Paul McFedries

Second Edition

Includes coverage of Windows Vista Service Pack 1!

Microsoft® Windows Vista™

UNLEASHED

SAMS

Microsoft® Windows® Vista™ Unleashed, Second Edition

ISBN-13: 978-0-672-33013-1
ISBN-10: 0-672-33013-X

Library of Congress Cataloging-in-Publication data

McFedries, Paul.

Microsoft Windows Vista unleashed / Paul McFedries. – 2nd ed.

p. cm.

Includes index.

ISBN-10: 0-672-33013-X

ISBN-13: 978-0-672-33013-1

1. Microsoft Windows (Computer file) 2. Operating systems (Computers)

I. Title.

QA76.76.063M398175 2008

005.4'46–dc22

2008010864

Printed in the United States on America

First Printing March 2008

Trademarks

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

Warning and Disclaimer

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

Bulk Sales

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

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact

International Sales
international@pearson.com

Publisher
Paul Boger

Associate Publisher
Greg Wiegand

Acquisitions Editor
Rick Kughen

Development Editor
Rick Kughen

Managing Editor
Patrick Kanouse

Project Editor
Seth Kerney

Copy Editor
Margo Catts

Proofreader
Paula Lowell

Publishing Coordinator
Cindy Teeters

Book Designer
Gary Adair

Composition
Mark Shirar

Introduction

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

—T. S. Eliot

My goal in writing *Microsoft Windows Vista Unleashed* is to cover the good, the bad, and, yes, even the ugly of Windows Vista. In particular, I give you complete coverage of the intermediate-to-advanced features of Windows Vista. This means that I bypass basic topics, such as wielding the mouse, in favor of more complex operations, such as working with the Registry, maintaining and troubleshooting your system, networking, and getting around the Internet.

I've tried to keep the chapters focused on the topic at hand and unburdened with long-winded theoretical discussions. However, there are plenty of situations in which you won't be able to unleash the full power of Windows Vista and truly understand what's going on unless you have a solid base on which to stand. In these cases, I'll give you whatever theory and background you need to get up to speed. From there, I'll get right down to brass tacks without any further fuss and bother.

Who Should Read This Book

All writers write with an audience in mind. Actually, I'm not sure whether that's true for novelists and poets and the like, but it *should* be true for any technical writer who wants to create a useful and comprehensible book. Here are the members of my own imagined audience:

- **IT professionals**—These brave souls must decide whether to move to Vista, work out deployment issues, and support the new Vista desktops. The whole book has information related to your job and Vista.
- **Power users**—These elite users get their power via knowledge. With that in mind, this book extends the Windows power user's know-how by presenting an exhaustive account of everything that's new and improved in Windows Vista.
- **Business users**—If your company is thinking of or has already committed to moving to Vista, you need to know what you, your colleagues, and your staff are getting into. You also want to know what Vista will do to improve your productivity and make your life at the office easier. You learn all of this and more in this book.

- **Road warriors**—If you travel for a living, you probably want to know what Vista brings to the remote computing table. Will you be able to synchronize data, connect to the network, and manage power better than before? What other new notebook features can be found in Vista? You'll find out in this book.
- **Small business owners**—If you run a small or home business, you probably want to know whether Vista will give you a good return on investment. Will it make it easier to set up and maintain a network? Will Vista computers be more stable? Will your employees be able to collaborate easier? The answer turns out to be "Yes" for all of these questions, and I'll show you why.
- **Multimedia users**—If you use your computer to listen to music or radio stations, watch TV, work with digital photographs, edit digital movies, or burn CDs and DVDs, you'll be interested to know that Vista has a handful of new features that affect all of these activities.

Also, to keep the chapters uncluttered, I've made a few assumptions about what you know and what you don't know:

- I assume that you have knowledge of rudimentary computer concepts such as files and folders.
- I assume that you're familiar with the basic Windows skills: mouse maneuvering, dialog box negotiation, pull-down menu jockeying, and so on.
- I assume that you can operate peripherals attached to your computer, such as the keyboard and printer.
- I assume that you've used Windows for a while and are comfortable with concepts such as toolbars, scrollbars, and, of course, windows.
- I assume that you have a brain that you're willing to use and a good supply of innate curiosity.

How This Book Is Organized

To help you find the information you need, this book is divided into seven parts that group related tasks. The next few sections offer a summary of each part.

