag. The color property assignment in the third rule conflicts with that of the second rule. CSS resolves this by using the color property from the more specifically selected rule to override the color property from the less specifically selected rule.

Adding Interface Elements

You have already seen an example of how to add a text control and a button control to an HTA. Now it is time to look deeper into how these two controls are constructed and to look at other types of controls that you can add to your HTAs. These include controls like radio buttons and checkboxes as well as various types of list box and text controls. Each of these controls is examined in the sections that follow.

Hint

If you want some help developing your HTAs, consider downloading HTA Helpomatic. HTA Helpomatic is a free tool provided by Microsoft that assists you in developing script code. HTA Helpomatic presents you with a list of HTA elements, such as buttons, checkboxes, and text area controls. It then displays the HTML code required to generate these controls. It also displays program code that can be used to generate a subroutine for each control, upon which you can then expand. To get HTA Helpomatic, go to www.microsoft.com/download/en/default.aspx and search for its name.

Creating Interface Controls Using the `<input>` Tag

As you have already seen, the `<input>` tag is one of the most versatile HTML tags. It is used to create a number of different types of interface controls. This range of controls includes the following:

- Text field controls
- Password field controls
- Checkbox controls
- Radio button controls
- Button controls

To specify the type of control you want using the `<input>` tag, you must specify the control's type using the `<input>` tag's `type` attribute as well as a number of other control attributes using the syntax outlined here:

```
<input type="ControlType" name="ControlName" value="ControlValue"
      onClick="FunctionName">
```

Table 14.3 outlines the function of each of the `<input>` tag attributes outlined in the previous example.

TABLE 14.3 `<INPUT>` TAG ATTRIBUTES

Attribute	Description
<code>type</code>	Specifies the type of control to be placed on the application window (text, password, checkbox, radio, button)
<code>name</code>	Specifies the name assigned to the control, allowing it to be referenced programmatically
<code>value</code>	Either the label displayed by a control or the default data assigned to the control
<code>onClick</code>	The name of a function or subroutine called by the control when the control is selected

© Jerry Lee Ford, Jr. All Rights Reserved.

Adding Text Controls

As demonstrated, if you place an `<input>` tag in the `<body>` `</body>` section of an HTA and assign its `type` attribute a value of `text`, a single-line text control is added to the application window. The text control is made up of a rectangular text field with an inset border. Text controls are used to collect small amounts of text input. The following example demonstrates how to add text control to an HTA.

```
<p>
  <label for = "userName">Enter your name:</label>
  <input type="text" name="textBox" id = "userName" size="50"
        maxLength = "50">
</p>
```

Here, the `<input>` tag includes a number of additional optional attributes that further configure the resulting text control. These attributes are explained here:

- **size.** This specifies the width (in number of characters) of the text control.
- **maxLength.** This specifies the total number of characters of input (including blank spaces) that the text control can store.

The following HTA demonstrates the use of a text and button control:

```
<html>

<head>
  <title>Text Box Demo</title>
  <HTA:APPLICATION
    ID="htaTextBoxApp"
    APPLICATIONNAME=Text Box Demo"
    SCROLL="auto"
    SINGLEINSTANCE="yes"
  >
</head>

<script language="VBScript">
  Sub Window_OnLoad
    self.resizeTo 600, 300
  End Sub

  Sub ShowResult
    MsgBox "Hello " & textBox.value
    textBox.value = ""
  End Sub
</script>

<body>
  <p>
    <label for = "userName">Enter your name:</label>
    <input type="text" name="textBox" id = "userName" size="50"
      maxLength = "50">
  </p>
  <input type="button" value="Click me!" onClick="ShowResult">
</body>

</html>
```

Figure 14.9 shows how the HTA generated by the previous example looks when executed on a computer running Windows 7.



Figure 14.9 An HTA whose GUI includes a text control and a button control. © 2014 Cengage Learning.

Adding a Password Control

A *password control* is a specialized type of text control that masks its input using asterisk or bullet characters to prevent it from being seen by spying eyes. To add a password control to an HTA, add an `<input>` tag and assign its `type` attribute a value of `password`, as shown here.

```
<p>
  <label for = "passwordBox">Enter your password:</label>
  <input type = "password" name = "passwordBox" id="passwordBox" size = "20"
    maxlength = "10" >
</p>
```

Figure 14.10 show how the previous example looks on a computer running Windows 7 when used to provide an HTA with its GUI. Note that nine characters have been typed into the password control, none of which are visible.

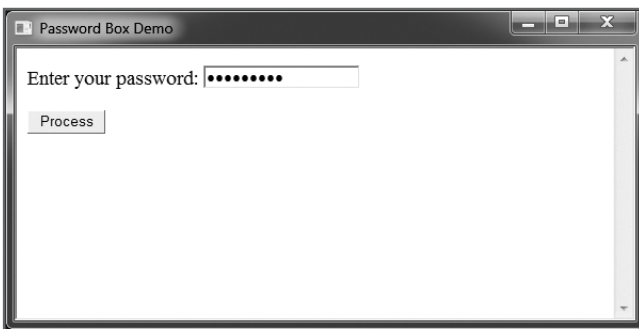


Figure 14.10 An HTA whose GUI includes a password control. © 2014 Cengage Learning.

Adding a Checkbox Control

Checkbox controls provide another way of collecting input. A checkbox control consists of a small square, which users can click to select or unselect it. When selected, a checkbox control displays a checkmark or similar character inside the control. An unselected checkbox appears as an empty box.

Checkbox controls can be used to display a list of choices. Each checkbox control works independently of other checkbox controls. You create a checkbox control by adding an `<input>` tag to an HTA and setting its `type` attribute to `checkbox`. Each checkbox control must be assigned a unique name and value. Optionally, you can pre-check a checkbox control by assigning the `input` element's `checked` attribute a value of `checked`.

The following example demonstrates the use of the checkbox control in an HTA. Note that the five `<input>` tags defined in the `<body>` `</body>` section are each followed by a text string that mirrors their assigned value. It is this trailing text that is displayed in the HTA window.

```
<html>

<head>
  <title>Checkbox Demo</title>
  <HTA:APPLICATION
    ID="htaCheckBoxApp"
    APPLICATIONNAME="Checkbox Demo"
    SCROLL="auto"
    SINGLEINSTANCE="yes"
  >
</head>

<script language="VBScript">
  Sub Window_OnLoad
    self.resizeTo 600, 300
  End Sub

  Sub ShowResult
    strChoices = vbCrLf
    If ChkBox1.Checked then strChoices = strChoices & "* " & _
      ChkBox1.value & vbCrLf
    If ChkBox2.Checked then strChoices = strChoices & "* " & _
      ChkBox2.value & vbCrLf
    If ChkBox3.Checked then strChoices = strChoices & "* " & _
      ChkBox3.value & vbCrLf
    If ChkBox4.Checked then strChoices = strChoices & "* " & _
      ChkBox4.value & vbCrLf
    If ChkBox5.Checked then strChoices = strChoices & "* " & _
      ChkBox5.value
    MsgBox "You selected: " & strChoices
  End Sub
</script>
```

```

<body>
  <label>Pick your toppings:</label>
  <p>
    <input type="checkbox" name="ChkBox1" value="Cheese"
      checked> Cheese
    <input type="checkbox" name="ChkBox2" value="Pepperoni"> Pepperoni
    <input type="checkbox" name="ChkBox3" value="Sausage"> Sausage
    <input type="checkbox" name="ChkBox4" value="Ham"> Ham
    <input type="checkbox" name="ChkBox5" value="Veggie"> Veggie
  </p>
  <input type="button" value="Process" onClick="ShowResult">
</body>

</html>

```

The ShowResult subroutine, located inside the <script> </script> tags, is used to inspect each of the checkbox controls and determine which ones were selected. Figure 14.11 shows an example of this HTA in action on a computer running Windows 7.

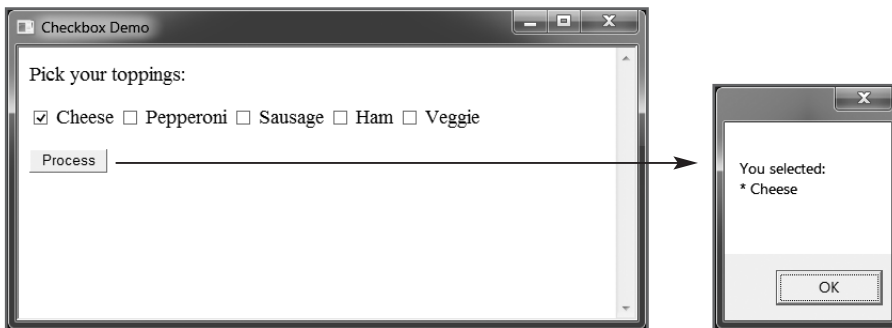


Figure 14.11 An example of the HTA in operation and the resulting application output.

© 2014 Cengage Learning.

Adding a Radio Button Control

Like checkbox controls, radio button controls allow for the selection of preconfigured options. They differ from checkbox controls in that radio button controls are designed to be organized into groups. This is accomplished by assigning the same name to every radio button control that is in the same group. Also, unlike checkbox controls, radio button controls are mutually exclusive, meaning that you can choose only one radio button out of the group.

Like checkbox controls, you should use the <input> tag's value attribute to assign a value to a radio control. This is necessary to be able to identify which radio button is selected. If beneficial, you can use the checked attribute to pre-select a given radio button control within its group.

The following example demonstrates how to work with the radio button control:

```
<label>Pick a color:</label>
<p>
  <input type="radio" name="radioButton" value="Blue"
    checked> Blue<br>
  <input type="radio" name="radioButton" value="Red"> Red<br>
  <input type="radio" name="radioButton" value="Green"> Green<br>
  <input type="radio" name="radioButton" value="Yellow"> Yellow<br>
  <input type="radio" name="radioButton" value="Orange"> Orange
</p>
<input type="button" value="Process" onClick="ShowResult">
```

Hint

The `
` tag forces a single line of elements to break into two lines of elements at the location of the tag, enabling you to add extra blank space where needed to improve presentation in your HTA GUIs.

Here, a group of five radio buttons named `radioButton` has been defined. The first radio button in the group has been pre-selected. The last statement in this example adds a button control to the HTA, which, when clicked, execute the `ShowResult` subroutine. This subroutine is shown next. When executed, it uses a loop to determine which radio button control has been selected. The result is displayed in a `MsgBox()` pop-up dialog box.

```
Sub ShowResult
  For Each i in radioButton
    If i.Checked Then
      MsgBox "You selected: " & i.value
    End If
  Next
End Sub
```

Figure 14.12 shows an example of this HTA in action on a computer running Windows 7.

Adding a Button Control

As has been the case with all the controls that have been examined so far, you can use the `<input>` tag to add a button control to an HTA. To do so, all you have to do is set the `<input>` tag's `type` attribute to `button`. Button controls usually display text indicating their purpose, which you can assign by setting the `<input>` tag's `value` attribute. Button controls have a single purpose: to initiate an `onClick` event when clicked. By assigning a subroutine or function name to a button control's `onClick` event, you can trigger its execution.

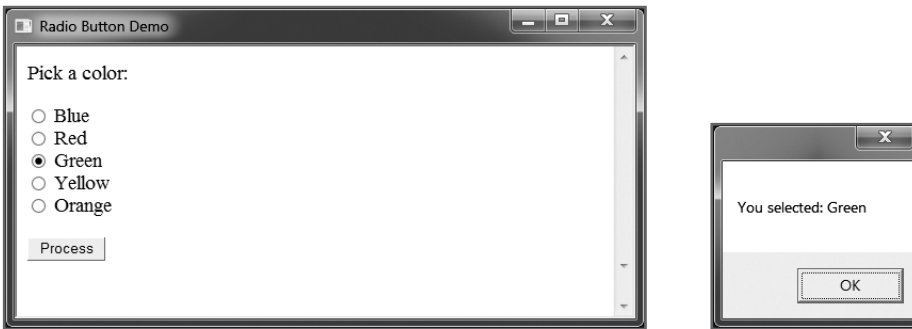


Figure 14.12 An example of the HTA in operation and the resulting application output.

© 2014 Cengage Learning.

The following example demonstrates how use the `<input>` tag to add a button control to an HTA.

```
<body>
  <input type="button" value="Click on Me" onClick="RunProgram">
</body>
```

Here, a single button control makes up the HTA's GUI. Its `onClick` setting has been configured to execute a subroutine named `RunProgram`. This subroutine is shown next. When executed, it displays a text message in a `MsgBox()` pop-up dialog box.

```
Sub RunProgram
  MsgBox "You clicked on the button!"
End Sub
```

Figure 14.13 shows the previous example in action on a computer running Windows 7.

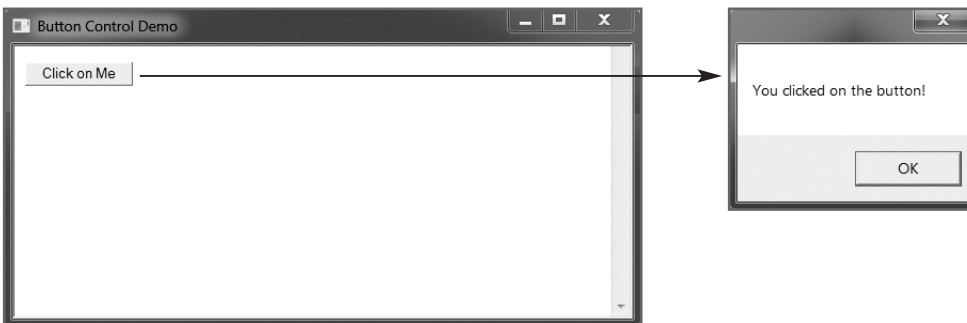


Figure 14.13 Buttons provide a means for manually initiating the execution of a specific subroutine or function in an HTA.

© 2014 Cengage Learning.

Adding a Button Control Using the <button> Tag

In addition to using the <input> tag to add button controls to you HTAs, you can use the <button> tag. To work with the <button> tag, all you have to do is assign its type attribute a value of button and configure its onClick event to call on a function or subroutine. Optionally, you can assign the button a name, enabling you to programmatically reference it if need be from within your VBScript.

The following example demonstrates how use the <button> tag to add a button to an HTA:

```
<body>
  <button onClick="RunProgram">Click on Me</button>
</body>
```

Both functionally and from a presentation standpoint, there is no outward difference in a button control added using the <button> tag versus one added using the <input> tag.

Adding a Multi-Line Text Control Using the <textarea> Tag

You learned how to add a text control and other types of controls by assigning a value of text to the <input> tag's type attribute. Text controls are great for collecting small amounts of text. But if you need to collect anything more than a single line of text, you will need to use the <textarea> tag, which creates a multi-line text field. The size of the field is specified by the numeric values assigned to its rows and cols attributes. If the amount of text entered exceeds the size of the control, the control's scrollbar is automatically enabled.

The following example demonstrates the usage of this control. Note that in this example, the control has been configured to be 10 rows high by 60 characters across.

```
<label>Type something here:</label>
<p>
  <textarea name="textBox" rows="10" cols="60"></textarea>
</p>
<input type="button" value="Process" onClick="ShowResult">
```

You can programmatically access the contents typed into this control by specifying the control's assigned name and its value property, as demonstrated in the following subroutine:

```
Sub ShowResult
  MsgBox "You typed: " & textBox.value
  textBox.value = ""
End Sub
```

If necessary, the control can be pre-populated with text, as shown here:

```
<label>Type something here:</label>
<p>
```

```

<textarea name="textBox" rows="10" cols="60">
  This text will be displayed within the multi-line text box control
when it is initially displayed.</textarea>
</p>
<input type="button" value="Process" onClick="ShowResult">

```

Figure 14.14 shows an example of a pre-populated multi-line text control on a computer running Windows 7.



Figure 14.14 A multi-line text control enables the collection of large amounts of text input.

© 2014 Cengage Learning.

Working with List Controls

Another type of useful controls are list-based controls. These include the list box control, the drop-down list control, and the multi-line list control. As with checkbox and radio button controls, list-based controls display a pre-defined list of options from which the user can select.

Adding a List Box

A list box control displays a pre-defined list of options from which a single selection can be made. Selection of an item from the list can be used to trigger an event. However, it is more common to *not* trigger an event in this manner, and instead to add a button control to trigger the event. This allows the user to make a selection, change his mind, and then make a *different* selection before initiating the processing of the selection.

A list box is added to an HTA using a combination of the `<select>` and `<option>` tags. The first step in creating a list box control is to use the `select` element to define the control using the format shown here:

```

<label>Pick Your Size</label>
<p>
  <select size="4" name="PickSize" onChange="ShowResult">
    </select>
</p>

```

As currently configured, any selection will trigger the execution of the ShowResult subroutine. The next step in the creation of a list box control is to populate it with items from which to choose. You do this by embedding multiple instances of the <option> </option> tags inside the <select> </select> tags. The <option> </option> tags can contain only text, as shown here.

```
<p>
  <label>T-shirt Size Options:</label>
  <select size="1" name="PickSize" onChange="ShowResult">
    <option value="">Pick your size</option>
    <option value="Small">Small</option>
    <option value="Medium">Medium</option>
    <option value="Large">Large</option>
    <option value="X-Large">X-Large</option>
  </select>
</p>
```

The following HTA provides a complete example of how to work with the list box control:

```
<html>

  <head>
    <title>List Box Demo</title>
    <HTA:APPLICATION
      ID="hta:ListBoxApp"
      APPLICATIONNAME="List Box Demo"
      SCROLL="auto"
      SINGLEINSTANCE="yes"
    >
  </head>

  <script language="VBScript">
    Sub Window_OnLoad
      self.resizeTo 600, 300
    End Sub

    Sub ShowResult
      If PickSize.value <> "" Then
        MsgBox "You selected " & PickSize.Value & "."
      End If
    End Sub
  </script>
```

```
<body>
  <p>
    <select size="4" name="PickSize" onChange="ShowResult">
      <option value="Small">Small</option>
      <option value="Medium">Medium</option>
      <option value="Large">Large</option>
      <option value="X-Large">X-Large</option>
    </select>
  </p>
</body>

</html>
```

Note that you can programmatically determine which item in the list box has been selected by referencing the `value` property of the control. Figure 14.15 shows an example of this HTA in action on a computer running Windows 7.



Figure 14.15 A list box control configured to trigger the execution of a script whenever an item is selected.

© 2014 Cengage Learning.

Adding a Multi-Selection List Box

The multi-selection list box is a variation of the list box control. The difference is that with a multi-selection list box, the user can select more than one item from the list. To do so, he needs only to hold down the `Alt` key while clicking different items in the list. The control is constructed exactly like the list box control except that a `size` attribute is specified and assigned a value of 2 or more, and a `multiple` keyword is added to the end of the opening `<select>` tag.

Unlike with the list box control, you cannot use the `onClick` event to trigger script execution. If you do, the script will execute as soon as the first selection is made. Instead, you should use another control such as a button control to initiate script execution. This will allow the user to select as many items as necessary from the list before initiating processing.

The following example shows how to add this control to an HTA:

```
<p>  
  <select size="4" name="PickTeam" multiple>  
    <option value="Small">Small</option>  
    <option value="Medium" selected="selected">Medium</option>  
    <option value="Large">Large</option>  
    <option value="X-Large">X-Large</option>  
  </select>  
</p>  
<input type="button" value="Process" onClick="ShowResult">
```

As shown next, you will need to process a multi-selection list box with a loop to determine which items have been selected:

```
Sub ShowResult  
  strChoices = vbCrLf  
  For Each i in PickTeam.Options  
    If i.Selected Then  
      strChoices = strChoices & i.Value & vbCrLf  
    End If  
  Next  
  MsgBox "You selected: " & strChoices  
End Sub
```

Figure 14.16 shows how this example would look if you were to turn it into an HTA and execute it on a computer running Windows 7.



Figure 14.16 A multi-selection list box control allows for the selection of one or more items from the list.

Adding a Drop-Down List Box

Another control that is commonly used is the drop-down list box. This control is also formulated using the `select` and `option` tags. Drop-down list boxes are space savers, allowing a lot of items to be displayed on demand, in a list that drops down, without requiring all the space that a list box control would otherwise need to display the same elements. The drop-down list box is formulated exactly like the list box except that the `size` attribute is set to 1. (Set it to anything higher, and you get a list box.)

The following example converts the list box presented earlier into a drop-down list box:

```
<p>
  <label>T-shirt Size Options:</label>
  <select size="1" name="PickSize" onChange="ShowResult">
    <option value="">Pick your size</option>
    <option value="Small">Small</option>
    <option value="Medium">Medium</option>
    <option value="Large">Large</option>
    <option value="X-Large">X-Large</option>
  </select>
</p>
```

Note the inclusion of an extra set of `<option>` `</option>` tags immediately following the opening `<select>` tag. Absent this extra item, the item that would otherwise appear first in the drop-down list box control is automatically made the default selection. By configuring the control with this extra item, you can programmatically determine whether the user has actually made a selection, as demonstrated when the control triggers the execution of the `ShowResult` subroutine:

```
Sub ShowResult
  If PickSize.value <> "" Then
    MsgBox "You selected " & PickSize.Value & "."
  End If
End Sub
```

Figure 14.17 shows how the drop-down list box looks on a computer running Windows 7 when initially selected on by the user.

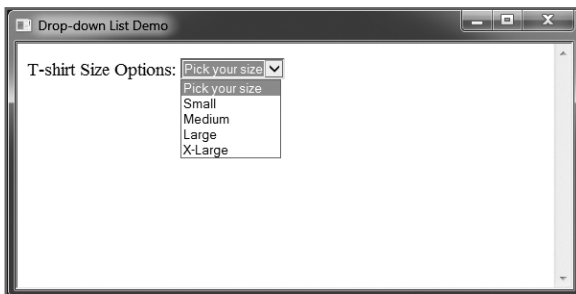


Figure 14.17 No action is taken until the user makes a valid selection from the list. © 2014 Cengage Learning.

Integrating WSH into Your HTAs

This far, the focus of this chapter has been teaching you the fundamentals of how to create HTA GUIs. With this accomplished, it is time to refocus your attention on the WSH and to demonstrate how to develop HTAs that integrate the WSH and WMI.

Starting Other Applications

The following HTA leverages the WSH to instantiate the `WScript.Shell` object. It can then execute the object's `Run` method to start the Notepad application and load a text file into it.

```
<html>
  <head>
    <title>Starting a Windows Application</title>
    <HTA:APPLICATION
      ID="htaStartWindowsApp"
      APPLICATIONNAME="Starting Notepad"
      SCROLL="auto"
      SINGLEINSTANCE="yes"
    >
    <script language="VBScript">
      Sub Window_OnLoad
        self.resizeTo 600, 300
      End Sub

      Sub RunProgram
        Set objShell = CreateObject("Wscript.Shell")
        objShell.Run "notepad.exe c:\scripts\test.txt"
      End Sub
    </script>
  </head>

  <body>
    <p> <button onclick="RunProgram">Run Program</button> </p>
  </body>
</html>
```

As you can see, a VBScript containing the WSH statements has been integrated into the HTA's `RunProgram` subroutine, which is executed when the button labeled "Run Program" is clicked. To test the execution of this HTA, create a text file named `test.txt` containing the following text and store it in a folder named `scripts` located on your computer's C: drive.

This text file has been opened by a WSH script embedded within an HTML Application.

Now, run the HTA and then click the button control. In response, Notepad will start and load the file. Figure 14.18 shows the results as displayed on a computer running Windows 7.

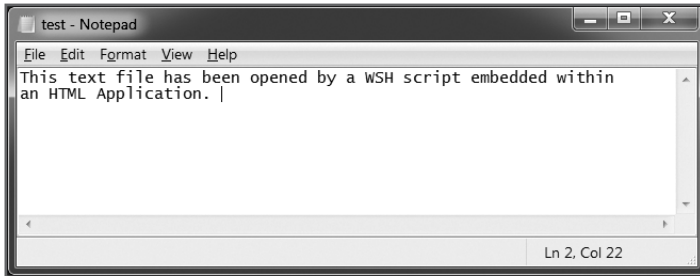


Figure 14.18 An HTA that includes a subroutine containing a VBScript that leverages WSH objects and methods.

© 2014 Microsoft Corporation.
Used with permission from Microsoft.

Hint

You can use the `WshShell` object's `ShellExecute` method instead of the `Run` method to start Notepad (or any other Windows application) and have it open the test file. To do so, simply replace the HTA's `RunProgram` subroutine with the subroutine provided here:

```
Sub RunProgram
    Set objShell = CreateObject("Shell.Application")
    objShell.ShellExecute "notepad.exe", "C:\scripts\test.txt", , , 1
End Sub
```

Using WMI to Capture Process Information

As demonstrated, integrating WSH into an HTA is a straightforward process. Let's take a look at a more involved example—one that leverages WMI and its ability to retrieve detailed information from the operating system. I should point out that WMI is not part of WSH; it is its own technology. So technically, what follows is not a WSH integration example. This HTA, shown here, begins with the `<HTA:APPLICATION>` tag and a little CSS, which applies text type, size, weight, and color formatting to the text displayed on the HTA window.

The HTA's programming logic is made up of two subroutines:

- The first resizes the HTA's window when the application is started and then uses the window object's `setInterval` method to execute the `GetServiceData` subroutine every second. As a result, the HTA will continuously display updated process data for as long as it runs.
- The second uses WMI to retrieve process data from the operating system. A display string is formulated and application output is then written to a pair of `` tags, displaying it directly on the HTA window.

```
<html>
<head>
  <title>Process Monitor Application</title>
  <HTA:APPLICATION
    ID="htaProcessMonitor"
    APPLICATIONNAME="Process Monitor"
    SINGLEINSTANCE="yes"
  >

  <style>
    body {
      /* Display all text in Courier font */
      font-family: Courier;
    }
    #ProcessOutPut {
      /* Set output font size to 9 points */
      font-size: 9pt;
    }
    #TotalOutPut {
      /* Display Total Process count */
      font-size: 12pt;
      /* in 12 point, bold, red font */
      font-weight:bold;
      color: #FF0000;
    }
  </style>
</head>

<script language="VBScript">

  'This subroutine resizes the HTA window and schedules
  'the execution of the GetServiceData subroutine to run once per second
  Sub Window_OnLoad
    self.resizeTo 400, 800
    serviceList = window.setInterval("GetServiceData", 1000)
  End Sub

  'This subroutine retrieves and displays process information
  Sub GetServiceData
    strProcessList = ""
    intCount = 0

    'Instantiate the WMI object and use it to collect process data
```

```

Set objWMI = GetObject("winmgmts:\\.\root\cimv2")
Set astrProcesses = objWMI.ExecQuery("Select * from Win32_Process")

'Loop though data and create string made up of process IDs and names
For Each strProcess in astrProcesses
    strProcessList = strProcessList & strProcess.ProcessId & " - " _
        & strProcess.name & "<br>"
    intCount = intCount + 1
Next

'Display output in the respective HTML span tags
ProcessOutPut.InnerHTML = strProcessList
TotalOutPut.InnerHTML = "Process Count = " & intCount
End Sub

```

```
</script>
```

```
<body>
```

```

<div id="ProcessOutPut"></div>    <!-- Display
process list here -->

```

```

<div id="TotalOutPut"></div>    <!-- Display
total count here -->

```

```
</body>
```

```
</html>
```

Note the use of the InnerHTML property to refer to the `` `` tags. By assigning the output string to InnerHTML, the new process data is automatically displayed within the tags, over-writing any previously displayed data. Figure 14.19 shows an example of this HTA in action on a computer running Windows 7.



Figure 14.19 This HTA auto-refreshes every second to provide up to date process data collected via WMI.

Other HTA Examples

For more HTA examples, check out Appendix A, “WSH Administrative Scripting.” There, you will find two additional HTAs, shown in Figure 14.20. One wraps the ping command inside an HTA GUI and another automates the shutdown of a Windows computer.

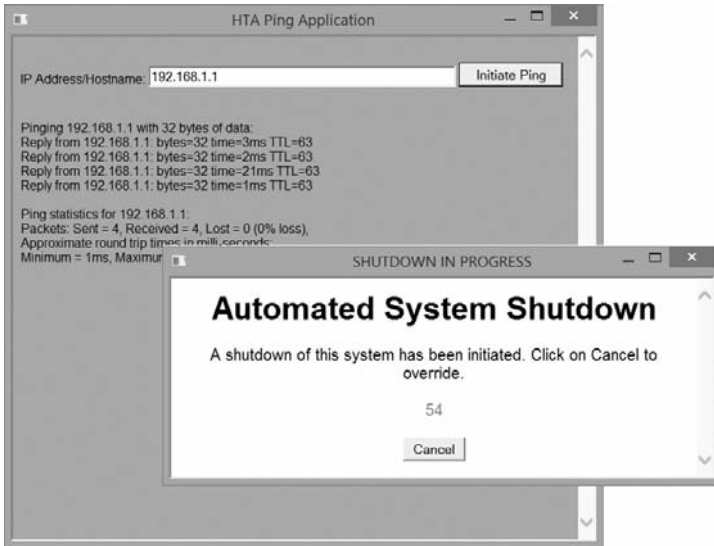


Figure 14.20 Additional HTA examples of greater complexity, shown here executing on Windows 8.1, are available in Appendix A.

© 2014 Cengage Learning.

Back to the Rock, Paper, Scissors Game

Now let’s return to where we began this chapter, by developing an HTA version of the Rock, Paper, Scissors game. As part of this project, you will modify the Rock, Paper, Scissors game discussed in Chapter 5. In doing so, you will convert the WSH script to an HTA, provide it with a suitable GUI, and enhance its presentation with a little CSS. Figure 14.21 provides a side-by-side comparison of the WSH and HTA versions of the Rock, Paper, Scissors game, as seen on a computer running Windows 7.

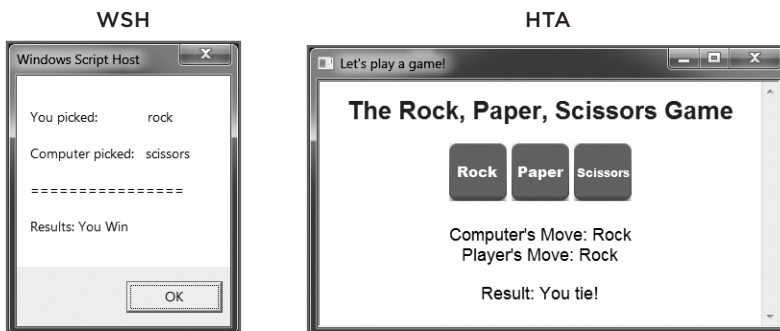


Figure 14.21 Side-by-side comparison of the WSH and HTA versions of the Rock, Paper, Scissors game.

© 2014 Cengage Learning.

Game Development

The following steps outline the process you'll go through to complete the development of the game:

1. Create a basic HTML file and add the `<HTA:APPLICATION>` tag to it.
2. Build the application's GUI by adding appropriate controls in the `<body>` `</body>` section.
3. Port over and convert the WSH Rock, Paper, Scissors game's programming logic.
4. Add a CSS to the application to further enhance its presentation.

Step 1: Start with a Basic HTML File

Begin this project by creating a new HTA named `RockPaperScissors.HTA`. Then add the following HTML to it. As you can see, there is nothing here other than the standard HTA template with a little tweaking of the `<HTA:APPLICATION>` tag.

```
<html>
  <head>
    <title></title>

    <HTA:APPLICATION
      ID="htaGame"
      APPLICATIONNAME="Rock Paper Scissors Game"
      SCROLL="auto"
      SINGLEINSTANCE="yes"
    >

    <style>
    </style>
  </head>

  <script language="VBScript">
  </script>

  <body>
  </body>
</html>
```

Step 2: Build the Application's GUI

Now it is time to add the HTML required to provide the new HTA with its GUI. Begin by adding the following statement in the `<body>` `</body>` section. This statement will place a level 1 heading on the HTA window that displays the name of the game in a large font.

```
<h1>The Rock, Paper, Scissors Game</h1>
```


Next, add the following statement to the `<body> </body>` section immediately following the previous statement. This statement places three image files, representing graphic buttons, inside the `<div> </div>` tags. These three graphic buttons will display side by side on the application window in the space set aside for the `<div> </div>` tags. Note that each of the embedded images is placed in the HTA window using the `<input>` tag while specifying a type of image. Also note the `onClick` event assignment for each `<input>` tag. These assignments initiate a new round of play by executing the `play()` function and passing it a text string specifying the player's move.

```
<div>
  <input type="image" src="rock.png" onClick="play(&quot;Rock&quot;);">
  <input type="image" src="paper.png" onClick="play(&quot;Paper&quot;);">
  <input type="image" src="scissors.png" onClick="play(&quot;Scissors&quot;);">
</div>
```

Hint

You can download copies of the `rock.png`, `paper.png`, and `scissors.png` files from this book's companion website, located at www.cengageptr.com/download. Search for this book's title to locate the site.

Instead of displaying game results in a pop-up dialog box, as was done in the WSH version of the game, the results of each round of play will be displayed in the HTA window. To set this up, add the following statements to the `<body> </body>` section immediately following the previous set of statements. The `<p> </p>` (paragraph) tags are used to pad a little blank space above and below the text embedded within them. This text includes two embedded pairs of ` ` tags, which are used to display the computer's and player's moves.

```
<p>
  Computer's Move: <span id="computer"> </span> <br>
  Player's Move: <span id="player"> </span>
</p>
```

Lastly, add the following statements to the end of the `<body> </body>` section. These statements display a text message showing the results of the game.

```
<p>
  Result: <span id="result"> </span>
</p>
```

Step 3: Port Over the Program Code

Now that you have created a new HTA file and built its GUI, it is time to modify and port over the application's programming logic as a function. Before you do that, however, you need to add one new piece of program code.

Its purpose is to resize the HTA's window at startup. To do this, add the following inside the <script> </script> tags:

```
Sub Window_OnLoad
    self.resizeTo 500, 300
End Sub
```

All the programming logic that controls the Rock, Paper, Scissors game will be placed within a single function named `play()`, which is provided here. Place this function inside the HTA's <script> </script> tags, immediately following the `Window_OnLoad` subroutine.

This function is executed each time the player clicks one of the game's three graphic controls, signaling a new round of play. The player's move is passed to the function as a text string argument. Next, a random number between 1 and 3 is generated, and is used to trigger the computer's move. A `Select Case` statement is then set up to determine the results of the game, comparing the player's and the computer's moves to determine who won. Once this analysis has been performed, the results of the game are displayed.

```
function play(strPlayerMove)

    'Time for the computer to randomly pick a choice
    Randomize
    intGetRandomNumber = Round(FormatNumber(Int((3 * Rnd) + 1)))

    If intGetRandomNumber = 3 then strComputerMove = "Rock"
    If intGetRandomNumber = 2 then strComputerMove = "Scissors"
    If intGetRandomNumber = 1 then strComputerMove = "Paper"

    'Compare the computer's and the player's move
    Select Case strPlayerMove
        Case "Rock"
            If strComputerMove = "Rock" Then
                document.getElementById("result").innerHTML = "You tie!"
            End If
            If strComputerMove = "Scissors" Then
                document.getElementById("result").innerHTML = "You win!"
            End If
            If strComputerMove = "Paper" Then
                document.getElementById("result").innerHTML = "You lose!"
            End If
        Case "Scissors"
            If strComputerMove = "Paper" Then
                document.getElementById("result").innerHTML = "You win!"
            End If
```

```
If strComputerMove = "Scissors" Then
    document.getElementById("result").innerHTML = "You tie!"
End If
If strComputerMove = "Rock" Then
    document.getElementById("result").innerHTML = "You lose!"
End If
Case "Paper"
    If strComputerMove = "Rock" Then
        document.getElementById("result").innerHTML = "You win!"
    End If
    If strComputerMove = "Scissors" Then
        document.getElementById("result").innerHTML = "You lose!"
    End If
    If strComputerMove = "Paper" Then
        document.getElementById("result").innerHTML = "You tie!"
    End If
End Select

document.getElementById("computer").innerHTML = strComputerMove
document.getElementById("player").innerHTML = strPlayerMove

End Function
```

Note that instead of displaying game results as a preformatted text string in a pop-up dialog box as was done in the WSH version of the game, the `document.getElementById()` method is now used to retrieve a reference to the `` `` tags where player moves and the result of game play is displayed. The `innerHTML` property is then used to overwrite this data in those locations.

Step 4: Spruce Up the Application's Presentation with a Little CSS

At this point, your copy of the HTA Rock, Paper, Scissors game is complete and is ready to run. However, before you run it, let's spruce things up by enhancing its presentation with some CSS, as shown here:

```
h1 {
    color: MidnightBlue;
    text-align: center;
    font-family: Arial;
    font-size: 26px;
}
div {
    text-align: center;
}
```

```
p {
  text-align: center;
  font-family: Arial;
  font-size: 18px;
}
```

Place these statements inside the `<style> </style>` tags, located in the HTA's `<head> </head>` section. As you can see, there are three CSS rules here:

- The first rule applies to the content of the `<h1> </h1>` tags, centering it and displaying it in an Arial font that is 26 pixels high.
- The second rule centers the display of the three graphical controls displayed in the HTA `<div> </div>` tags.
- The third CSS rule center-aligns text stored within paragraph tags, displaying it in an Arial font that is 18 pixels high.

The Fully Assembled Script

Okay, you have all the information you need to create an HTA version of the Rock, Paper, Scissors game. Once complete, start the game and put it through its paces. While you are at it, start up the WSH Rock, Paper, Scissors game and compare the look and feel of the two games. I think you will agree that the HTA version has definitely given the game a nice face lift!

Summary

In this chapter, you learned how to make the leap from command line to Windows desktop development by leveraging the technologies associated with HTA. You learned about the different components that make up HTAs and how to use them to wrap a GUI around your WSH scripts. This included leaning how to work with HTML, the `<HTA:APPLICATION>` tag, and CSS.

Challenges

1. This new version of the Rock, Paper, Scissors game is more inviting than its WSH counterpart. Consider enhancing its appearance even further by giving it a different background color or by using a more visually appealing font.
2. Another way to improve the Rock, Paper, Scissors game is to enhance its graphics. Consider replacing the game's three graphic buttons with graphics that are more eye popping.
3. The text string "Let's play a game!" is displayed in the title bar of the game window. Consider changing this text during game play to reflect the state of the game.

This page intentionally left blank

IV Appendices

Appendix A: WSH Administrative Scripting

Appendix B: Introducing Remote WSH

Appendix C: The WSH Core Object Model

Appendix D: Built-in VBScript Functions

**Appendix E: What's on the Companion
Web Site?**

This page intentionally left blank

A

WSH Administrative Scripting

In this book, you learned a great deal about both VBScript and the WSH by developing computer games. In the real world, of course, VBScript and the WSH are used to automate tasks. These tasks are typically mundane, repetitive, and time-consuming, or extremely complex and therefore subject to human error. Automating such tasks using VBScript and the WSH makes perfect sense. The purpose of this appendix is to provide you with a collection of sample scripts that demonstrate some real-world tasks that can be scripted.

None of the VBScripts that you will see in this appendix should be considered finished products. For example, you won't see any complex programming logic or a lot of error checking. These scripts were developed on computers running Windows 7 and Windows 8.1; you should review and test the scripts before running them on other operating systems. My intention for providing these sample scripts is to give you a feel for some of the real-world tasks that you can automate using VBScript and the WSH. I wanted to provide you with a collection of starter scripts from which you can begin to create and develop your own collection of scripts. Also included in this appendix are a pair of HTML Applications (HTAs) that demonstrate the execution of WSH administrative scripts that have been given a Windows graphical user interface.

I won't spend a lot of time going over the development of these scripts, nor will I attempt to explain every operation they perform. By now, you should be able to look at each of these scripts and determine what it is doing. To help you out a little, I made sure to include plenty of comments.

Desktop Administration

The administration of the Windows desktop on a single computer isn't terribly time-consuming. However, for those responsible for the maintenance and care of a large number of computers, scripting is a godsend. For example, a lot of small companies purchase their computers directly from the manufacturer. These computers arrive with the operating system already installed. However, desktop settings such as the color of the Windows desktop background or screensaver settings will vary depending on how the computer manufacturer chose to set them up.

Companies often try to keep the configuration of their computer settings standardized. This makes maintaining their computers easier and reduces a lot of user confusion. One way of configuring computers in this scenario is to develop VBScripts that automate the configuration of desktop settings according to company policy. Then, all you need to do to prepare a new computer for deployment is to copy over the scripts and have the user run them the first time he logs on to the computer.