Part I: Unleashing Day-to-Day Windows Vista

Part I takes your basic, workaday Windows chores and reveals their inner mysteries, allowing you to become more productive. After an initial chapter on what's new in Vista, topics include the myriad ways to get Windows Vista off the ground (Chapter 2), how to use Windows Vista to work with files and folders (Chapter 3), getting the most out of file types (Chapter 4), installing and running applications (Chapter 5), working with user accounts (Chapter 6), dealing with digital media (Chapter 7), using Contacts, Calendar, and faxing (Chapter 8), and Vista's mobile computing tools (Chapter 9).

Part II: Unleashing Essential Windows Vista Power Tools

The chapters in Part II get your advanced Windows Vista education off to a flying start by covering the ins and outs of four important Vista power tools: Control Panel and group policies (Chapter 10), the Registry (Chapter 11), and the Windows Script Host (Chapter 12).

Part III: Unleashing Windows Vista Customization and Optimization

In Part III, you dive into the deep end of advanced Windows work: customizing the interface (Chapter 13), performance tuning (Chapter 14), maintaining Windows Vista (Chapter 15), troubleshooting problems (Chapter 16), and working with devices (Chapter 17).

Part IV: Unleashing Windows Vista for the Internet

Part IV shows you how to work with Windows Vista's Internet features. You learn how to get the most out of a number of Internet services, including the Web (Chapter 18), email (Chapter 19), and newsgroups (Chapter 20). I close this part with an extensive look at the Internet security and privacy feature that come with Windows Vista (Chapter 21).

Part V: Unleashing Windows Vista Networking

To close out the main part of this book, Part V takes an in-depth look at Windows Vista's networking features. You learn how to set up a small network (Chapter 22), how to access and use that network (Chapter 23), and how to access your network from remote locations (Chapter 24).

Part VI: Appendixes

To further your Windows Vista education, Part VI presents a few appendixes that contain extra goodies. You'll find complete list of Windows Vista shortcut keys (Appendix A), a detailed look at using the Windows Vista command prompt (Appendix B), and a batch file primer (Appendix C).

What's New in the Second Edition

Microsoft Windows Vista Unleashed 2E includes coverage of the new features that are part of Windows Vista Service Pack 1 (SP1). Most of those features are under-the-hood tweaks that improve Vista's performance, reliability, security, application compatibility, and driver support. I'll talk about those where appropriate, and of course I'll also talk about any changes that SP1 makes to the Vista interface. (For example, in Chapter 15, "Maintaining Your Windows Vista System," I talk about the new Disk Defragmenter feature that enables you to select which disks get defragmented.)

Conventions Used in This Book

To make your life easier, this book includes various features and conventions that help you get the most out of this book and Windows Vista itself:

Steps	Throughout the book, I've broken many Windows Vista tasks into easy-to-follow step-by-step procedures.
Things you type	Whenever I suggest that you type something, what you type appears in a **`bold monospace`** font.
Filenames, folder names, and code	These things appear in a `monospace` font.
Commands	Commands and their syntax use the `monospace` font as well. Command placeholders (which stand for what you actually type) appear in an *`italic monospace`* font.
Pull-down menu commands	I use the following style for all application menu commands: *Menu, Command*, where *Menu* is the name of the menu that you pull down and *Command* is the name of the command you select. Here's an example: File, Open. This means that you pull down the File menu and select the Open command.
Code continuation character	When a line of code is too long to fit on only one line of this book, it is broken at a convenient place and continued to the next line. The continuation of the line is preceded by a code continuation character (➥). You should type a line of code that has this character as one long line without breaking it.

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

NOTE

The Note box presents asides that give you more information about the current topic. These tidbits provide extra insights that give you a better understanding of the task. In many cases, they refer you to other sections of the book for more information.

TIP

The Tip box tells you about Windows Vista methods that are easier, faster, or more efficient than the standard methods.

CAUTION

The all-important Caution box tells you about potential accidents waiting to happen. There are always ways to mess things up when you're working with computers. These boxes help you avoid at least some of the pitfalls.

CHAPTER 12

Programming the Windows Script Host

IN THIS CHAPTER

In Appendix C, "Automating Windows XP with Batch Files," you learn how to tame the command prompt by creating **batch files**—small, executable text files that run one or more commands. You'll see that with a little ingenuity and a dash of guile, it's possible to make batch files perform some interesting and useful tasks. Indeed, for many years, batch files were the only way to automate certain kinds of tasks. Unfortunately, the batch file world is relentlessly command-line–oriented. So, with the exception of being able to launch Windows programs, batch files remain ignorant of the larger Windows universe.

If you're looking to automate a wider variety of tasks in Windows, you need to supplement your batch file knowledge with scripts that can deal with the Registry, shortcuts, files, and network drives, and that can even interact with Windows programs via Automation. The secret to these powerful scripts is the **Windows Script Host** (**WSH**). This chapter introduces you to the Windows Script Host, shows you how to execute scripts, and runs through the various elements in the Windows Script Host object model.