Configuring the Desktop Background

The following VBScript demonstrates how to use the `WshShell` object's `RegWrite()` method to configure values that are stored in the Windows Registry and affect the Windows desktop background:

```
'*****  
'Script Name: Background.vbs  
'Author: Jerry Ford  
'Created: 02/25/14  
'Description: This script changes the user's background selection to none  
'and sets the default background color to white.  
'*****  
  
'Initialization Section  
  
Option Explicit  
On Error Resume Next  
  
Dim objWshShl, intChangeSettings  
Set objWshShl = WScript.CreateObject("WScript.Shell")  
  
'Main Processing Section  
  
'Verify that the user intends to change his screensaver settings  
intChangeSettings = PromptForConfirmation()
```

```
If intChangeSettings = 6 Then
    ModifySettings()
End If

WScript.Quit()

'Procedure Section

'This function determines if the user wishes to proceed
Function PromptForConfirmation()
    PromptForConfirmation = MsgBox("Set standard desktop background?", 36)
End Function

'This subroutine alters screensaver settings
Sub ModifySettings()
    'Turn off the wallpaper setting
    objWshShl.RegWrite "HKCU\Control Panel\Desktop\Wallpaper", ""

    'Setting the background color to white
    objWshShl.RegWrite "HKCU\Control Panel\Colors\Background", "255 255 255"
End Sub
```

The script begins by prompting for confirmation and then proceeds to modify the following Registry values:

```
objWshShl.RegWrite "HKCU\Control Panel\Desktop\Wallpaper", ""
objWshShl.RegWrite "HKCU\Control Panel\Colors\Background", "255 255 255"
```

The first statement sets the Windows Wallpaper setting to "". This is equivalent to right-clicking the Windows desktop, selecting Properties, and then setting the Background setting on the Desktop tab of the Windows Display Properties dialog box to None.

The second statement sets the value of the Background setting to 255 255 255. This is the equivalent of selecting white as the color setting on the Desktop tab.

To test this script, run it and then log off and on again.

Configuring the Screensaver

The following VBScript demonstrates how to change the configuration of the Windows screensaver. The overall construction of this script is very similar to the previous example, the only difference being which Registry keys are edited.

```
*****
'Script Name: ScreenSaver.vbs
'Author: Jerry Ford
'Created: 02/25/14
'Description: This script changes the user's screensaver to a default
'collection of settings.
*****

'Initialization Section

Option Explicit
On Error Resume Next

Dim objWshShl, intChangeSettings
Set objWshShl = WScript.CreateObject("WScript.Shell")

'Main Processing Section

'Verify that the user intends to change his screensaver settings
intChangeSettings = PromptForConfirmation()

If intChangeSettings = 6 Then
    ModifySettings()
End If

WScript.Quit()

'Procedure Section

'This function determines if the user wishes to proceed
Function PromptForConfirmation()
    PromptForConfirmation = _
        MsgBox("Set standard screensaver settings?", 36)
End Function

'This subroutine alters screensaver settings
Sub ModifySettings()
    'Enables the Windows screensaver
    objWshShl.RegWrite "HKCU\Control Panel\Desktop\ScreenSaveActive", 1
```

```
'Turns on password protection
objWshShl.RegWrite "HKCU\Control Panel\Desktop\ScreenSaverIsSecure", 1

'Establishes a 10-minute inactivity period (600 seconds)
objWshShl.RegWrite "HKCU\Control Panel\Desktop\ScreenSaveTimeOut", 600

'Enables the Mystify screensaver
objWshShl.RegWrite "HKCU\Control Panel\Desktop\SCRNSAVE.EXE", _
    "C:\Windows\System32\Mystify.scr"
```

End Sub

As you can see, this script modifies four Registry values. The modification of the first value enables the Windows screensaver. The modification of the second value enables screensaver password protection, which means that if the screensaver kicks in, the user has to retype his password to get back into Windows. The third modification sets up the screensaver to begin running after a 10-minute period of user inactivity. Finally, the last modification selects the screensaver that is to be run. To test this script, run it and then log off and on again.

Network Administration

Network administration means many things to many people. For one thing, it may mean establishing connections to network drives so that a script can move, copy, create, and delete files and folders residing on network computers. Network management also means establishing or removing connections to network printers. In the next several sections, I'll provide you with scripts that demonstrate how to connect to, and disconnect from, network drives and printers.

Mapping Network Drives

When you create a connection to a network drive (known as *mapping*), you make the network drive look as if it were local to your computer by assigning it a local drive letter—that is, as long as you have the appropriate set of security permissions on the network drive. Connecting to and disconnecting from a network drive is achieved using methods belonging to the `WshNetwork` object.

To create a drive mapping, you must use the `WshNetwork` object's `MapNetworkDrive()` method:

```
WshNetwork.MapNetworkDrive letter, name, [persistent], [username],
[password]
```

letter is an available logical disk drive letter on your computer. *name* is the universal naming convention (UNC) name and network path of the network drive. *persistent* is optional; it determines whether the mapping is permanent. A value of `True` creates a permanent mapping. The default value of this setting is `False`, which causes the connection to last only for the current working session. *username* and *password* are optional and are used to supply the username and password required to access the drive.

Trap

When you run scripts from the Windows desktop or command line, they execute using your security credentials. However, if you schedule the execution of your VBScript, then your scripts will not have the authority that you have and will be unable to establish a network drive connection. One way to get around this is to embed a username and password inside your script. However, doing so is really bad for security. Another option is to set up your script to prompt for a valid username and password at execution time and authorize someone who might be around to supply these credentials.

The following VBScript demonstrates how to establish a temporary network drive mapping:

```

*****
'Script Name: DriveMapper.vbs
'Author: Jerry Ford
'Created: 02/25/14
'Description: This script demonstrates how to add logic to VBScripts in
'order to support network drive mapping.
*****

'Initialization Section

Option Explicit
On Error Resume Next

Dim objWshNet

'Instantiate the objWshNetwork object
Set objWshNet = WScript.CreateObject("WScript.Network")

'Main Processing Section

'Call the procedure that maps drive connections, passing it an available
'drive letter and the UNC pathname of the drive
MapNetworkDrive "X:", "\\HP-PC\Scripts"

WScript.Quit() 'Terminate script execution

'Procedure Section

'This subroutine creates network drive mappings

```

```
Sub MapNetworkDrive(DriveLetter, NetworkPath)
    'Use the objWshNetwork object's MapNetworkDrive() method to map to drive
    objWshNet.MapNetworkDrive DriveLetter, NetworkPath
End Sub
```

Disconnecting Mapped Drives

You can use the `WshNetwork` objects' `RemoveNetworkDrive()` method to disconnect a mapped drive when it's no longer needed. For example, you might want to do this at the end of the script that created the drive mapping, after it has completed its assigned task. The syntax of the `RemoveNetworkDrive()` method is as follows:

```
WshNetwork.RemoveNetworkDrive letter, [kill], [persistent]
```

letter is the drive that has been assigned to the mapped drive. *kill* is an optional setting with a value of either `True` or `False`. Setting it to `True` disconnects a mapped drive even if it is currently in use. *persistent* is also optional. Set it to `True` to disconnect a permanently mapped drive.

The following VBScript demonstrates how to disconnect the network drive that was mapped by the previous script:

```
'*****
'Script Name: DriveBuster.vbs
'Author: Jerry Ford
'Created: 02/25/14
'Description: This script demonstrates how to add logic to VBScripts in
'order to terminate a network drive mapping.
'*****

'Initialization Section

Option Explicit
On Error Resume Next

Dim objWshNet

'Instantiate the objWshNetwork object
Set objWshNet = WScript.CreateObject("WScript.Network")

'Main Processing Section

'Call procedure that deletes network drive connections, passing it
'the drive letter to be removed
```

```
MapNetworkDrive "X:"
```

```
WScript.Quit() 'Terminate script execution
```

```
'Procedure Section
```

```
'This subroutine disconnects the specified network drive connection
```

```
Sub MapNetworkDrive(DriveLetter)
```

```
    'Use the objWshNetwork object's RemoteNetworkDrive() method to disconnect
```

```
    'the specified network drive
```

```
    objWshNet.RemoveNetworkDrive DriveLetter
```

```
End Sub
```

Printer Administration

Printer administration involves many tasks. One task is setting up network printer connections. Others include managing print jobs and physically managing the printer, including refilling its paper, ink, ribbon, or toner supply. Another task includes removing printer connections when they are no longer needed. The next two sections demonstrate how to use VBScript and the WSH to set up and disconnect network printer connections.

Connecting to a Network Printer

To create a connection to a network printer, you need to use the `WshNetwork` object's `AddWindowsPrinterConnection()` method. This method has two different types of syntax, depending on the operating system on which the script is executed.

The syntax for the `AddWindowsPrinterConnection()` method, when used on a computer running Windows 7 and Windows 8.1 is as follows:

```
WshNetwork.AddWindowsPrinterConnection(strPrinterPath)
```

strPrinterPath is the UNC path and name for the network printer.

The following VBScript demonstrates how to set up a network printer connection:

```
*****
'Script Name: PrinterMapper.vbs
'Author: Jerry Ford
'Created: 02/25/14
'Description: This script demonstrates how to use a VBScript to set up a
'connection to a network printer.
*****
```

```
'Initialization Section

Option Explicit

Dim objWshNet

'Instantiate the objWshNetwork object
Set objWshNet = WScript.CreateObject("WScript.Network")

'Main Processing Section

'Call the procedure that creates network printer connections, passing
'it a port number and the UNC pathname of the network printer
SetupNetworkPrinterConnection "\\HP-PC\HPLaserJet"
WScript.Quit() 'Terminate script execution

'Procedure Section

Sub SetupNetworkPrinterConnection(NetworkPath)
    'Use the objWshNetwork object's AddWindowsPrinterConnection() method
    'to connect to the network printer
    objWshNet.AddWindowsPrinterConnection NetworkPath
End Sub
```

Disconnecting from a Network Printer

As with network drives, removing a printer connection is a little easier to do than connecting it initially. Printer connections need to be removed for a number of reasons. For example, every printer eventually breaks and must be replaced. Sometimes people move from one location to another, necessitating changes to printer connections. By scripting the setup and removal of printer connections, you can automate this process. To remove a network printer connection, you need to use the `WshNetwork` object's `RemovePrinterConnection()` method:

```
WshNetwork.RemovePrinterConnection resource, [kill], [persistent]
```

resource identifies the printer connection and may be either the connection's assigned port number or its UNC name and path. *kill* is an optional setting with a value of either `True` or `False`. Setting it to `True` disconnects a printer connection even if it is currently in use. *persistent* is also optional. Set it to `True` to disconnect a permanent printer connection.

The following VBScript demonstrates how to remove the printer connection established by the previous VBScript:

```
'*****  
'Script Name: PrinterBuster.vbs  
'Author: Jerry Ford  
'Created: 02/25/14  
'Description: This script demonstrates how to use a VBScript to disconnect  
'a network printer connection.  
'*****  
  
'Initialization Section  
  
Option Explicit  
  
Dim objWshNet  
  
'Instantiate the objWshNetwork object  
Set objWshNet = WScript.CreateObject("WScript.Network")  
  
'Main Processing Section  
  
'Call the procedures that disconnect network printer connections, passing  
'it the UNC pathname of the network printer  
SetupNetworkPrinterConnection "\\HP-PC\HPLaserJet"  
  
'Terminate script execution  
WScript.Quit()  
  
'Procedure Section  
  
Sub SetupNetworkPrinterConnection(NetworkPath)  
    'Use the objWshNetwork object's RemovePrinterConnection() method to  
    'disconnect from a network printer  
    objWshNet.RemovePrinterConnection NetworkPath, "True", "True"  
End Sub
```

Computer Administration

The term *computer administration* represents a very broad category of tasks. Rather than try to list or explain them all, I'll simply present you with two computer administration examples. The first example demonstrates how to use VBScript and the WSH to manage Windows services, and the second example demonstrates automated user account creation.

Managing Services

On computers running the Windows 7 and Windows 8.1 operating systems, much of the operating system's core functionality is provided in the form of services. These services perform tasks, such as managing Windows plug and play, handling the spooling of printer jobs, and administering the execution of scheduled tasks. By starting and stopping Windows services, you can enable and disable specific Windows functions—that is, control just what users can and cannot do.

You can use the Windows `net stop` and `net start` commands to stop and start Windows services. To execute these commands from within a VBScript, you can use the `WshShell` object's `Run()` method, as demonstrated in the following script:

Trap

This script requires that you execute it with elevated privileges. To do so, locate and right-click the Windows Command Prompt icon and choose Run as Administrator from the menu that appears. Then navigate to the folder where the script resides and execute it.

```

'*****
'Script Name: ServiceCycler.vbs
'Author: Jerry Ford
'Created: 02/25/14
'Description: This script demonstrates how to use VBScript to stop and
'start Windows services.
'*****

'Initialization Section

Option Explicit
On Error Resume Next

Dim objWshShl, strServiceToManage

```

```
'Instantiate the WshShell object
Set objWshShl = WScript.CreateObject("WScript.Shell")

'Main Processing Section

'Prompt the user to specify the name of the service to cycle
strServiceToManage = InputBox("What service would you like to cycle?")

'Call the procedure that stops a service
StopService(strServiceToManage)

'Pause for five seconds
WScript.Sleep(5000)

'Call the procedure that starts a service
StartService(strServiceToManage)

'Terminate script execution
WScript.Quit()

'Procedure Section

'This subroutine stops a specified service
Function StopService(ServiceName)
    objWshShl.Run "net stop " & ServiceName, 0, "True"
End Function

'This subroutine starts a specified service
Function StartService(ServiceName)
    objWshShl.Run "net start " & ServiceName, 0, "True"
End Function
```

Hint

The `ServiceCycler.vbs` script uses the Windows `net stop` and `net start` commands to stop and start services. You can also use the WMI to control service status. An example of how to stop and start services using the WMI was provided in Chapter 13, “Working with the Windows Management Instrumentation.”

User Account Administration

User administration involves many tasks, including creating, modifying, and removing user accounts from the computer or the Windows domain to which the computer is a member. To perform user account administration, you need to have administrative privileges within the context that the script will execute (that is, on the computer or at the domain level).

One way to create a new user account is with the Windows `net user` command.

Trick

You also can use the `net group` command to add a newly created user account into a global domain group account or the `net localgroup` command to add the user account to a local group.

For example, the following VBScript uses the `WshShell` object's `Run()` method. The `net user` command can be used to create a new user account on Windows 7 or Windows 8.1.

Trap

This script requires that you execute it with elevated privileges. To do so, locate and right-click the Windows Command Prompt icon and click `Run as Administrator` in the menu that appears. Once started in this manner, navigate to the folder where the script resides and execute it.

```

'*****
'Script Name: AccountCreator.vbs
'Author: Jerry Ford
'Created: 02/25/14
'Description: This script demonstrates how to use VBScript to create new
'user accounts.
'*****

'Initialization Section

Option Explicit
On Error Resume Next

Dim objFsoObject, objWshShl, strNewAccts, strAcctName

'Instantiate the FileSystemObject object
Set objFsoObject = CreateObject("Scripting.FileSystemObject")

```

```
'Instantiate the WshShell object
Set objWshShl = WScript.CreateObject("WScript.Shell")

'Specify the location of the file containing the new user account name
Set strNewAccts = _
    objFsoObject.OpenTextFile("C:\Temp\UserNames.txt", 1, "True")

'Main Processing Section

CreateNewAccts() 'Call the procedure that creates new user accounts
WScript.Quit()  'Terminate script execution

'Procedure Section

Sub CreateNewAccts() 'This procedures creates new accounts
    'Create a Do...While loop to process each line in the input file
    Do while False = strNewAccts.AtEndOfStream

        'Each line of the file specifies a unique username
        strAcctName = strNewAccts.ReadLine()

        'Create the new account
        objWshShl.Run "net user " & strAcctName & " " & strAcctName & _
            " /add", 0
    Loop

    'Close the input file
    strNewAccts.Close
End Sub
```

To make the script more flexible, it has been set up to use VBScript `FileSystemObject` methods, which enable it to open and retrieve a list of names from an external file called `UserNames.txt`, located in the `C:\Temp` folder. That way, the script can be used over and over again without any modification. All you need to do is modify the script's input text file.

Disk Management

VBScripts provide an excellent tool for automating the execution of various Windows system administration utilities. An example of one such utility is Windows Disk Cleanup. Like many Windows utilities, this utility provides a command-line interface, meaning you can control its execution via your scripts using the `WshShell` object's `Run()` method.

The Windows Disk Cleanup utility recovers lost disk space by deleting unnecessary files stored on the computer's disk drive. When executed, the Disk Cleanup utility deletes the following files:

- Files found in the Recycle Bin
- Temporary files
- Downloaded program files
- Temporary Internet files
- Catalog files for the Content Indexer
- WebClient/Publisher temporary files

Before you can automate the execution of the Disk Cleanup utility, you must perform a one-time configuration process as outlined here:

1. Click Start and then Run. The Run dialog box opens.
2. Type `cleanmgr /sageset:1` and then click on OK.
3. The Disk Cleanup Settings dialog box opens. Select the types of files that you want the Disk Cleanup process to remove and then click on OK.

After you've completed the configuration process, you can create your Disk Cleanup execution script using the following example as a template:

```

'*****
'Script Name: VBSCleanup.vbs
'Author:      Jerry Ford
'Created:     02/25/14
'Description: This script automates the execution of the Windows Disk
'              Cleanup utility.
'*****

'Initialization Section

Option Explicit

Dim objWshShl
Set objWshShl = WScript.CreateObject("WScript.Shell")

'Main Processing Section

ExecuteCleanupUtility()
RecordMsgToAppEventLog()

```

```
WScript.Quit() 'Terminate the script's execution
```

```
'Procedure Section
```

```
Function ExecuteCleanupUtility() 'Run the Windows Disk Cleanup utility  
    objWshShl.Run "C:\WINDOWS\SYSTEM32\cleanmgr /sagerun:1"  
End Function
```

```
Function RecordMsgToAppEventLog() 'Record message in Application event log  
    objWshShl.LogEvent 4, "VBSCleanup.vbs - Disk Cleanup has been started."  
End Function
```

When you create your script, make sure you specify the `/sagerun:1` parameter exactly as shown here:

```
objWshShl.Run "C:\WINDOWS\SYSTEM32\cleanmgr /sagerun:1"
```

Integrating VBScript with Other Applications

In addition to creating VBScripts that can interact with and control Windows resources, you can also create VBScripts that automate the execution of popular Windows applications such as Microsoft Word and WinZip. In this section, you'll see examples of how to use VBScript and the WSH to create a Word document and a ZIP file.

Automating the Generation of Microsoft Word Reports

To use VBScript to automate Word tasks, you need to know a little something about the Word object model. The `Application` object resides at the top of the Word object model. The `Application` object is automatically instantiated when Word is started. Using properties and methods associated with the `Application` object, you can access lower-level objects and collections in the Word model. Using the properties and methods associated with the lower-level objects, you can automate any number of Word tasks.

The following script provides a working example of how to use VBScript and the WSH to automate the creation of a Word document. Comments embedded within the script provide additional information about the Word object model.

Trick

The Word object model is far too large and detailed to be covered in this book. You can learn more about it here: <http://msdn.microsoft.com/office>.

```
*****
'Script Name: WordObjectModelExample.vbs
'Author:      Jerry Ford
'Created:     02/25/14
'Description: This script demonstrates how to integrate VBScript and
'             the Microsoft Word object model.
*****

'Initialization Section

Option Explicit
On Error Resume Next

Dim objWord 'Used to establish a reference to Word application object
Set objWord = WScript.CreateObject("Word.Application") 'Instantiate Word

'Main Processing Section

CreateNewWordDoc()
WriteWordReport()
SaveWordDoc()
CloseDocAndEndWord()
TerminateScript()

'Procedure Section

Function CreateNewWordDoc()
    'Documents is a collection. Add() is a method belonging to the Documents
    'collection that opens a new empty Word document
    objWord.Documents.Add()
End Function

Function WriteWordReport()
    'Specify Font object's Name, Size, Underline, and Bold properties
    objWord.Selection.Font.Name = "Arial"
    objWord.Selection.Font.Size = 16
    objWord.Selection.Font.Underline = True
    objWord.Selection.Font.Bold = True

    'Use the Selection object's TypeText() method to write text output
    objWord.Selection.TypeText("Sample VBScript Word Report")
End Function
```



```
'Use the Selection object's TypeParagraph() method to insert linefeeds
objWord.Selection.TypeParagraph
objWord.Selection.TypeParagraph
objWord.Selection.TypeParagraph
```

```
'Use the Font object's Underline and Bold properties
objWord.Selection.Font.Underline = False
objWord.Selection.Font.Bold = False
```

```
'Use the Font object's Size and Bold properties
objWord.Selection.Font.Size = 12
objWord.Selection.Font.Bold = False
```

```
'Use the Selection object's TypeText() method to write text output
objWord.Selection.TypeText("Prepared on " & Date())
```

```
'Use the Selection object's TypeParagraph() method to insert linefeeds
objWord.Selection.TypeParagraph
objWord.Selection.TypeParagraph
```

```
'Use the Selection object's TypeText() method to write text output
objWord.Selection.TypeText("Copyright - Jerry Lee Ford, Jr.")
```

```
End Function
```

```
Function SaveWordDoc()
```

```
'The Applications object's ActiveDocument property establishes a
'reference to the current Word document.
```

```
'The Document object's SaveAs() method provides the ability to save
'the Word file
```

```
'Save the new document to C:\Temp
objWord.ActiveDocument.SaveAs("c:\Temp\TextFile.doc")
```

```
End Function
```

```
Function CloseDocAndEndWord()
```

```
'Use the Document object's Close() method to close the document
objWord.ActiveDocument.Close()
```

```
'Terminate Word  
objWord.Quit()  
End Function  
  
Function TerminateScript()  
    WScript.Quit() 'Terminate script execution  
End Function
```

Figure A.1 shows how the Word document created by this script looks after it has been created.

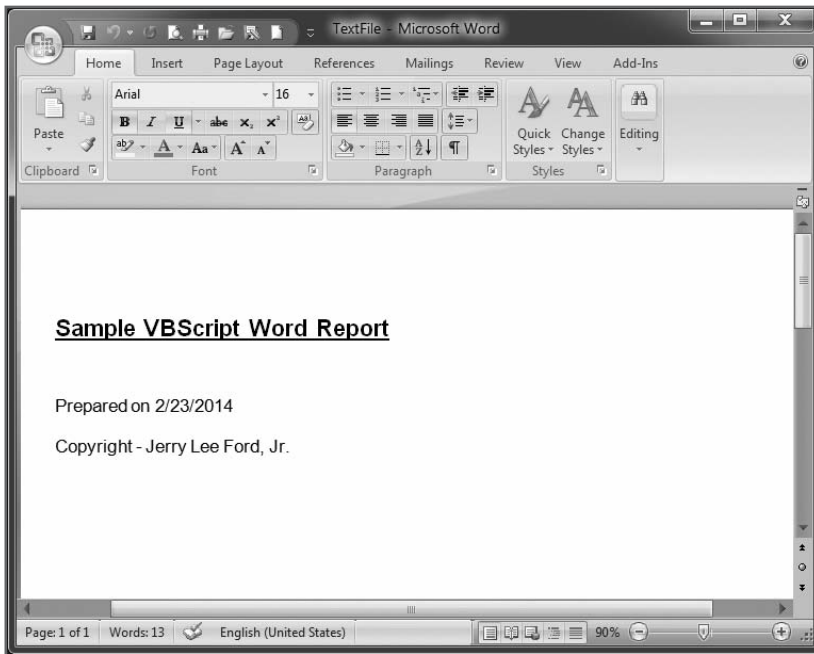


Figure A.1 Automating the creation of a Word document.

© 2014 Microsoft Corporation.
Used with permission from Microsoft.

Automating the Execution of Third-Party Applications

Using VBScript and the WSH, you can automate the functionality of any application that exposes its object model. However, not every Windows application does this. Instead, many applications provide the capability to automate functions via built-in command-line interfaces, meaning that you can send commands to the application, which the application then processes. A good example of one such application is WinZip. For example, you can send commands to WinZip by executing its WinZip32.exe program and passing it arguments. The following VBScript demonstrates how to create a script that automates the creation of a new ZIP file named VBScripts.zip. The syntax for WinZip32.exe is embedded as comments within the script.

Hint

For this script to run, you must have a registered copy of WinZip installed on your computer.

```
'*****  
'Script Name: WinZipDemo.vbs  
'Author: Jerry Ford  
'Created: 02/25/14  
'Description: This script creates a new ZIP file made up of all the  
' VBScripts found in the C:\VBScriptsGames folder.  
  
'*****  
  
'Initialization Section  
  
Option Explicit  
Dim intUserResponse, objWshShl  
  
'Instantiate the Windows shell object  
Set objWshShl = WScript.CreateObject("WScript.Shell")  
  
'Main Processing Section  
  
PromptForPermission()  
  
If intUserResponse = vbYes Then  
    CreateZipFile()  
End If  
  
TerminateScriptExecution()  
  
'Procedure Section  
  
Function PromptForPermission() 'Ask user for permission to continue  
    intUserResponse = MsgBox("This script creates a ZIP file containing" & _  
        " all the VBScripts found in C:\VBScriptGames." & vbCrLf & vbCrLf & _  
        "Do you wish to continue?", 36, "VBScript Zipper!")  
End Function
```

```
Function CreateZipFile() 'Create the new ZIP file
'WINZIP32 Command Syntax:
'WINZIP32 [-min] action [options] filename[.zip] files
' -min - Tells WinZip to run minimized
' action - Represents any one of the following arguments
'         -a Create new ZIP file
'         -f Refresh existing archive
'         -u Update an existing archive
'         -m Move archive to specified location
' options - Optional arguments that include
'         -r Add files and folders when adding to ZIP file
'         -p Include information about any added folders
' filename[.zip] - name of ZIP file to be created
' files - names of file to be added to the ZIP file

objWshShl.Run _
    "WINZIP32 -a C:\Temp\VBScripts.zip D:\VBScriptGames\*.vbs", 0, True
End Function

Function TerminateScriptExecution() 'Terminate the script's execution
    WScript.Quit()
End Function
```

HTML Applications

Chapter 14, “Adding a GUI to Your Scripts,” introduced you to HTML Applications (HTAs) and explained how you can use them as a means for providing a graphical user interface (GUI) for your scripts. This appendix closes with two sample HTAs. One demonstrates how to wrap a GUI around a WSH that executes and retrieves the results of the ping command. The other demonstrates how to wrap a WSH script that automates the shutdown of a computer inside a GUI, providing the user with a friendly notification of the pending shutdown and means for overriding it.

Wrapping a GUI Around a WSH ping Script

You can create HTAs for the purpose of wrapping a GUI around any number of command-line commands. In doing so, you can simplify their use and even make them available to non-technical people, who might otherwise be unable to make proper use of them. The following HTA provides a working example of how to wrap the ping command inside a GUI. Comments are embedded throughout the application to explain its construction. Figure A.2 shows an example of the HTA in action.

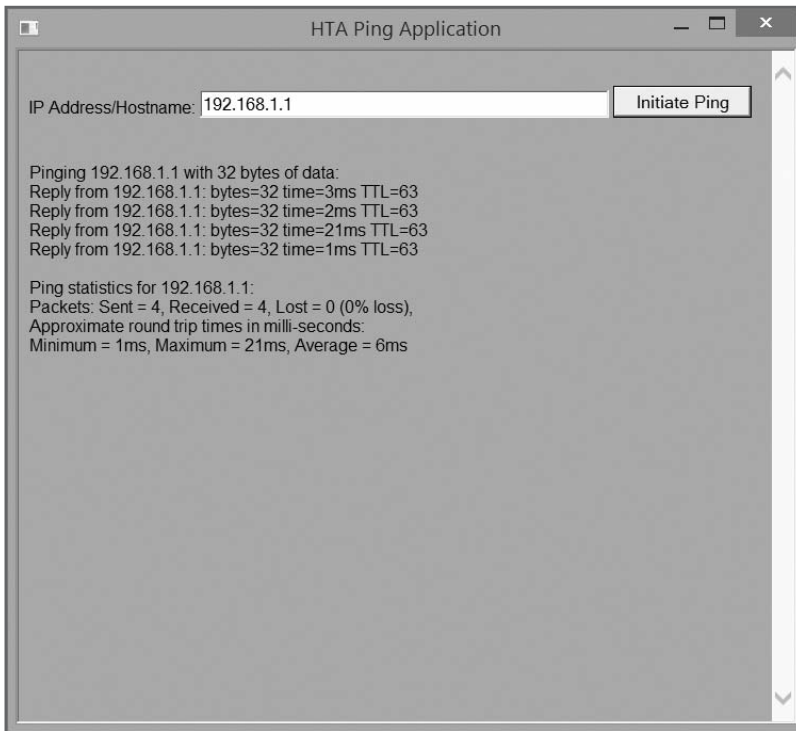


Figure A.2 Using an HTA to wrap a GUI around the ping command on a computer running Windows 8.1.

© 2014 Cengage Learning.

Begin Note

Note the different types of comments in the HTA. HTML comments begin with `<!--` and end with `-->`. CSS comments begin with `/*` and end with `*/`. VBScript comments begin with `'` and have no ending character.

```
<head>
```

```
<title>HTA Ping Application</title> <!--Specify title bar text-->
<HTA:APPLICATION ID="htaPingApp"
  APPLICATIONNAME="HTA Ping"
  BORDER="thin"
  BORDERSTYLE="normal"
  SINGLEINSTANCE="yes"
>
```

```
<style>
  body {
    background-color:#AAAAAA; /*Set style for body section*/
    font:16px "Arial"; /*Set background color to gray*/
  } /*Set font type size*/
```

```
    span {color:red;                /*Set color in the span tag*/
        font-weight:bold}          /*Set font-weight for span tag*/
</style>

<script type="text/vbscript">

Sub Window_OnLoad                  'Specify window size at load time
    self.resizeTo 750, 240
End Sub

Sub submitButton_onClick           'Subroutine that executes ping

    textOutput.innerHTML = ""
    self.resizeTo 750, 680        'Resize window to display data

    'Specify temporary text file to use to hold ping command results
    strTempFile = "C:\Temp\PingFile.txt"

    'Instantiate FileSystemObject and WshShell objects
    Set WshFSO = CreateObject("Scripting.FileSystemObject")
    Set WshShell = CreateObject("Wscript.Shell")

    'Execute ping command and pipe output to temporary file
    WshShell.Run "cmd.exe /c ping.exe " & textField.value & " > " & _
        strTempFile, 0, True

    'Open temporary file, loop through results, and build display string
    Set objTextFile = WshFSO.OpenTextFile(strTempFile, 1)
    Do While objTextFile.AtEndOfStream <> True
        strOutput = strOutput & objTextFile.ReadLine & "<br>"
    Loop

    'Display ping command output inside the <div></div> tags
    textOutput.innerHTML = strOutput

    objTextFile.Close             'Close the temporary file
End Sub

</script>
</head>
```

```

<body>
  <p>IP Address/Hostname: <input type="text" name="textField" size="50">
    <input type="button" name="submitButton" value="Initiate Ping"></p>
  <div id="textOutput">Please be patient when waiting for results. The
    ping command can take a while to execute. </div>
</body>

</html>

```

Automating Windows Shutdown

The following example demonstrates how to create an HTA that automates the shutdown of a Windows computer. This application begins by displaying a GUI window announcing that the automated shutdown process has begun. A 60-second countdown is then started. Unless interrupted, the shutdown process is initiated at the end of the countdown. The user may, however, click the Cancel button at any time during the 60-second countdown to halt it. Embedded comments document the constructions of the HTA. Figure A.3 shows an example of the HTA when executed on a computer running Windows 8.1.



Figure A.3 An example of the automated shutdown HTA in action. © 2014 Cengage Learning.

```

<html>
  <title>SHUTDOWN IN PROGRESS</title> <!--Specify title bar text-->
  <HTA:APPLICATION
    ID="htaShutdownApp"
    APPLICATIONNAME="Shutdown Demo"
    SCROLL="auto"
    SINGLEINSTANCE="yes"
  >

  <head>
    <style>
      body {

```

/*Set style for body section*/

```
    font-family: Arial;           /*Set font type*/
    text-align: center;          /*Set text alignment*/
}
#msgTitle {                      /*Set style for the title element*/
    margin: 12px 0 0;           /*Set margin to 12 pixels*/
}
#msgCountdown {                 /*Set Style for the countdown element*/
    color: #FF00FF;            /*Set color to purple*/
}
</style>

<script type="text/vbscript">
    Sub Window_OnLoad            'Subroutine that specifies window size
        self.resizeTo 700, 300
    End Sub

    Sub SystemShutdown          'Subroutine that shuts down Windows
        Set objSystemSet = GetObject _
            ("winmgmts:{impersonationLevel=impersonate, (Shutdown )}")_
            .InstancesOf("Win32_OperatingSystem")
        For Each objSystem In objSystemSet
            objSystem.Win32Shutdown 5
        Next
    End Sub

    Sub StopCountdown           'Subroutine that terminates the HTA
        Set WshShell = CreateObject("WScript.Shell")
        Window.Close
    End Sub
</script>
</head>

<body>
    <!--Display text as Level 1 heading-->
    <h1 id="msgTitle">Automated System Shutdown</h1>

    <!--Display instructions explaining how to halt automated shutdown-->
    <p id="msgText">A shutdown of this system has been initiated.
        Click on Cancel to override.</p>
```



```
<!--Displays countdown. Shutdown occurs when count reaches zero.-->
<div id="msgCountdown">
</div>

<!--Displays button used to call subroutine that terminates shutdown-->
<p>
  <input type="button" value="Cancel" name="btnCancel"
    onclick="StopCountdown">
</p>

<script type="text/vbscript">
  noOfIterations = 60          'Specify countdown length (in seconds)

  SixtySecondCountdown()     'Call subroutine that manages countdown

  'Subroutine responsible for counting down and starting system shutdown
  Sub SixtySecondCountdown

    'Pause application execution for one second
    timerProcess = _
      window.setTimeout("SixtySecondCountdown", 1000, "VBScript")

    'Keep count of the remaining number of pauses
    noOfIterations = noOfIterations - 1

    'Display the number of pauses remaining (e.g. the countdown)
    document.getElementById("msgCountdown").innerHTML = noOfIterations

    'When the number of pauses remaining is zero, stop counting and
    'execute the subroutine that shuts down the computer system
    If noOfIterations = 0 Then
      window.clearTimeout(timerProcess)
      SystemShutdown()
    End If
  End Sub
</script>

</body>
</html>
```

B

Introducing Remote WSH

One new WSH technology introduced in WSH 5.6 was Remote WSH. Remote WSH provides the ability to initiate and monitor the remote execution of scripts. Remote WSH initially faced some problems with deadlocks and error code reliability. These issues have been pretty much eliminated with Remote WSH 5.7. This appendix will introduce you to Remote WSH, explain its underlying architecture, and demonstrate its use.

Introducing Remote WSH

Remote WSH provides programmers with the ability to remotely start and monitor the execution of scripts across a network. Using data returned during script execution, you can programmatically react to remote script execution, terminating it if necessary. For you to be able to take advantage of Remote WSH, the following requirements must be met:

- The local computer must be running WSH version 5.6 or higher.
- The remote computer must be running WSH version 5.6 or higher.
- Both the local and remote computers must be running Windows 2000, XP, 2003, Vista, 2008, 2012, 7, or 8.
- You must have administrative privileges on the remote computer.

Trap

Remote WSH has several limitations. First, it cannot support the remote execution of statements that generate pop-up dialog boxes. These include the VBScript `MsgBox()` and `InputBox()` functions and the WSH `Echo()` and `Popup()` methods. In addition, Remote WSH scripts cannot access shared folders on remote computers. Finally, Remote WSH does not provide a means of returning output from remotely executed scripts.

To work with Remote WSH, it must be enabled on any computer where remote scripts will be executed. To enable it, you must add a new value named `Remote` to the following Registry key and set it equal to 1:

```
HKCU\Software\Microsoft\Windows Script Host\Settings\
```

In addition to modifying the Registry, you must also register the `WScript.exe` execution host as a remote COM server by executing the following command on the remote Windows computer:

```
WScript -regserver
```

Understanding Remote WSH's Supporting Architecture

Remote WSH is made up of three objects, each of which supports various methods, properties, and events. Figure B.1 shows the relationships that these objects and their properties, methods, and events have to one another.

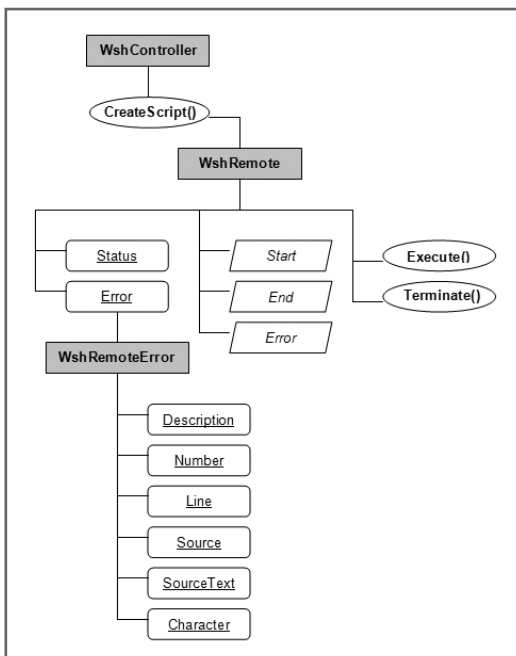


Figure B.1 The Remote WSH is made up of three high-level objects and their associated methods, properties, and events. © 2014 Cengage Learning.

As shown in Figure B.1, the `WshController` object is the top-most of these three objects. The following statement demonstrates how to instantiate the `WshController` object:

```
Set WshControl = CreateObject("WshController")
```

The `WshController` object has a single method, `CreateScript()`, that is used when instantiating a new `WshRemote` object. The `CreateScript()` method has the following syntax:

```
ObjectReference.CreateScript(CommandLine, [ComputerName])
```

ObjectReference represents a reference to a `WshController` object. *CommandLine* is a text string specifying the location of the remote script and any switches that need to be included. *ComputerName* identifies the UNC name of the network computer on which the remote script is to run. If this is omitted, the script will be run locally.

The following statements demonstrate how to instantiate a `WshController` object named `RemoteScript` to copy and then load a script named `Test.vbs` to a remote computer named `DSKTP001`.

```
Set WshControl = CreateObject("WshController")
Set RemoteScript = WshControl.CreateScript(Test.vbs, DSKTP001)
```

Although this example instantiates a `WshRemote` object and copies a script to a remote computer, it does not execute the script. You will learn how to do this shortly.

Executing Remote WSH Methods

The `WshRemote` object supports two methods. The `Execute()` method provides the ability to initiate the remote execution of a script. The syntax supported by this method is outlined here:

```
ObjectReference.Execute
```

ObjectReference specifies the name of a `WshRemote` object. So to execute the `Test.vbs` script that was copied over to the remote computer in the previous example, you would need to execute the following statement:

```
RemoteScript.Execute
```

The second method supported by the `WshRemote` object is the `Terminate()` method. This method gives you the ability to halt a remote script's execution. This method has the following syntax:

```
ObjectReference.Terminate
```

ObjectReference specifies the name of a `WshRemote` object. So to halt the execution of the `Test.vbs` script on the `DSKTP001` computer, you could do so by executing the following statement:

```
RemoteScript.Terminate
```

Responding to WSH Remote Events

Remotely executed scripts generate any of three different events when they execute. These three types of events are outlined in Table B.1.

TABLE B.1 WSH REMOTE SCRIPT EVENTS

Event	Description
Start	Triggered when the remote script begins executing
End	Triggered when the remote script stops executing
Error	Triggered if the remote script experiences an error

© Jerry Lee Ford, Jr. All Rights Reserved.

You can capture and respond to these events. To capture these remote events, you must use the `WScript` object's `ConnectObject()` method, which has the following syntax:

```
ObjectReference.ConnectObject(TargetObject, EventPrefix)
```

ObjectReference represents the `WScript` object. *TargetObject* identifies the object for which the connection is being set up, and *EventPrefix* specifies the event's prefix.