WSH: Your Host for Today's Script

As you might know, Internet Explorer is really just an empty container application that's designed to host different data formats, including ActiveX controls, various file formats (such as Microsoft Word documents and Microsoft Excel worksheets), and several ActiveX scripting engines. A **scripting engine** is a dynamic link library (DLL) that provides programmatic support for a particular scripting language. Internet Explorer supports two such scripting engines: VBScript (`VBScript.dll`) and JavaScript

(`JScript.dll`). This enables web programmers to write small programs—**scripts**—that interact with the user, control the browser, set cookies, open and close windows, and more. Although these scripting engines don't offer full-blown programmability (you can't compile scripts, for example), they do offer modern programming structures such as loops, conditionals, variables, objects, and more. In other words, they're a huge leap beyond what a mere batch file can do.

The Windows Script Host is also a container application, albeit a scaled-down application in that its only purpose in life is to host scripting engines. Right out of the box, the Windows Script Host supports both the VBScript and JavaScript engines. However, Microsoft designed the Windows Script Host to be a universal host that can support any ActiveX-based scripting engine. Therefore, there are also third-party vendors offering scripting engines for languages such as Perl, Tcl, and Rexx.

The key difference between Internet Explorer's script hosting and the Windows Script Host is the environment in which the scripts run. Internet Explorer scripts are web page–based, so they control and interact with either the web page or the web browser. The Windows Script Host runs scripts within the Windows Vista shell or from the command prompt, so you use these scripts to control various aspects of Windows. Here's a sampling of the things you can do:

- Execute Windows programs
- Create and modify shortcuts
- Use Automation to connect and interact with Automation-enabled applications such as Microsoft Word, Outlook, and Internet Explorer
- Read, add, and delete Registry keys and items
- Access the VBScript and JavaScript object models, which give access to the file system, runtime error messages, and more
- Use pop-up dialog boxes to display information to the user, and determine which button the user clicked to dismiss the dialog box
- Read environment variables, which are system values that Vista keeps in memory, such as the folder into which Vista is installed—the `%SystemRoot%` environment variable—and the name of the computer—the `%ComputerName%` environment variable
- Deal with network resources, including mapping and unmapping network drives, accessing user data (such as the username and user domain), and connecting and disconnecting network printers

Clearly, we've gone *way* beyond batch files!

What about speed? After all, you wouldn't want to load something that's the size of Internet Explorer each time you need to run a simple script. That's not a problem because, as I've said, the Windows Script Host does nothing but host scripting engines, so

it has much less memory overhead than Internet Explorer. That means that your scripts run quickly. For power users looking for a Windows-based batch language, the Windows Script Host is a welcome tool.

NOTE

This chapter does not teach you how to program in either VBScript or JavaScript and, in fact, assumes that you're already proficient in one or both of these languages. If you're looking for a programming tutorial, my *VBA for the 2007 Microsoft Office System* (Que, 2007) is a good place to start (VBScript is a subset of VBA—Visual Basic for Applications). For JavaScript, try my *Special Edition Using JavaScript* (Que, 2001).

Scripts and Script Execution

Scripts are simple text files that you create using Notepad or some other text editor. You can use a word processor such as WordPad to create scripts, but you must make sure that you save these files using the program's Text Only document type. For VBScript, a good alternative to Notepad is the editor that comes with either Visual Basic or any program that supports VBA (such as the Office suite). Just remember that VBScript is a subset of VBA (which is, in turn, a subset of Visual Basic), so it does not support all objects and features.

In a web page, you use the `<script>` tag to specify the scripting language you're using, as in this example:

```
<SCRIPT LANGUAGE="VBScript">
```

With the Windows Script Host, the script file's extension specifies the scripting language:

- For VBScript, save your text files using the `.vbs` extension (which is registered as the following file type: VBScript Script File).
- For JavaScript, use the `.js` extension (which is registered as the following file type: JScript Script File).

As described in the next three sections, you have three ways to run your scripts: by launching the script files directly, by using `WSscript.exe`, or by using `CScript.exe`.

Running Script Files Directly

The easiest way to run a script from within Windows is to launch the `.vbs` or `.js` file directly. That is, you either double-click the file in Windows Explorer or type the file's path and name in the Run dialog box. Note, however, that this technique does not work at the command prompt. For that, you need to use the CScript program described a bit later.