Using the `ConnectObject()` method, you connect an object's events to a subroutine or function whose name contains the specified prefix. You must then create a procedure for each event you wish to capture. You do so by assigning each procedure a name that begins with the specified prefix followed by the underscore character and the event name. For example, the following statement would enable local script to capture events returned by a remote script using a prefix of `RemoteScript_`.

```
WScript.ConnectObject RemoteScript, "RemoteScript_"
```

Having established a means of capturing remote events, you next need to create event handlers for the events to which you want to be able to respond, as demonstrated here:

```
Function RemoteScript_Start()
    'Add statements here to process the start event.
End Function
```

In this example, an event handler has been set up to execute when a remote script starts its execution. You can set up similar event-handling procedures for both the `End` and `Error` events.

Accessing WSH Remote Properties

If an error occurs with a remotely executed script, the `Error` event will trigger, which you can capture by setting up an event handler. Using this procedure, you can evaluate error information by examining different properties belonging to the `WshRemoteError` object. WSH Remote automatically captures and stores error information available through this object.

If an error occurs, you can retrieve information about the error using the `WshRemote` object's `Error` property. This property is used to access to the `WshRemoteError` object, so that you can then access `WshRemoteError` object properties where detailed error information is stored. Table B.2 provides a listing of all the properties belonging to the `WshRemoteError` object.

TABLE B.2 REMOTE WSH ERROR PROPERTIES

Property	Description
Description	A description of error
Number	The numeric error code associated with the error
Line	The line number where the error occurred
Source	The object responsible for reporting the error
SourceText	The line of code that generated the error
Character	The character position in the line of code where the error occurred

© Jerry Lee Ford, Jr. All Rights Reserved.

In addition to the `Error` property, the `WshRemote` object also has a `Status` property you can use to keep track of the status of remote scripts as they execute. Table B.3 provides a list of the possible range of values supported by this property and explains their meaning.

TABLE B.3 WSH REMOTE EXECUTION STATUS

Value	Description
0	The remote script has not started executing yet.
1	The remote script is now executing.
2	The remote script has finished executing.

© Jerry Lee Ford, Jr. All Rights Reserved.

Working with Remote WSH: A Demonstration

To understand how to work with Remote WSH, it helps to see an example in action. The following example demonstrates how to remotely execute a VBScript named `Test.vbs` on a remote computer named `DSKTP001`.

```

*****
'Script Name: WSHRemoteDemo.vbs
'Author: Jerry Ford
'Created: 03/14/14
'Description: This script controls the remote execute of a VBScript named
'             Test.vbs on a computer named DSKTP001.
*****

'Initialization Section
Option Explicit

Set wshController = CreateObject("WshController")
Set wshRemote = wshController.CreateScript("Test.vbs", "\\DSKTP001")

'Main Processing Section

WScript.ConnectObject wshRemote, "RemoteScript_"
wshRemote.Execute

Do Until wshRemote.Status = 2
    WScript.Sleep 1000
Loop

'Procedure Section

Sub RemoteScript_Start()
    MsgBox "Test.vbs has started!"
End Sub

Sub RemoteScript_End()
    MsgBox "Test.vbs has terminated!"
End Sub

```

When executed, this script begins by setting up an instance of the `WshController` object. It then sets up an instance of the `WshRemote` object and copies the `Test.vbs` script into the memory of the `DSKTP001` computer. The `ConnectObject()` method is then used to define an event prefix, allowing the script to capture events returned during the remote execution of `Test.vbs`. The `WshRemote` object's `Execute()` method is then used to initiate the execution of the remote script. A loop is set up that executes every two seconds, stopping only when the value of the `WshRemote` object's `Status` property indicates that the remote script has finished executing.

Two subroutines are defined in the script's procedure section that serve as event handlers. The first subroutine executes when the `Start` event occurs and the second subroutine executes when the `End` event occurs.

C

The WSH Core Object Model

The WSH core object model provides programmatic access to Windows resources. There are 14 objects in the WSH core object model, as depicted in Figure C.1. Each of these objects provides access to a particular category of Windows resources.

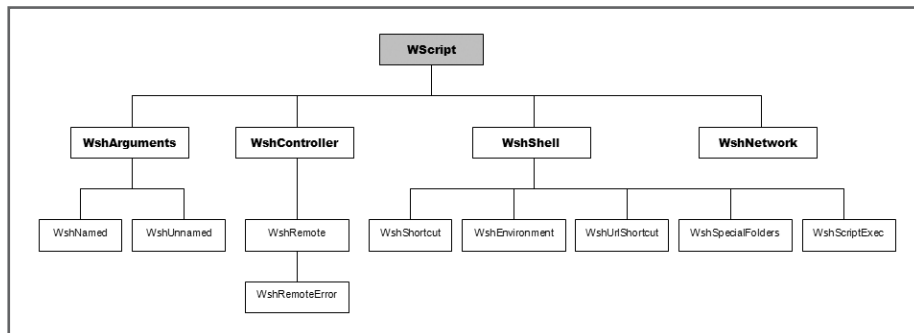


Figure C.1 The WSH core object model consists of 14 objects, all of which have properties and methods that expose various parts of the Windows operating system. © 2014 Cengage Learning.

At the top, or *root*, of the WSH core object model is the `WScript` object. All other objects are instantiated from this object. The `WScript` object is automatically established during the startup of the execution host and can therefore be referenced without first being instantiated within your scripts. The `WScript` object is referred to as a public or exposed object. The WSH core object model has three other public objects: `WshController`, `WshShell`, and `WshNetwork`. Each of these three objects must be instantiated within your scripts using the `WScript` object's `CreateObject()` method. All the other objects in the WSH core object model can only be instantiated by using properties or methods associated with the `WScript`, `WshController`, `WshShell`, and `WshNetwork` objects.

Table C.1 lists the rest of the objects in the WSH core object model, as well as the object properties or methods required to instantiate them.

TABLE C.1 WORKING WITH LOWER-LEVEL WSH OBJECTS

Object	Method of Instantiation
WshArguments	WScript.Arguments
WshNamed	WScript.Arguments.Named
WshUnnamed	WScript.Arguments.Unnamed
WshRemote	WshController.CreateScript()
WshRemoteError	WshRemote.Error
WshShortcut	WshShell.CreateShortcut()
WshUrlShortcut	WshShell.CreateShortcut()
WshEnvironment	WshShell.Environment
WshSpecialFolders	WshShell.SpecialFolders
WshScriptExec	WshShell.Exec()

© Jerry Lee Ford, Jr. All Rights Reserved.

WSH Objects and Their Properties and Methods

Each object in the WSH core object model provides access to, or exposes, a particular subset of Windows functionality. Table C.2 lists all 14 of the WSH core objects, provides a high-level description of these objects, and lists all the properties and methods associated with each object.

TABLE C.2 WSH CORE OBJECTS

Object	Description	Properties	Methods
WScript	This is the WSH root object. It provides access to a number of useful properties and methods. It also provides access to the rest of the objects in the WSH core object model.	Arguments, FullName, Interactive, Name, Path, ScriptFullName, ScriptName, StdErr, StdIn, StdOut, and Version	ConnectObject(), CreateObject(), DisconnectObject(), Echo(), GetObject(), Quit(), and Sleep()
WshArguments	This object enables you to access command-line arguments passed to the script at execution time.	Count, Item, Length, Named, Unnamed	Count() and ShowUsage()

TABLE C.2 WSH CORE OBJECTS (CONTINUED)

Object	Description	Properties	Methods
WshNamed	This object provides access to a set of named command-line arguments.	Item and Length	Count() and Exists()
WshUnnamed	This object provides access to a set of unnamed command-line arguments.	Item and Length	Count()
WshController	This object provides the capability to create a remote script process.	None	CreateScript()
WshRemote	This object provides the capability to administrator remote computer systems using scripts over a network.	Status and Error	Execute() and Terminate()
WshRemoteError	This object provides access to information on errors produced by remote scripts.	Description, Line, Character, SourceText, Source, and Number	None
WshNetwork	This object provides access to a number of different network resources such as network printers and drives.	ComputerName, UserDomain, and UserName	AddWindowsPrinterConnection(), AddPrinterConnection(), EnumNetworkDrives(), EnumPrinterConnection(), MapNetworkDrive(), RemoveNetworkDrive(), RemovePrinterConnection(), and SetDefaultPrinter()
WshShell	This object provides access to the Windows Registry, event log, environmental variables, shortcuts, and applications.	CurrentDirectory, Environment, and SpecialFolders	AppActivate(), CreateShortcut(), ExpandEnvironmentStrings(), LogEvent(), Popup(), RegDelete(), RegRead(), RegWrite(), Run(), SendKeys(), and Exec()
WshShortcut	This object provides scripts with methods and properties for creating and manipulating Windows shortcuts.	Arguments, Description, FullName, HotKey, IconLocation, TargetPath, WindowStyle, and WorkingDirectory	Save()

TABLE C.2 WSH CORE OBJECTS (CONTINUED)

Object	Description	Properties	Methods
WshUrlShortcut	This object provides scripts with methods and properties for creating and manipulating URL shortcuts.	FullName and TargetPath	Save()
WshEnvironment	This object provides access to Windows environmental variables.	Item and Length	Remove() and Count()
WshSpecialFolders	This object provides access to special Windows folders that allow scripts to configure the Start menu, desktop, Quick Launch Toolbar, and other special Windows folders.	Item	Count()
WshScriptExec	This object provides access to error information from scripts run using the Exec method.	Status, StdOut, StdIn, and StdErr	Terminate()

© Jerry Lee Ford, Jr. All Rights Reserved.

Examining Object Properties

By accessing object properties, your scripts can gather all kinds of information when they execute. For example, using the properties associated with the `WshNetwork` object, your scripts can collect information about the Windows domain that the person who ran the script has logged in to, as well as the computer's name and the user's name. This information could then be used, for example, to prevent the script from executing on certain domains or computers.

More than three dozen properties are associated with various WSH objects. In many cases, properties are associated with more than one object. Refer to Table C.2 to see which properties are associated with which objects. Table C.3 provides a complete review of WSH object properties.

TABLE C.3 WSH OBJECT PROPERTIES

Property	Description
Arguments	Sets a pointer reference to the <code>WshArguments</code> collection
AtEndOfLine	Returns either <code>true</code> or <code>false</code> depending on whether the end-of-line marker has been reached in the stream
AtEndOfStream	Returns either <code>true</code> or <code>false</code> depending on whether the end of the input stream has been reached
Character	Identifies the specific character in a line of code where an error occurs
Column	Returns the current column position in the input stream
ComputerName	Retrieves a computer's name
CurrentDirectory	Sets or retrieves a script's current working directory
Description	Retrieves the description for a specified shortcut
Environment	Sets a pointer reference to the <code>WshEnvironment</code>
Error	Provides the ability to expose a <code>WshRemoteError</code> object
ExitCode	Returns the existing code from a script started using <code>Exec()</code>
FullName	Retrieves a shortcut or executable program's path
HotKey	Retrieves the hotkey associated with the specified shortcut
IconLocation	Retrieves an icon's location
Interactive	Provides the ability to programmatically set script mode
Item	Retrieves the specified item from a collection or provides access to items stored in the <code>WshNamed</code> object
Length	Retrieves a count of enumerated items
Line	Returns the line number for the current line in the input stream or identifies the line number within a script on which an error occurred
Name	Returns a string representing the name of the <code>WScript</code> object
Number	Provides access to an error number
Path	Returns the location of the folder where the <code>CScript.exe</code> or <code>WScript.exe</code> execution hosts reside
ProcessID	Retrieves the process ID (PID) for a process started using the <code>WshScriptExec</code> object
ScriptFullName	Returns an executing script's path
ScriptName	Returns the name of the executing script
Source	Retrieves the identity of the object that caused a script error
SourceText	Retrieves the source code that created the error

TABLE C.3 WSH OBJECT PROPERTIES (CONTINUED)

Property	Description
SpecialFolders	Provides access to the Windows Start menu and desktop folders
Status	Provides status information about a remotely executing script or a script starting with Exec()
StdErr	Enables a script to write to the error output stream or provides access to read-only error output from an Exec object
StdIn	Enables read access to the input stream or provides access to the write-only input stream for the Exec object
StdOut	Enables write access to the output stream or provides access to the write-only output stream of the Exec object
TargetPath	Retrieves a shortcut's path to its associated object
UserDomain	Retrieves the domain name
UserName	Retrieves the currently logged-on user's name
Version	Retrieves the WSH version number
WindowStyle	Retrieves a shortcut's window style
WorkingDirectory	Returns the working directory associated with the specified shortcut

© Jerry Lee Ford, Jr. All Rights Reserved.

Working with Object Properties

Now let's take a look at an example of a VBScript that demonstrates how to instantiate an instance of the WshNetwork object and access its properties. The script is called NetInfo.vbs and is as follows:

```
Set WshNtwk = WScript.CreateObject("WScript.Network")

PropertyInfo = "User Domain" & vbTab & "= " & WshNtwk.UserDomain & _
vbCrLf & "Computer Name" & vbTab & "= " & WshNtwk.ComputerName & _
vbCrLf & "User Name" & vbTab & "= " & WshNtwk.UserName & vbCrLf

MsgBox PropertyInfo, vbOkOnly, "WshNtwk Properties Example"
```

As you can see, it isn't a very big script. It begins by using a Set statement to create an instance of the WshNetwork object, which is associated with a variable name of WshNtwk. After you have established an instance of the WshNetwork object in this manner, you can reference the object's properties and methods using its variable name assignment.

The next statement is very long. To improve the script's readability, I decided to break it into three lines and end each of the first two lines with the `&` and `_` characters. The `&` character is a concatenation character and is used to append two strings. The `_` character is a continuation character and is used to indicate that a statement is continued on the next line. This statement displays the values of the following `WshShell` properties:

- **WshNetwork.UserDomain.** The name of the domain that the person running the script is logged in to
- **WshNetwork.ComputerName.** The name of the computer on which the script is being executed
- **WshNetwork.UserName.** The username of the person who ran the script

To improve the presentation of the message, I formatted it using the VBScript `vbTab` and `vbCrLf` constants. The `vbTab` constant is used to line up the output at the point of the equals sign. The `vbCrLf` constant is used to execute a line feed and carriage return at the end of each line of output. The last thing the script does is display the message using the following statement:

```
MsgBox PropertyInfo, vbOkOnly, "WshNetwork Properties Example"
```

`MsgBox()` is a built-in VBScript function that displays a text message in a pop-up dialog box. `PropertyInfo` is a variable that I used to store the output message. `vbOkOnly` is a VBScript constant that tells the `MsgBox()` function to only display the OK button in the pop-up dialog box. The last part of the previous statement is a message that will be displayed in the pop-up dialog box's title bar. If you save and execute this script on a computer running Windows 7, you should see a pop-up dialog box similar to the one shown in Figure C.2.

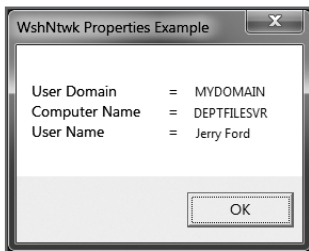


Figure C.2 A pop-up dialog box displaying properties associated with the `WshNetwork` object.

© 2014 Cengage Learning.

Examining Object Methods

The WSH also provides a large collection of object methods. By using these methods in your VBScripts, you'll be able to manipulate the Windows resources associated with objects. For example, using the `WshShell` object's `RegRead()`, `RegWrite()`, and `RegDelete()` methods, you can create scripts that can access and manipulate the contents of the Windows Registry. Using these methods, you can create scripts that can configure just about any Windows resource. Table C.4 provides a complete review of WSH object methods.

TABLE C.4 WSH OBJECT METHODS

Method	Description
AddPrinterConnection()	Creates printer mappings
AddWindowsPrinterConnection()	Creates a new printer connection
AppActivate()	Activates the targeted application window
Close()	Terminates or ends an open data stream
ConnectObject()	Establishes a connection to an object
Count	Retrieves the number of switches found in the WshNamed and WshUnnamed objects
CreateObject()	Creates a new instance of an object
CreateScript()	Instantiates a WshRemote object representing a script that is running remotely
CreateShortcut()	Creates a Windows shortcut
DisconnectObject()	Terminates a connection with an object
Echo()	Displays a text message
EnumNetworkDrives()	Enables access to network drives
EnumPrinterConnections()	Enables access to network printers
Exec()	Executes an application in a child command shell and provides access to the environment variables
Execute()	Initiates the execution of a remote script object
Exists()	Determines a specified key exists within the WshNamed object
ExpandEnvironmentStrings()	Retrieves a string representing the contents of the Process environmental variable
GetObject()	Retrieves an Automation object
GetResource()	Retrieves a resource's value as specified by the <resource> tag
LogEvent()	Writes a message in the Windows event log
MapNetworkDrive()	Creates a network drive mapping
Popup()	Displays a text message in a pop-up dialog box
Quit()	Terminates, or ends, a script
Read()	Retrieves a string of characters from the input stream
ReadAll()	Retrieves the string that is made up of the characters in the input stream
ReadLine()	Retrieves a string containing an entire line of data from the input stream

TABLE C.4 WSH OBJECT METHODS (CONTINUED)

Method	Description
RegDelete()	Deletes a Registry key or value
RegRead()	Retrieves a Registry key or value
RegWrite()	Creates a Registry key or value
Remove()	Deletes the specified environmental variable
RemoveNetworkDrive()	Deletes the connection to the specified network drive
RemovePrinterConnection()	Deletes the connection to the specified network printer
Run()	Starts a new process
Save()	Saves a shortcut
SendKeys()	Emulates keystrokes and sends typed data to a specified window
SetDefaultPrinter()	Establishes a default Windows printer
ShowUsage()	Retrieves information regarding the way a script is supposed to be executed
Skip()	Skips <i>x</i> number of characters when reading from the input stream
SkipLine()	Skips an entire line when reading from the input stream
Sleep()	Pauses script execution for <i>x</i> number of seconds
Terminate()	Stops a process started by Exec()
Write()	Places a string in the output stream
WriteBlankLines()	Places a blank in the output stream
WriteLine()	Places a string in the output stream

© Jerry Lee Ford, Jr. All Rights Reserved.

Working with Object Methods

To really understand how object methods work, it helps to have an example. In this example, let's work with the `WshShell` object. This object provides access to a number of Windows resources, including the following:

- The Windows application log
- The Windows Registry
- Any Windows command-line command

Let's look at an example of how to use the `WshShell` object's `LogEvent()` method to write a message to the Windows event log. The Windows event log is accessed differently depending on which version of Windows

you use. For example, on Windows 8.1 you can click the Start button, type View Event Logs, and press Enter, and then click View Event Logs to open the Event Viewer console. To view the application event log, drill down into the Windows Logs node and click on Application log. You can then click on any event entry in the application event log to examine it.

The scripting logic to write a message to the Windows application event log is very simple:

```
Set WshSh1 = WScript.CreateObject("WScript.Shell")
WshSh1.LogEvent 0, "EventLogger.vbs - Beginning script execution."
```

The first line of this script establishes an instance of the WshShell object. The second line uses the WshShell object's LogEvent() method to write a message to the event log.

Tip

One really good use of the WshShell object's LogEvent() method is to log the execution of scripts run using the Windows Event Scheduler service. This way, you can review the application event log each day and make sure that your scripts are executing when you expect them to.

Using your script editor, create a new script called EventLogger.vbs that contains the previous statements. Run the script and then check the application event log; you should find the message added by the script. Select it and you should see the Event Properties dialog box for the event, as shown in Figure C.3.

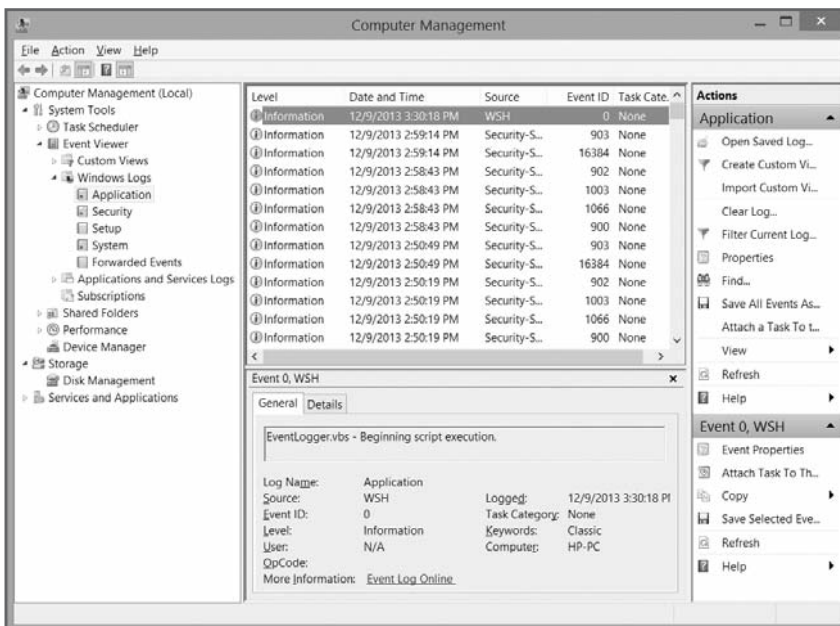


Figure C.3 Viewing the event that the EventLogger.vbs script added to the Windows application event log on a computer running Windows 8.1. © 2014 Microsoft Corporation. Used with permission from Microsoft.

D

Built-in VBScript Functions

VBScript provides an enormous collection of built-in functions, as outlined in Table D.1. You can use these functions in your VBScripts to shorten your development time and save yourself from having to reinvent the wheel.

TABLE D.1 BUILT-IN VBSCRIPT FUNCTIONS

Function Name	Description
Abs	Returns a number's absolute value
Array	Returns an array based on the supplied argument list
Asc	Returns the ANSI code of the first letter in the supplied argument
Atn	Inverse trigonometric function that returns the arctangent of the argument
CBool	Converts an expression to a Boolean value and returns the result
CByte	Converts an expression to a variant subtype of Byte and returns the result
CCur	Converts an expression to a variant subtype of Currency and returns the result
CDate	Converts an expression to a variant subtype of Date and returns the result
CDbl	Converts an expression to a variant subtype of Double and returns the result
Chr	Returns a character based on the supplied ANSI code
CInt	Converts an expression to a variant subtype of Integer and returns the result
CLng	Converts an expression to a variant subtype of Long and returns the result
Cos	Trigonometric function that returns the cosine of the argument
CreateObject	Creates an automation object and returns a reference to it
CSng	Converts an expression to a variant subtype of Single and returns the result
Date	Returns the current date
DateAdd	Adds an additional time interval to the current date and returns the result
DateDiff	Compares two dates and returns the number of intervals between them
DatePart	Returns a portion of the specified date
DateSerial	Returns a variant (subtype Date) based on the supplied year, month, and day
DateValue	Converts a string expression into a variant of type Date and returns the result
Day	Converts an expression representing a date into a number between 1 and 31 and returns the result
Eval	Returns the results of an evaluated expression
Exp	Returns the value of an argument raised to a power
Filter	Returns an array based on a filtered set of elements using supplied filter criteria
FormatCurrency	Returns an expression that has been formatted as a currency value
FormatDateTime	Returns an expression that has been formatted as a date or time value
FormatNumber	Returns an expression that has been formatted as a numeric value
FormatPercent	Returns an expression that has been formatted as a percentage (including the accompanying %)

TABLE D.1 BUILT-IN VBSCRIPT FUNCTIONS (CONTINUED)

Function Name	Description
GetLocale	Returns the locale ID
GetObject	Returns a reference for an automation object
GetRef	Returns a reference for a procedure
Hex	Returns a hexadecimal string that represents a number
Hour	Returns a whole number representing an hour in a day (0 to 23)
InputBox	Returns user input from a dialog box
InStr	Returns the starting location of the first occurrence of a substring within a string
InStrRev	Returns the ending location of the first occurrence of a substring within a string
Int	Returns the integer portion from the supplied number
isArray	Returns a value of True or False, depending on whether a variable is an array
IsDate	Returns a value of True or False, depending on whether an expression is properly formatted for a date conversion
IsEmpty	Returns a value of True or False, depending on whether a variable is initialized
IsNull	Returns a value of True or False, depending on whether an expression is set to Null
IsNumeric	Returns a value of True or False, depending on whether an expression evaluates to a number
isObject	Returns a value of True or False, depending on whether an expression has a valid reference for an automation object
Join	Returns a string that has been created by concatenating the contents of an array
Lbound	Returns the smallest possible subscript for the specified array dimension
Lcase	Returns a lowercase string
Left	Returns characters from the left side of a string
Len	Returns a number or string's character length
LoadPicture	Returns a picture object
Log	Returns the natural log of the specified argument
LTrim	Trims any leading blank spaces from a string and returns the result
Mid	Returns a number of characters from a string based on the supplied start and length arguments
Minute	Returns a number representing a minute within an hour in range of 0 to 59
Month	Returns a number representing a month within a year in the range of 1 to 12
MonthName	Returns a string containing the name of the specified month

TABLE D.1 BUILT-IN VBSCRIPT FUNCTIONS (CONTINUED)

Function Name	Description
MsgBox	Returns a value specifying the button users click in a dialog box
Now	Returns the current date and time
Oct	Returns a string containing an octal number representation
Replace	Returns a string after replacing occurrences of one substring with another substring
RGB	Returns a number that represents an RGB color
Right	Returns characters from the right side of a string
Rnd	Returns a randomly generated number
Round	Returns a number after rounding it by a specified number of decimal positions
RTrim	Trims any trailing blank spaces from a string and returns the result
ScriptEngine	Returns a string identifying the current scripting language
ScriptEngineBuildVersion	Returns the scripting engine's build number
ScriptEngineMajorVersion	Returns the scripting engine's major version number
ScriptEngineMinorVersion	Returns the scripting engine's minor version number
Second	Returns a number representing a second within a minute in range of 0 to 59
Sgn	Returns the sign of the specified argument
Sin	A trigonometric function that returns the sine of the argument
Space	Returns a string consisting of a number of blank spaces
Split	Organizes a string into an array
Sqr	Returns a number's square root
StrComp	Returns a value that specifies the results of a string comparison
String	Returns a character string made up of a repeated sequence of characters
Tan	A trigonometric function that returns the tangent of the argument
Time	Returns a variant of subtype Date that has been set equal to the system's current time
Timer	Returns a value representing the number of seconds that have passed since midnight
TimeSerial	Returns a variant of subtype Date that has been set equal to the specified hour, minute, and second

TABLE D.1 BUILT-IN VBSCRIPT FUNCTIONS (CONTINUED)

Function Name	Description
TimeValue	Returns a variant of subtype Date that has been set using the specified time
Trim	Returns a string after removing any leading or trailing spaces
TypeName	Returns a string that specifies the variant subtype information regarding the specified variable
Ubound	Returns the largest subscript for the specified array dimension
Ucase	Returns an uppercase string
VarType	Returns a string that specifies the variant subtype information regarding the specified variable
Weekday	Returns a whole number in the form of 1 to 7, which represents a given day in a week
WeekdayName	Returns a string identifying a particular day in a week
Year	Returns a number specifying the year

This page intentionally left blank

E

What's on the Companion Website?

The best way to become a good script developer is to spend time writing new scripts. However, it helps to have a collection of scripts from which you can cut and paste when starting out. Hopefully, you've been creating the scripts that you've seen in this book as you've gone along. But just in case you missed some, I've added copies of each script to the book's companion website. In this appendix, I'll provide a brief reference to each of the scripts. The companion website can be found at www.cengageptr.com/downloads. Enter the name of the book or ISBN to find the files.

Script Examples

Table E.1 provides a quick overview of all the sample scripts from this book that are located on the companion website.

TABLE E.1 SAMPLE SCRIPTS ON THE COMPANION WEBSITE

Reference	Script	Description
Chapter 1	Hello-1.vbs	Displays the classic “Hello World!” message
	Hello-2.vbs	Displays a message using the WshShell object’s Popup() method
	Hello-3.vbs	Displays a message using the WScript object’s Echo() method
	KnockKnock.vbs	A “knock knock” joke game
Chapter 2	RockPaperScissors.vbs	A “rock, paper, and scissors” game
Chapter 3	FreeSpace.vbs	Demonstrates how to determine how much free space is left on a disk drive
	MathGame.vbs	Prompts the user to solve a mathematical equation and demonstrates how to solve it in the event that the user cannot do so
	SquareRootCalc - 1.vbs	Demonstrates how to solve square-root calculations using a mathematic solution devised by Sir Isaac Newton
	SquareRootCalc - 2.vbs	Demonstrates how to solve square-root calculations using VBScript’s built-in Sqr() function
Chapter 4	ArgumentProcessor.vbs	Demonstrates how to work with arguments passed to the script by the user at execution time
	ArrayDemo.vbs	Demonstrates how to store and retrieve data using a single-dimension VBScript array
	BigBadWolf.vbs	Demonstrates how to use the Option Explicit statement
	CaptainAvenger.vbs	Prompts the user to answer a number of questions and then uses the answers to create a comical action-adventure story
	ComputerAnalyzer.vbs	Demonstrates how to access environment variables using the WSH
	HappyHour.vbs	Tells the user whether it’s Friday
	LittlePigs.vbs	Demonstrates how to use a constant to create a standardized title-bar message for pop-up dialog boxes displayed by the script
	MathDemo.vbs	Demonstrates how to use various VBScript arithmetic operators
	MsgFormatter.vbs	Demonstrates how to use VBScript string constants to control how text messages are displayed
	ResizeArray.vbs	Demonstrates how to resize an array during execution

**TABLE E.1 SAMPLE SCRIPTS ON
THE COMPANION WEBSITE (CONTINUED)**

Reference	Script	Description
Chapter 5	RockPaperScissors - 2.vbs	Revisits the RockPaperScissors.vbs script first introduced in Chapter 2 and updates it using advanced conditional logic
	RockPaperScissors - 3.vbs	Revisits the RockPaperScissor - 2.vbs script, replacing some of the If statement logic with a Select Case statement
	PlanetTrivia.vbs	Creates a "planet trivia" quiz game
Chapter 6	GuessANumber.vbs	Plays a number-guessing game with the user
	ShortcutMaker.vbs	Creates shortcuts on the Windows desktop, Programs menu, and Quick Launch Toolbar for the GuessANumber.vbs VBScript
Chapter 7	BlackJack.vbs	Creates a scaled-down version of casino blackjack
	GuessANumber - 2.vbs	Plays a number-guessing game with the user
Chapter 8	LuckyLotteryMachine.vbs	Assists players by automating the random generation of lottery numbers
	ExtractFileProperties.vbs	Demonstrates how to access any file's properties
	FileCreate.vbs	Demonstrates how to create and write to a new text file
	INIDemo.vbs	Demonstrates how to read and process the content of an INI file
Chapter 9	Hangman.vbs	Demonstrates how to create a game of hangman using VBScript and the WSH
Chapter 10	Hangman - 2.vbs	Completes the Chapter 9 hangman game by configuring it to store and retrieve game settings using the Windows Registry
	HangmanSetup.vbs	Loads configuration settings for the hangman game into the Windows Registry
	ProcessorInfo.vbs	Demonstrates how to retrieve information about the computer's processor.
Chapter 11	NewObjectDemo.vbs	Demonstrates how to create customized objects
	TicTacToe.vbs	Creates a two-player tic-tac-toe game
Chapter 12	VBScriptGameConsole.wsf	Creates a game console that builds a dynamic list of VBScript games for the player to select from

**TABLE E.1 SAMPLE SCRIPTS ON
THE COMPANION WEBSITE (CONTINUED)**

Reference	Script	Description
Chapter 13	GetBIOSInfo.vbs	Demonstrates how to retrieve BIOS information using WMI
	GetEventLogInfo.vbs	Demonstrates how to retrieve event-log data using WMI
	GetMotherboardInfo.vbs	Demonstrates how to retrieve motherboard information using WMI
	GetOSInfo.vbs	Demonstrates how to retrieve operating-system information using WMI
	GetServicesInfo.vbs	Demonstrates how to retrieve information about services using WMI
	WMIServiceCycler.vbs	Demonstrates how to stop and start Windows services using WMI
Chapter 14	HelloWorld.html	A basic HTML file
	HelloWorld.hta	A simple one-line HTA
	htaHelloWorld.hta	A basic HTA
	WindowWithoutTitleBar.hta	An HTA that displays a windows without a title bar
	TextBox.hta	An HTA that demonstrates how to create and work with a text box control
	PasswordBox.hta	An HTA that demonstrates how to create and work with a password box control
	Checkbox.hta	An HTA that demonstrates how to create and work with a checkbox control
	RadioButtons.hta	An HTA that demonstrates how to create and work with radio button controls
	Button.hta	An HTA that demonstrates how to create and work with a button control created with the <input> tag
	Button-2.hta	An HTA that demonstrates how to create and work with a button control created with the <button> tag
	Multi-lineTextBox.hta	An HTA that demonstrates how to create and work with a multi-line text box control
	Multi-lineTextBox-2.hta	An HTA that demonstrates how to create and work with a checkbox control that contains pre-populated text
	ListBox.hta	An HTA that demonstrates how to create and work with a list box control

**TABLE E.1 SAMPLE SCRIPTS ON
THE COMPANION WEBSITE (CONTINUED)**

Reference	Script	Description
Chapter 14	Multi-ListBox.hta	An HTA that demonstrates how to create and work with a multiple selection list box control
	Drop-downListBox.hta	An HTA that demonstrates how to create and work with a drop-down list box control
	StartNotepad.hta	An HTA that demonstrates how to start a Windows application using the WScript.Shell object's Run method
	StartNotepad-2.hta	An HTA that demonstrates how to start a Windows application using the WshShell object's ShellExecute method
	AutoRefreshProcessList.hta	An HTA that demonstrates how to use WMI to retrieve and continuously monitor a list of Windows processes
	RockPaperScissors.hta	An HTA implementation of the Rock, Paper, Scissors game
	HTATempFile.hta	A template used as the basis for creating new HTAs
Appendix A	ScreenSaver.vbs	Changes the user's screensaver settings
	BackGround.vbs	Changes the user's background selection to None and sets the default background color to white
	DriveMapper.vbs	Demonstrates how to add logic to VBScripts to set up a network drive mapping
	DriveBuster.vbs	Demonstrates how to add logic to VBScripts to terminate a network drive mapping
	PrinterMapper.vbs	Demonstrates how to use a VBScript to set up a connection to a network printer
	PrinterBuster.vbs	Demonstrates how to use a VBScript to disconnect a network printer connection
	ServiceCycler.vbs	Demonstrates how to use a VBScript to stop and start Windows services
	AccountCreator.vbs	Demonstrates how to use a VBScript to create new user accounts
	VBSCleanup.vbs	Demonstrates how to use a VBScript to automate the execution of the Windows Disk Cleanup utility
	WordObjectModelExample.vbs	Demonstrates how to use a VBScript to automate the creation of a new Word document
	WinZipDemo.vbs	Demonstrates how to use a VBScript to automate the creation of a new ZIP file

TABLE E.1 SAMPLE SCRIPTS ON
THE COMPANION WEBSITE (CONTINUED)

Reference	Script	Description
Appendix B	WSHRemoteDemo.vbs	Demonstrates how to remotely execute a VBScript on a network computer
Appendix C	EventLogger.vbs	Demonstrates how to write messages to the Windows application event log
	NetInfo.vbs	Demonstrates how to collect network information

Index

SYMBOLS

- _ (underscore), 95
- %systemroot% environment variable, 248
- & character, 89
- * (ampersand), 200
- = (equals sign), 96
- ? (question mark), 200

A

- AboutFunction() function, 314
- accounts, managing users, 391–392
- adding
 - blank lines, 195–196
 - comments, 57–58
 - controls
 - buttons, 358–359, 360
 - checkboxes, 355–357
 - multi-line text, 360–361
 - passwords, 355
 - radio buttons, 357–358
 - text, 353–355
 - dictionary items, 113–114
 - drop-down lists, 365
 - elements to interfaces, 352–365
 - GUIs to scripts, 339–375
 - Run as Administrator menu option, 13
 - scripts
 - Apps group, 162
 - Programs menu, 161–162
 - style rules, 351–352
- administrative scripting, 379–404
 - computers, 389–392
 - desktop, 380–383
 - disk management, 392–394
 - networks, 383–386
 - printers, 386–388
- ampersand (*), 200
- answers
 - Math game
 - checking for correct, 78–79
 - collecting player's, 78
 - Planet Trivia game, 139–141

applications

- event logs, recording, 229
- HTAs, 3, 339
 - adding style rules, 351–352
 - overview of, 340–341
- HTML, 399–404
- starting, 366–367
- third-party, automating execution, 397–399
- VBScript, integrating, 394–399
- Windows, 33
- WMI, 317. *See also* WMI
- Apps group, adding scripts, 162
- architecture, Remote WSH, 406–409
- arguments
 - definitions, 114
 - definitions of, 39
 - input, accepting, 115–116
 - scripts, passing, 114–115
- arithmetic operators, 97, 99
- ArrayName, 108
- arrays, 83
 - contents, processing, 105–107
 - definitions, 103
 - dynamic, building, 111–112
 - erasing, 112
 - multiple-dimension, 105
 - resizing, 109–110
 - single-dimension, 103–104
 - sizing, 108
- ASP (Active Server Pages), 21
- attributes
 - <HTA:APPLICATION> tag, 344–347
 - <input> tag, 353
- automating
 - third-party applications, executing, 397–399
 - Windows shutdown, 402–404
 - Word (Microsoft) reports, 394–397

B

- backgrounds, configuring desktop, 380–381
- BIOS, retrieving information, 331–332
- BlackJack Lite game, 165–166
 - ComputerPlay() function, 182
 - DealAnotherCard() function, 182
 - DealFirstHand() function, 180
 - design, 176–177
 - DetermineWinner() function, 182–183
 - DisplaySplashScreen() function, 183
 - DoYouWantToPlay() function, 179
 - GetRandomNumber() function, 181
 - initialization sections, 177–178
 - main processing sections, 178
 - NowGoPlay() function, 180
 - PlayTheGame() function, 180–181
 - results, 184
- blank lines, adding, 195–196
- blocks, declaration, 350
- boxes, lists, 361
- BuildDisplayString() function, 215
- built-in functions, 66–68, 174, 421–425
- built-in objects, 265–293
 - accessing, 267–269
 - customizing, 269–274
 - methods, 268–269
 - properties, 268
 - VBScript, 59
- buttons, adding controls, 358–359, 360
- <button> tag, 360

- C**
- C++, 4
 - Calculator**, 51–52, 80
 - Call statement**, 53
 - characters**
 - wildcard, 200
 - writing, 194–195
 - checkboxes, adding controls**, 355–357
 - CheckIfGameWon() function**, 240
 - CIM (Common Information Model)**, 318, 320, 321–322, 336
 - CIMOM (CIM object manager)**, 320–321
 - classes, WMI**, 326
 - Class_Initialize event**, 274
 - Class object**, 268
 - Class statement**, 53
 - ClearGameBoard() function**, 283
 - Clear() method**, 226, 268, 275
 - closing**
 - files, 191–194
 - Windows Console, 14
 - code**
 - pseudo, 125
 - reusable, 170, 303
 - collections**, 103–112, 143–164
 - built-in objects, 268
 - Do...Until statements, 145, 152–153
 - Do...While statements, 144, 149–151
 - For Each...Next statements, 144, 147–149
 - For...Next statements, 144, 145–147
 - Matches, 279–280
 - scripts, adding looping logic to, 144–154
 - While...Wend statements, 145, 153–154
 - CollectPlayerInput() function**, 212–213
 - command-line execution**
 - configuring, overriding settings, 38–39
 - CScript.exe/WScript.exe, configuring, 36–37
 - command prompts**
 - accessing, 10
 - navigating, 14
 - scripts, executing, 19–20
 - commands, entering**, 14
 - comments, adding**, 57–58
 - <comment> tag**, 299–300
 - common website, navigating**, 427–432
 - comparison operators**, 135–136
 - compatibility**
 - operating systems, 15–16
 - third-party script engines, 21
 - components**, 33–35
 - WMI infrastructure, 319
 - WSH, 5
 - ComputerPlay() function**, 182
 - computers, administrative scripting**, 389–392
 - concatenating strings**, 89
 - conditional logic**, 123–142
 - conditions, testing**, 126
 - configuration settings, storing in external files**, 204–207
 - configuring**
 - BlackJack Lite game
 - initialization sections, 177–178
 - main processing sections, 178
 - command-line execution, 36–37
 - CScript.exe, 36–37
 - desktops
 - backgrounds, 380–381
 - screensavers, 381–383
 - execution hosts, 35–43
 - file properties, 41
 - multiple users, 36
 - Rock, Paper, and Scissors game random selection, 45
 - Run as Administrator menu option, 12–13
 - scripts, Registry (Windows), 245–262
 - settings, overriding, 38–39
 - WScript.exe
 - command-line execution, 36–37
 - customizing, 39–41
 - desktop execution, 37–38
 - ConsoleLoop() function**, 309–311
 - consoles (game)**, 295–296, 304–315
 - AboutFunction() function, 314
 - ConsoleLoop() function, 309–311
 - design, 304
 - development, 307–314
 - HelpFunction() function, 314
 - initialization sections, 307
 - InvalidChoice() function, 313–314
 - JScript, writing, 305–306, 315
 - main processing sections, 308–309
 - PickAGame() function, 313
 - results, 315
 - RunScript() function, 313
 - ValidateAndRun() function, 311–312
 - XML, using to outline script structure, 304–305
 - constants**, 83
 - defining, 86
 - definitions, 85
 - OpenTextFile() method, 191
 - Planet Trivia game, 137–138
 - run-time, 88–91
 - scripts, 85, 86–87
 - Story of Captain Adventure, The, 118
 - strings, 89–91
 - values, referencing, 87
 - Const statement**, 53, 86
 - consumers (WMI)**, 319, 320
 - contents, processing arrays**, 105–107
 - controls**
 - buttons, adding, 358–359, 360
 - checkboxes, adding, 355–357
 - interfaces, formatting, 352–353
 - lists, applying, 361–365
 - multi-line text, adding, 360–361
 - passwords, adding, 355
 - radio buttons, adding, 357–358
 - text, adding, 353–355
 - converting**
 - data, 92
 - functions, 92
 - copying**
 - files, 200
 - folders, 202–203
 - multiple files, 201
 - core object models**, 5, 6–7, 34, 411–420
 - CreateObject() method**, 35, 44, 102
 - creating. See formatting**
 - cross-name space queries**, 328
 - CScript.exe**, 6
 - command-line execution, configuring, 36–37
 - scripts, executing, 20
 - CSS (Cascading Style Sheets)**, 339
 - rules, 350
 - selectors, 351
 - customizing**
 - built-in objects, 269–274
 - functions, 168–169
 - <HTA:APPLICATION> tag, 344–347
 - log files, 228
 - WScript.exe, 39–41

- D**
- data**
- collections, processing, 143–164
 - converting, 92
 - definitions, 85
 - operating systems, retrieving, 324–326
 - program, viewing, 124–135
 - receiving, 187–216
 - scripts, viewing, 84–85
 - storage, 187–216
 - applying WSH, 189–191
 - dictionaries, 113–114
 - modifying during script execution, 91–103
 - Registry (Windows), 249–250
 - types
 - Registry (Windows), 249
 - VBScript, 91–92
 - VBScript statements, defining, 85
- databases, 247. See also Registry (Windows)**
- Date() function, 89**
- dates, constants, 88–91**
- DealAnotherCard() function, 182**
- DealFirstHand() function, 180**
- debugging, enabling/disabling, 298**
- declarations**
- blocks, 350
 - statements, 350
- default command-line execution hosts, 37**
- definitions**
- arguments, 39, 114
 - arrays, 103
 - command prompts, 9
 - constants, 85
 - data, 85
 - declaration blocks, 350
 - dynamic arrays, 111
 - endless loops, 151
 - errors
 - logical, 221
 - run-time, 221
 - syntax, 221
 - event handlers, 225
 - HTAs, 340–341
 - IDEs, 24
 - instantiation, 34
 - keys (Registry), 249
 - loops, 144
 - methods, 7
 - .NET, 24
 - object models, 33
 - procedures, 86, 167
 - properties, 7, 350
 - pseudo code, 125
 - Registry (Windows), 247
 - selectors, 350
 - shortcuts, 158
 - special folders, 159
 - strings, 92
 - text strings, 92
 - values, 249
 - variables, 44, 85
 - variants, 91
 - Windows Resource Kits, 8
 - Windows Script Files, 297
 - XML, 297
- deleting**
- dictionary items, 114
 - files, 200
 - folders, 203
 - multiple files, 202
- demos**
- improved square-root calculator, 67–68
 - Remote WSH, 409–410
 - square-root calculators, 66–67
- Description property, 226, 269**
- design**
- BlackJack Lite game, 176–177
 - consoles (game), 304
 - Guess a Number game, 154–155
 - Hangman game, 230–231
 - Knock Knock game, 25–26
 - Lucky Lottery Number Picker game, 208
 - Math game, 77–80
 - Rock, Paper, and Scissors game, 43–44
 - scripts, accepting argument input, 115–116
 - Story of Captain Adventure, The, 117
 - Tic Tac Toe game, 281
- desktops**
- administrative scripting, 380–383
 - backgrounds, configuring, 380–381
 - execution, configuring WScript.exe, 37–38
 - screensavers, configuring, 381–383
 - shortcuts, 158–161
- DetermineIfSetIsComplete() function, 214–215**
- DetermineWinner() function, 182–183**
- development**
- consoles (game), 307–314
 - flowcharts, 127
 - Knock Knock game, 26
 - Microsoft Developer Network, 337
 - Planet Trivia game, 137
 - Rock, Paper, and Scissors game, 371–375
 - scripts, 19
 - WMI scripts, 323–324
- dialog boxes**
- Environment Variables, 100
 - Properties, 40
 - User Account, 11
 - Windows Script Host Settings, 38
- dictionaries, 83, 113–114**
- dimensions, 104**
- Dim statement, 53, 93**
- directories, 14**
- DisconnectObject() method, 35**
- disk management, 392–394**
- DisplayBoard() function, 286**
- DisplayData() function, 325**
- DisplayFinalResults() function, 215–216**
- DisplayGameResults() function, 241–242, 286–287**
- DisplaySplashScreen() function, 183, 216, 293**
- diverting keystrokes, 74**
- DLL (dynamic link library), 32. See also execution hosts**
- DMTF (Distributed Management Task Force), 318**
- documentation, 58, 170**
- Do...Loop statement, 53**
- DOM (Document Object Model), 33**
- Do...Until statement, 145, 152–153**
- Do...While statement, 144, 149–151**
- downloading WSH, 16**
- DoYouWantToPlay() function, 179, 233**
- drives, networks**
- disconnecting, 385–386
 - mapping, 383–385
- drop-down lists, adding, 365**
- dynamic arrays, building, 111–112**
- dynamic link library. See DLL**

EEcho() **method**, 34, 35, 69**elevated mode, accessing Windows Console**, 10–13ElseIf **keywords**, 127End If **keyword**, 126**endless loops**, 151**engines**

- scripts, 5
- selecting, 6

environments

- scripts, navigating, 32–33
- variables, 100–103

Environment Variables dialog box, 100**equals sign (=)**, 96Erase **statement**, 53**erasing arrays**, 112Err **object**, 268, 274–275**errors**

- definitions
 - logical, 221
 - run-time, 221
 - syntax, 221

handlers, creating, 225–227

logical, preventing, 222–223

Remote WSH properties, 409

reporting, enabling/disabling, 298

scripts, 219–242

- messages, 221–222
- overview of, 221–223
- reporting, 227–229
- troubleshooting, 223–227

testing, 78

events

Class_Initialize, 274

handlers, 225

logs, 227–229

procedures, formatting, 271–274

records, retrieving, 329–331

Remote WSH, 407–408

examples

- HTAs, 370
- INI files, 205–207
- scripts, 427–432

ExecuteGlobal **statement**, 54Execute() **method**, 268Execute **statement**, 53**executing**

- command-lines, configuring, 36–37
- For . . . Next statements, 144, 145–147

HTAs, 341, 342–343

Remote WSH, 409

scripts, 15

- command prompt, 19–20

- modifying data during, 91–103
- statements, 18

- third-party applications, automating, 397–399

Windows Script Files, 303–304

WMI queries, 327–329

execution hosts

- configuring, 35–43
- default command-line, 37
- scripts, 5

Exit Function **statement**, 227Exit **statement**, 54Exit Sub **statement**, 227ExpandEnvironmentStrings() **method**, 102**exponentiation**, 99**expressions**

- regular, applying, 275–280
- variables, modifying, 96–100

external files, storing configuration settings, 204–207**F**FileExists() **method**, 191**files**, 7. *See also* **objects**

- closing, 191–194
- copying, 200
- deleting, 200
- external, storing configuration settings, 204–207
- formatting, 40
- INI, 205
 - examples, 205–207
 - Lucky Lottery Number Picker game, 208
- logs, customizing, 228
- managing, 199–203
- moving, 200
- multiple
 - copying, 201
 - deleting, 202
 - moving, 201
- opening, 191–194
- properties, configuring, 41
- reading from, 196–199
- text, viewing, 197

Windows Script Files

- definitions, 297
- executing, 303–304
- writing to, 194–196

FileSystemObject **object**, 59, 65, 66, 190, 191FillArray() **function**, 233–234**finishing user input**, 28FirstIndex **property**, 269FirstLevelValidation() **function**, 238FlipString() **function**, 240–241**flowcharts**, 127**folders**

- copying, 202–203
- deleting, 203
- formatting, 202
- managing, 199–203
- moving, 203
- special
 - applying, 161
 - definitions, 159

For Each . . . Next **loop**, 107For Each . . . Next **statement**, 54, 144, 147–149FormatNumber() **function**, 155**formatting**

- comments, 57
- errors, handlers, 225–227
- events, procedures, 271–274
- files, 40
- flowcharts, 127
- folders, 202
- HTAs, 342–343, 343–352
- interfaces, controls, 352–353
- keys, Registry (Windows), 251–254
- loops, 155
- Matches collections, 279–280
- scripts, 255–256, 302–303
- shortcuts (Guess a Number game), 158–164
- splash screens (Planet Trivia game), 138
- splash screens (The Story of Captain Adventure), 118–119
- tags, 297
- values, Registry (Windows), 251–254
- WScript objects, 34

For . . . Next **statement**, 54, 144, 145–147ForReading **mode**, 193**frameworks**, WMI. *See* **WMI****functionality**, VBScript, 17

functions

AboutFunction(), 314
 BuildDisplayString(), 215
 built-in, 421–425
 CheckIfGameWon(), 240
 ClearGameBoard(), 283
 CollectPlayerInput(), 212–213
 ComputerPlay(), 182
 ConsoleLoop(), 309–311
 converting, 92
 customizing, 168–169
 Date(), 89
 DealAnotherCard(), 182
 DealFirstHand(), 180
 definition of, 18
 DetermineIfSetIsComplete(), 214–215
 DetermineWinner(), 182–183
 DisplayBoard(), 286
 DisplayData(), 325
 DisplayFinalResults(), 215–216
 DisplayGameResults(), 241–242, 286–287
 DisplaySplashScreen(), 183, 216, 293
 DoYouWantToPlay(), 179, 233
 FillArray(), 233–234
 FirstLevelValidation(), 238
 FlipString(), 240–241
 FormatNumber(), 155
 GetRandomNumber(), 181, 213
 GetWordFileLocation(), 260–261
 HelpFunction(), 314
 InitialDisplayString(), 237
 InputBox(), 109
 Int(), 155
 InvalidChoice(), 313–314
 ManageGamePlay(), 284–286
 MarkPlayerSelection(), 290–291
 MsgBox(), 18, 55
 NonExistentFunction(), 226
 NonGuessedString(), 239–240
 NowGoPlay(), 180
 PickAGame(), 313
 PlayTheGame(), 180–181, 234–236
 ProcessRandomNumber(), 213–214
 ProcessScriptIniFile(), 210–212
 RandomNumber(), 174
 ResetVariableDefaults(), 215
 RetrieveWord(), 236–237, 259–260
 Rnd(), 155
 RunScript(), 313
 SecondLevelValidation(), 238–239
 SeeIfWon(), 291–293
 SelectAWordCategory(), 261–262
 SetVariableDefaults(), 210, 283
 SplashScreen(), 242
 Sqr(), 66
 StartService(), 336
 StopService(), 336
 TestLevelGuess(), 239
 UCase(), 112
 ValidateAndRun(), 311–312
 ValidateInput(), 287–290
 VBScript, 66–68, 174
 Weekday(), 89

Function **statement**, 54

G**games. See also specific games**

BlackJack Lite, 165–166
 consoles, 295–296, 304–315. *See also* consoles (game)
 Guess a Number, 143–144, 154–164
 Hangman, 219–220, 230–242, 245–247, 254–262
 Knock Knock, 3–4, 25–29
 Lucky Lottery Number Picker, 187–188, 207–216
 Math, 51–53, 74–81
 Planet Trivia, 123–124, 136–142
 Rock, Paper, and Scissors, 31–32, 43–47, 129–133, 339–340, 370–375
 Story of Captain Adventure, The, 83–84, 117–120
 Tic Tac Toe, 265–267

GetDrive() **method**, 65
 GetRandomNumber() **function**, 181, 213
 GetWordFileLocation() **function**, 260–261

Global **property**, 269

groups, adding scripts to Apps, 162

Guess a Number game, 143–144
 design, 154–155
 loops, creating to control games, 155
 player input
 testing, 156–157
 verifying, 157
 procedures, 170–174
 random numbers, generating, 155

results, 157

shortcuts, creating, 158–164

starting, 155

GUIs (graphical user interfaces), 3, 4

ping scripts, 399–402

scripts

 adding, 339–375

 comparing HTAs to HTML pages, 341–342

 elements, 352–365

 executing HTAs, 342–343

 formatting HTAs, 343–352

 navigating HTAs, 340–341

H**handlers**

errors, creating, 225–227

events, definitions, 225

Hangman game, 219–220, 230–242,

245–247, 254–262

 CheckIfGameWon() function, 240

 design, 230–231

 DisplayGameResults() function, 241–242

 DoYouWantToPlay() function, 233

 FillArray() function, 233–234

 FirstLevelValidation() function, 238

 FlipString() function, 240–241

 InitialDisplayString() function, 237

 initialization sections, 231–232, 257–258

 main processing sections, 232–233, 255–256

 NonGuessedString() function, 239–240

 PlayTheGame() function, 234–236
 results, 242

 RetrieveWord() function, 236–237
 scripts

 creating setup, 255–256

 viewing, 262

 SecondLevelValidation() function, 238–239

 SplashScreen() function, 242

 templates, 231–232

 TestLevelGuess() function, 239

 updating, 257–262

Hello World, creating, 17–19

HelpContext **property**, 269

HelpFile **property**, 269

HelpFunction() **function**, 314

history of VBScript, 22–23

hosts

definition of, 6

execution. *See* execution hosts

<HTA:APPLICATION> **tag**, 343–346

HTAs (HTML Applications), 3, 339

definitions, 340–341

formatting, 342–343, 343–352

HTML, comparing to, 341–342

interfaces, adding elements to, 352–365

overview of, 340–341

style rules, adding, 351–352

WSH, integrating, 366–370

HTML (Hypertext Markup Language)

applications, 399–404

HTAs, comparing to, 341–342

<html> **tag**, 342

Hungarian Notation, 104

I

If **statement**, 124, 125–133

If...Then...Else **statement**, 54

IgnoreCase **property**, 269

IIS (Internet Information Service), 21

improved square-root calculator demo, 67–68

indexes, dynamic arrays, 111

indicators, event log errors, 229

infrastructure, WMI, 319–323

INI files, 205

examples, 205–207

Lucky Lottery Number Picker game, 208

InitialDisplayString() **function**, 237

initialization sections

BlackJack Lite game, 177–178

comment templates, 58

consoles (game), 307

Hangman game, 231–232, 257–258

Lucky Lottery Number Picker game, 208–209

Tic Tac Toe game, 281–282

input

arguments, accepting, 115–116

collecting (The Story of Captain Adventure), 119

testing, 156–157

verifying, 157

InputBox() **method**, 45, 69, 71–72, 109

<input> **tag**, 352–353

installing WSH, 16–17

InstancesOf **method**, 325

instantiation, definition of, 34

integrating

VBScript into other applications, 394–399

WSH into HTAs, 366–370

interfaces, 3

controls, formatting, 352–353

elements, adding, 352–365

scripts, adding, 339–375

Internet Explorer

scripting environments, 33

VBScript, 21

interpreters, accessing built-in objects, 267

Int() **function**, 155

InvalidChoice() **function**, 313–314

iteration, processing arrays, 107

iterative statements, 144

J

<job> **tag**, 300–301

<?job ?> **tag**, 298

JScript, 5, 305–306, 315

K

keys

definitions, 249

deleting, 252

dictionaries, 113

Registry (Windows)

accessing, 250–251

formatting, 251–254

root (Registry), 248

keystrokes

diverting, 74

SendKeys() **method**, 75–76

keywords

ElseIf, 127

End If, 126

If, 126

Preserve, 110

Then, 126

WQL, 328

Knock Knock game, 3–4, 25–29

L

languages, 4

learning new, 22

programming, 23

scripts

combining, 295–315

support, 297

support, 20–21

WQL, 327–329

last responses, validating, 28

Lbound() **method**, 108

learning new languages, 22

Length **property**, 269

libraries, WMI scripting, 320

limiting scope, variables, 174–175

lines

blank, adding, 195–196

skipping, 198

writing, 195

lists

boxes, 361

controls, applying, 361–365

drop-down, adding, 365

logic

conditional, 123–142

Hangman game, 232–233

main processing sections (BlackJack Lite game), 178

scripts, adding looping, 144–154

Tic Tac Toe game, 282–283

logical errors

definitions, 221

preventing, 222–223

logs

events, 227–229

files, customizing, 228

records, retrieving, 329–331

loops. *See also* statements

creating, 155

definitions, 144

endless, 151

For Each...Next, 107

logic, adding to scripts, 144–154

lowercase, formatting tags, 297

Lucky Lottery Number Picker game, 187–188, 207

BuildDisplayString() **function**, 215

CollectPlayerInput() **function**, 212–213

design, 208

DetermineIfSetIsComplete() function, 214–215
 DisplayFinalResults() function, 215–216
 DisplaySplashScreen() function, 216
 GetRandomNumber() function, 213
 initialization sections, 208–209
 main processing sections, 209–210
 ProcessRandomNumber() function, 213–214
 ProcessScriptIniFile() function, 210–212
 ResetVariableDefaults() function, 215
 results, 216
 SetVariableDefaults() function, 210

M

main processing sections

BlackJack Lite game, 178
 comment templates, 58
 consoles (game), 308–309
 Hangman game, 232–233, 255–256
 Lucky Lottery Number Picker game, 209–210
 Tic Tac Toe game, 282–283

managed resources (WMI), 319, 323, 334–336

ManageGamePlay() function, 284–286

managing

disk management, 392–394
 files, 199–203
 Registry (Windows), 248
 scripts, 165–184
 limiting scope with variables, 174–175
 optimizing, 169–170
 services, 389–390
 user accounts, 391–392
 WMI, 317–336

mapping network drives, 383–385

MarkPlayerSelection() function, 290–291

Matches collections, creating, 279–280

Matches object, 268

matching patterns

replacing, 276–278
 testing, 279

Match object, 268

Math game, 51–53, 74–81

answers

checking for correct, 78–79
 collecting player's, 78
 Calculator, 80
 design, 77–80
 results, 81
 SendKeys method, 74–77
 starting, 77–78
 WordPad, 79–80

menus

Programs, adding scripts, 161–162
 Run as Administrator, configuring, 12–13

messages

errors, 221–222
 recording, 229

methods

built-in objects, 268–269
 Clear(), 226, 268, 275
 CreateObject(), 35, 44, 102
 definition of, 7
 DisconnectObject(), 35
 Echo(), 34, 35, 69
 Execute(), 268
 ExpandEnvironmentStrings(), 102
 FileExists(), 191
 GetDrive(), 65
 InputBox(), 45, 69, 71–72
 InstancesOf, 325
 Lbound(), 108
 MsgBox(), 69, 72–74
 objects, 412–414, 417–420
 OpenTextFile(), 191
 Popup(), 44, 69–71
 Quit(), 35
 Raise(), 268, 275
 RegDelete(), 251
 RegRead(), 251
 RegWrite(), 251, 380
 Remote WSH, 407
 Replace(), 268
 Rnd(), 45
 SendKeys, 74–77
 Sleep(), 35
 Test(), 268
 Ubound(), 108
 VBScript run-time, 62–64
 WScript.Quit(), 232, 233

Microsoft Developer Network, 337

Microsoft Windows Script Console, 21

models

core object, 5, 6–7, 34, 411–420
 objects, 58–59

modes, ForReading, 193

motherboard data, retrieving, 332–333

moving

files, 200
 folders, 203
 multiple files, 201

MsgBox() method, 18, 55, 69, 72–74

multi-line text controls, adding, 360–361

multiple-dimension arrays, 105

multiple files

copying, 201
 deleting, 202
 moving, 201

multiple users, 36

N

named values, 250

naming

constants, 86, 87
 variables, 95–96

negation, 99

nesting If statements, 127–128

networks

administrative scripting, 383–386
 drives
 disconnecting, 385–386
 mapping, 383–385
 Microsoft Developer Network, 337

Newton, Isaac, 67

NonExistentFunction() function, 226

NonGuessedString() function, 239–240

normal mode, accessing Windows console, 9–10

notation, Hungarian Notation, 104

NowGoPlay() function, 180

Number property, 226, 269

numbers, generating random, 155

numeric variables, modifying, 99

O

objects, 7

built-in, 265–293. *See also* built-in objects
 Class, 268
 core object models, 34, 411–420
 defining, 255
 Err, 268, 274–275

objects (continued...)

FileSystemObject, 59, 65, 66, 190, 191
 Match, 268
 Matches, 268
 methods, 412–414, 417–420
 properties, 414–417
 SubMatches Collection, 268
 VBScript
 built-in, 59
 models, 58–59
 run-time, 59–61
 WScript, 18, 19
 applying, 35
 formatting, 34
 WshShortcut properties, 160

obsolete statements, removing, 258–259**Office (Windows) applications, 33**

On Error Resume Next **statement, 226**

On Error **statement, 54**

opening

files, 191–194
 Windows console in normal mode,
 9–10

OpenTextFile() **method, 191**

operating systems

compatibility, 15–16
 data, retrieving, 324–326
 Windows Registry, 248
 WMI, 317. *See also* WMI

operators

& (string concatenation), 89
 arithmetic, 97, 99
 VBScript, 135–136
 WQL, 329

optimizing scripts

managing, 169–170
 with procedures, 167–174

Option Explicit **statement, 54, 93, 94**

organizing. *See also* managing

Registry (Windows), 248
 scripts, 165–184

outlining script structures, 304–305

Outlook Express, 21

output, scripts, 68–74

overriding command-line execution
 settings, 38–39

P

<package> **tag, 301–302**

passing arguments to scripts, 114–115

passwords, adding controls, 355

Pattern **property, 269**

patterns, matching

replacing, 276–278
 testing, 279

permissions, 15

PickAGame() **function, 313**

ping **scripts, 399–402**

Planet Trivia game, 123–124, 136–142

assembling, 142
 constants, 137–138
 development, 137
 questions/answers, 139–141
 scores, 141–142
 splash screens, formatting, 138
 starting, 137
 variables, 137–138

player input

testing, 156–157
 verifying, 157

player's answers (Math game), 78

player's selections (Rock, Paper, and
 Scissors game), 45

PlayTheGame() **function, 180–181,
 234–236**

Popup() **method, 44, 69–71**

PowerShell, 8–9

precedence, 51, 99

Preserve **keyword, 110**

printers

administrative scripting, 386–388
 connecting, 386–387
 disconnecting, 387–388

Private **statement, 54**

privileges, configuring, 13

procedures, 165–184

definitions, 86, 167
 events, formatting, 271–274
 Guess a Number game, 170–174
 scope with variables, limiting, 174–175
 scripts, optimizing, 167–174

procedure section (comment template), 58

ProcessRandomNumber() **function, 213–214**

ProcessScriptIniFile() **function,
 210–212**

program data, viewing, 124–135

programming

Knock Knock game, 25–29
 languages, 23
 overview of, 4–21

Programs menu, adding scripts, 161–162

projects. *See also* specific games

BlackJack Lite game, 165–166
 game consoles, 295–296
 Guess a Number game, 143–144,
 154–164
 Hangman game, 219–220, 230–242,
 245–247, 254–262
 Knock Knock game, 3–4
 Lucky Lottery Number Picker game,
 187–188, 207–216
 Math game, 51–53, 74–81
 Planet Trivia game, 123–124, 136–142
 Rock, Paper, and Scissors game,
 31–32, 129–133, 339–340, 370–375
 Story of Captain Adventure, The,
 83–84, 117–120
 Tic Tac Toe game, 265–267

properties

built-in objects, 268
 definitions, 350
 Description, 226, 269
 files, configuring, 41
 FirstIndex, 269
 Global, 269
 HelpContext, 269
 HelpFile, 269
 <HTA:APPLICATION> tag, 344–347
 IgnoreCase, 269
 Length, 269
 Number, 226, 269
 objects, 414–417
 Pattern, 269
 Remote WSH, 408–409
 shortcuts, 158
 Source, 226, 269
 Value, 269
 VBScript run-time, 61–62
 viewing, 40
 WshShortcut object, 160

Properties dialog box, 40

Property Get **statement, 54**

Property Let **statement, 54**

Property Set **statement, 54**

providers (WMI), 320, 322

pseudo code, 125

Public **statement, 54**

- Q**
- queries**
 - cross-namespaces, 328
 - WMI, executing, 327–329
 - question mark (?), 200**
 - questions (Planet Trivia game), 139–141**
 - `Quit()` **method, 35**
- R**
- radio buttons, adding controls, 357–358**
 - `Raise()` **method, 268, 275**
 - Randomize statement, 54, 155**
 - `RandomNumber()` **function, 174**
 - random numbers, generating, 155**
 - random selection, 43, 45**
 - ranking scores (Planet Trivia game), 141–142**
 - reading**
 - files
 - all at once, 199
 - character by character, 198
 - from files, 196–199
 - receiving data, 187–216**
 - `ReDim` **statement, 54, 109**
 - referencing constant values, 87**
 - `RegDelete()` **method, 251**
 - Registry (Windows)**
 - accessing, 250–251
 - keys
 - accessing, 250–251
 - formatting, 251–254
 - modifying, 256
 - overview of, 247–250
 - scripts, configuring, 245–262
 - system information from, retrieving, 253–254
 - values
 - accessing, 250–251
 - formatting, 251–254
 - `RegRead()` **method, 251**
 - regular expressions, applying, 275–280**
 - `RegWrite()` **method, 251, 380**
 - Remote WSH, 405–401, 406–409**
 - removing obsolete statements, 258–259**
 - `Rem` **statement, 54**
 - `Replace()` **method, 268**
 - replacing matching patterns, 276–278**
 - reporting errors, 227–229, 298**
 - reports, automating Word (Microsoft), 394–397**
 - reproducing errors, 223**
 - reserved words, 56**
 - `ResetVariableDefaults()` **function, 215**
 - resizing arrays, 109–110**
 - resources**
 - accessing, 8
 - defining, 44
 - managed. *See* managed resources (WMI)
 - types of, 6
 - WMI, 318. *See also* WMI
 - `<resource>` **tag, 302**
 - results**
 - BlackJack Lite game, 184
 - consoles (game), 315
 - Guess a Number game, 157
 - Hangman game, 242
 - Lucky Lottery Number Picker game, 216
 - Math game, 81
 - Rock, Paper, and Scissors game, 46–47
 - Story of Captain Adventure, The, 120
 - Tic Tac Toe game, 293
 - `RetrieveWord()` **function, 236–237, 259–260**
 - retrieving**
 - BIOS information, 331–332
 - dictionary items, 114
 - event log records, 329–331
 - motherboard data, 332–333
 - operating system data, 324–326
 - system information from Registry, 253–254
 - Windows services information, 326–327
 - reusable code, 170, 303**
 - rights, user, 15**
 - `Rnd()` **function, 155**
 - `Rnd()` **method, 45**
 - Rock, Paper, and Scissors game, 31–32, 43–47, 129–133, 339–340, 370–375**
 - assembling, 375
 - design, 43–44
 - development, 371–375
 - player's selection, collecting, 45
 - random selection, configuring, 45
 - resources, defining, 44
 - results, 46–47
 - rules, displaying, 44
 - script selection, assigning choice, 45
 - root keys (Registry), 248**
- rules**
- CSS, 350
 - displaying (Rock, Paper, and Scissors game), 44
 - styles, adding, 351–352
 - variables, naming, 95–96
 - VBScript syntax, 54–58
- Run as Administrator menu, configuring, 12–13**
- `RunScript()` **function, 313**
- run-time**
- constants, 88–91
 - errors, 221
 - methods, 62–64
 - objects
 - scripts, 64–66
 - VBScript, 59–61
 - properties, VBScript, 61–62
 - scripts, processing data passed at, 114–116
- S**
- scope**
 - limiting, 174–175
 - variables, 96
 - scores, Planet Trivia game, 141–142**
 - screensavers, configuring desktop, 381–383**
 - scripts, 4**
 - administrative scripting, 379–404
 - Apps group, adding, 162
 - arguments, passing, 114–115
 - assembling, 256
 - assigning (Rock, Paper, and Scissors game), 45
 - command prompt, executing, 19–20
 - constants, 85, 86–87
 - data
 - processing passed to at run time, 114–116
 - viewing, 84–85
 - design, accepting argument input, 115–116
 - development, 19
 - documentation, 170
 - engines, 5, 6
 - environments, navigating, 32–33
 - errors, 219–242
 - messages, 221–222
 - overview of, 221–223

scripts (continued...)

- preventing logical errors, 222–223
- reporting, 227–229
- troubleshooting, 223–227
- examples, 427–432
- executing, 15, 91–103
- execution hosts, 5
- formatting, 302–303
- GUIs
 - adding, 339–375
 - comparing HTAs to HTML pages, 341–342
 - elements, 352–365
 - executing HTAs, 342–343
 - formatting HTAs, 343–352
 - navigating HTAs, 340–341
- Hangman game, viewing, 262
- Hello World, creating, 17–19
- Knock Knock game, 25–29
- languages
 - combining, 295–315
 - support, 297
- libraries, 320
- logic, adding looping, 144–154
- managing, 165–184
- optimizing, 169–170
- procedures, optimizing, 167–174
- Programs menu, adding, 161–162
- Registry (Windows), 245–262. *See also* Registry (Windows)
- run-time
 - objects, 64–66
 - processing data passed at, 114–116
- setup, creating, 255–256
- shortcuts, 163–164
- sorting, 37
- storage, configuration settings in external files, 204–207
- structure, outlining, 304–305
- testing, 34
- VBScript, 53, 68–74. *See also* VBScript WMI, 323–336
- <script> tag, 302–303, 347–348
- searching CIM, 336**
- SecondLevelValidation() function, 238–239
- sections of documentation, 58**
- SeeIfWon function, 291–293
- SelectAWordCategory() function, 261–262

- Select Case statement, 54, 124, 133–135
- selecting script engines, 6**
- selectors**
 - CSS, 351
 - definitions, 350
- SendKeys method, 74–77
- services**
 - managing, 389–390
 - Windows, retrieving information about, 326–327
- Set statement, 54
- setup, creating scripts, 255–256**
- SetVariableDefaults() function, 210, 283
- shell scripts**
 - how to work with, 9
 - WSH, comparing to, 7–8
- shortcuts**
 - Guess a Number game, 158–164
 - scripts, 163–164
- shutdown, automating Windows, 402–404**
- single-dimension arrays, 103–104**
- sizing**
 - arrays, 108
 - dynamic arrays, 111
- skipping lines, 198**
- Sleep() method, 35
- sorting scripts, 37**
- Source property, 226, 269
- special folders**
 - applying, 161
 - definitions, 159
- special keystrokes, SendKeys() method, 77**
- SplashScreen() function, 242
- splash screens**
 - Planet Trivia game, 138
 - Story of Captain Adventure, The, 118–119
- Sqr() function, 66
- square-root calculator demo, 66–67**
- starting. See also enabling**
 - an elevated Windows Console, 10–12
 - applications, 366–367
 - Guess a Number game, 155
 - Knock Knock game, 26–27
 - math games, 77–78
 - Planet Trivia game, 137
 - user input, 26–27
- StartService() function, 336

statements, 106

- Const, 86
- declarations, 350
- defining, 299
- Dim, 93
- Do...Until, 145, 152–153
- Do...While, 144, 149–151
- On Error Resume Next, 226
- executing, 18
- Exit Function, 227
- Exit Sub, 227
- For Each...Next, 144, 147–149
- For...Next, 144, 145–147
- If, 124, 125–133
- iterative, 144
- obsolete, removing, 258–259
- Option Explicit, 93, 94
- Randomize, 155
- ReDim, 109
- Select Case, 124, 133–135
- VBScript, 53–54, 85
- While...Wend, 145, 153–154
- StopService() function, 336
- storage**
 - data, 187–216
 - applying WSH, 189–191
 - dictionaries, 113–114
 - modifying during script execution, 91–103
 - Registry (Windows), 249–250
 - scripts, configuration settings in external files, 204–207
- Story of Captain Adventure, The, 83–84, 117–120**
- assembling, 120
- constants, 118
- design, 117
- input, collecting, 119
- results, 120
- splash screen, 118–119
- variables, 118

strings

- concatenation, 89
- constants, 89–91
- definitions, 92
- text, 92

structure, outlining scripts, 304–305**styles, adding rules, 351–352**

- <style> tag, 349–352

- SubMatches Collection object, 268

subroutines, overview of, 167–168Sub **statement, 54****subtypes, variants, 91–92****support**

data types, Registry (Windows), 249

languages, 20–21

PowerShell, 8

scripting languages, 297

VBScript, 21

XML, 297

syntax

errors, 221

rules, VBScript, 54–58

SendKeys() method, 75

system environment variables, 100**system information from Registry, retrieving, 253–254****T****tags**

<button>, 360

formatting, 297

<HTA:APPLICATION>, 343–346

<html>, 342

<input>, 352–353

<script>, 347–348

<style>, 349–352

<textarea>, 360–361

XML, 297–298. *See also* XML

<comment>, 299–300

<job>, 300–301

<?job ?>, 298

<package>, 301–302

<resource>, 302

<script>, 302–303

<?xml ?>, 299

TaskScheduler, 6**templates**

comments, adding, 57

Hangman game, 231–232

Tic Tac Toe game, 281–282

testing

conditions, 126

errors, 78

matching patterns, 279

player input, 156–157

scripts, 34

TestLevelGuess() **function, 239**Test() **method, 268****text**

controls, adding, 353–355

files, viewing, 197

scripts, comparing WSH to shell scripts, 7–8

strings, 92

<textarea> **tag, 360–361****third-party applications, automating execution, 397–399****third-party script engines, compatibility, 21****Tic Tac Toe game, 265–267, 280–293**ClearGameBoard() **function, 283**

design, 281

DisplayBoard() **function, 286**DisplayGameResults() **function, 286–287**DisplaySplashScreen() **function, 293**

initialization sections, 281–282

main processing sections, 282–283

ManageGamePlay() **function, 284–286**MarkPlayerSelection() **function, 290–291**

results, 293

SeeIfWon **function, 291–293**SetVariableDefaults() **function, 283**

templates, 281–282

ValidateInput() **function, 287–290****time**

constants, 88–91

scripts, sorting, 37

tools. See also statements**troubleshooting scripts, 219–242****types**

data

Registry (Windows), 249

VBScript, 91–92

of resources, 6

UUbound() **method, 108**UCase() **function, 112****UNC (universal naming convention), 383****undefined variables, preventing, 94****underscore (_), 95****unnamed values, 250****updating Hangman game, 257–262****uppercase, formatting tags, 297****User Account dialog box, 11****users**

accounts, managing, 391–392

input

collecting (The Story of Captain Adventure), 119

finishing, 28

starting, 26–27

validating, 27–28

multiple, configuring, 36

rights, 15

VValidateAndRun() **function, 311–312**ValidateInput() **function, 287–290****validating**

last responses, 28

user input, 27–28

Value **property, 269****values**

constants, referencing, 87

definitions, 249

deleting, 252

dictionaries, 113

named/unnamed, 250

Registry (Windows)

accessing, 250–251

formatting, 251–254

variables, modifying, 96–100

variables, 44, 83

defining, 93–95, 255

definitions, 85

environments, 100–103

modifying, 96–100

naming, 95–96

numeric, modifying, 99

Planet Trivia game, 137–138

scope, 96, 174–175

Story of Captain Adventure, The, 118

undefined, preventing, 94

variants, 91**VBA (Visual Basic for Applications), 23, 24****VBScript, 3, 5**

applications, integrating, 394–399

arithmetic operators, 97, 99

built-in functions, 66–68, 174, 421–425

built-in objects, 265–293

capabilities, 22

comments, adding, 57–58

VBScript (continued...)

- constants
 - date/time, 88–89
 - run time, 88–91
 - strings, 89
- data types, 91–92
- development, 19
- errors, 221. *See also* errors
- executing, 15
- game consoles, 295–296, 304–315. *See also* consoles (game)
- Hello World, creating, 17–19
- history of, 22–23
- navigating, 51–81
- objects
 - built-in, 59
 - models, 58–59
 - run-time, 59–61
- operators, 135–136
- overview of, 21–25
- reserved words, 56
- run-time
 - methods, 62–64
 - properties, 61–62
- scripts
 - applying run-time objects in, 64–66
 - output, 68–74
 - statements, 53–54, 85
 - support, 21
 - syntax rules, 54–58
 - WSH, functionality, 17
- verifying player input, 157
- versions of WSH, 15–16
- viewing
 - Hello World dialog box, 17
 - program data, 124–135
 - properties, 40
 - scripts
 - data, 84–85
 - Hangman game, 262
 - VBScript, 68–74
 - Story of Captain Adventure, The, 120
 - text files, 197

Visual Basic, 4, 23, 24

Visual Basic for Applications. *See* **VBA**

W

WBEM (Web-Based Enterprise Management), 318

Web-Based Enterprise Management.

See **WBEM**

websites, navigating, 427–432

Weekday() function, 89

While...Wend statement, 54, 145, 153–154

wildcard characters, 200

Windows

- applications, 33
- services, retrieving information
 - about, 326–327
- shutdown, automating, 402–404

Windows Console

- closing, 14
- elevated mode, accessing, 10–13
- Normal mode, accessing, 9–10

Windows GUI, 9. *See also* **GUIs**

Windows Management

Instrumentation. *See* **WMI**

Windows PowerShell, comparing to WSH, 8–9

Windows Registry. *See also* **Registry (Windows)**

- accessing, 250–251
- overview of, 247–250
- scripts, configuring, 245–262

Windows Script Files

- definitions, 297
- executing, 303–304

Windows Script Host. *See* **WSH**

Windows Script Host Settings dialog box, 38

Windows shell, how it works, 9

Windows Task Scheduler, 6

With statement, 54

WMI (Windows Management Instrumentation), 317–336

- infrastructure, 319–323
- overview of, 317–318
- scripts, 323–336

WMI Query Language. *See* **WQL**

Word (Microsoft), automating reports, 394–397

WordPad, 51–52

WordPad (Math game), 79–80

words, reserved, 56

WQL (WMI Query Language), 327–329

wrapping GUIs, ping scripts, 399–402

writing

- characters, 194–195
- comments, 57

- to files, 194–196

- JSript to consoles (game), 305–306, 315

- Knock Knock game, 25–29

- lines, 195

- reusable code, 170

- shell scripts, 8

WScript.exe, 6

- command-line execution, configuring, 36–37

- customizing, 39–41

- desktop execution, configuring, 37–38

WScript objects, 18, 19

- applying, 35

- formatting, 34

WScript.Quit() method, 232, 233

WSH (Windows Script Host), 3

- administrative scripting, 379–404

- applying, 189–191

- command prompt, navigating, 14

- core object models, 6–7, 411–420

- disabling, 42–43

- enabling, 41–42

- HTAs, integrating, 366–370

- installing, 16–17

- navigating, 31–47

- operating system compatibility, 15–16

- overview of, 4–21

- Remote, 405–401

- scripting engines, 5, 6

- shell scripts, comparing to, 7–8

- VBScript, 17, 21. *See also* **VBScript**

- Windows PowerShell, comparing to, 8–9

WshShortcut object, properties, 160

X

XML (Extensible Markup Language), 297

- consoles (game), using to outline

- script structure, 304–305

- tags, 297–298

- <comment>, 299–300

- <job>, 300–301

- <?job ?>, 298

- <package>, 301–302

- <resource>, 302

- <script>, 302–303

- <?xml ?>, 299