Using WScript for Windows-Based Scripts

The `.vbs` and `.js` file types have an `open` method that's associated with WScript (`WScript.exe`), which is the Windows-based front-end for the Windows Script Host. In other words, launching a script file named `MyScript.vbs` is equivalent to entering the following command in the Run dialog box:

```
wscript myscript.vbs
```

The WScript host also defines several parameters that you can use to control how the script executes. Here's the full syntax:

```
WSCRIPT filename arguments //B //D //E:engine //H:host //I //Job:xxxx
➥//S //T:ss //X
```

filename	Specifies the filename, including the path of the script file, if necessary.
arguments	Specifies optional arguments required by the script. An **argument** is a data value that the script uses as part of its procedures or calculations.
//B	Runs the script in batch mode, which means script errors and `Echo` method output lines are suppressed. (I discuss the `Echo` method later in this chapter.)
//D	Enables Active Debugging. If an error occurs, the script is loaded into the Microsoft Script Debugger (if it's -installed) and the offending statement is highlighted.
//E:*engine*	Executes the script using the specified scripting *engine*, which is the scripting language to use when running the script.
//H:*host*	Specifies the default scripting host. For *host*, use either `CScript` or `WScript`.
//I	Runs the script in interactive mode, which displays script errors and `Echo` method output lines.
//Job:*xxxx*	In a script file that contains multiple jobs, executes only the job with `id` attribute equal to *xxxx*.
//S	Saves the specified WScript arguments as the default for the current user; uses the following Registry key to save the settings: `HKCU\Software\Microsoft\Windows Script Host\Settings`
//TT:*ss*	Specifies the maximum time in seconds (*ss*) that the script can run before it shuts down automatically.
//X	Executes the entire script in the Microsoft Script Debugger (if it's installed).

For example, the following command runs `MyScript.vbs` in batch mode with a 60-second maximum execution time:

```
wscript myscript.vbs //B //TT:60
```


Creating Script Jobs

A script **job** is a section of code that performs a specific task or set of tasks. Most script files contain a single job. However, it's possible to create a script file with multiple jobs. To do this, first surround the code for each job with the `<script>` and `</script>` tags, and then surround those with the `<job>` and `</job>` tags. In the `<job>` tag, include the `id` attribute and set it to a unique value that identifies the job. Finally, surround all the jobs with the `<package>` and `</package>` tags. Here's an example:

```
<package>
<job id="A">
<script language="VBScript">
    WScript.Echo "This is Job A."
</script>
</job>

<job id="B">
<script language="VBScript">
      WScript.Echo "This is Job B."
</script>
</job>
</package>
```

Save the file using the `.wsf` (Windows Script File) extension.

NOTE

If you write a lot of scripts, the Microsoft Script Debugger is an excellent programming tool. If there's a problem with a script, the debugger can help you pinpoint its location. For example, the debugger enables you to step through the script's execution one statement at a time. If you don't have the Microsoft Script Debugger, you can download a copy from msdn.microsoft.com/scripting.

Using CScript for Command-Line Scripts

The Windows Script Host has a second host front-end application called CScript (`CScript.exe`), which enables you to run scripts from the command line. In its simplest form, you launch CScript and use the name of the script file (and its path, if required) as a parameter, as in this example:

```
cscript myscript.vbs
```

The Windows Script Host displays the following banner and then executes the script:

```
Microsoft (R) Windows Script Host Version 5.7 for Windows
Copyright  Microsoft Corporation. All rights reserved.
```

As with WScript, the CScript host has an extensive set of parameters you can specify:

```
CSCRIPT filename arguments //B //D //E:engine //H:host //I //Job:xxxx
➥//S //T:ss //X //U //LOGO //NOLOGO
```

This syntax is almost identical to that of WScript, but adds the following three parameters:

`//LOGO`	Displays the Windows Script Host banner at startup
`//NOLOGO`	Hides the Windows Script Host banner at startup
`//U`	Uses Unicode for redirected input/output from the console

Script Properties and `.wsh` Files

In the last two sections, you saw that the WScript and CScript hosts have a number of parameters you can specify when you execute a script. It's also possible to set some of these options by using the properties associated with each script file. To see these properties, right-click a script file and then click Properties. In the properties sheet that appears, display the Script tab, shown in Figure 12.1. You have two options:

- **Stop Script After Specified Number of Seconds**—If you activate this check box, Windows shuts down the script after it has run for the number of seconds specified in the associated spin box. This is useful for scripts that might hang during execution. For example, a script that attempts to enumerate all the mapped network drives at startup might hang if the network is unavailable.
- **Display Logo When Script Executed in Command Console**—As you saw in the previous section, the CScript host displays some banner text when you run a script at the command prompt. If you deactivate this check box, the Windows Script Host suppresses this banner (unless you use the `//LOGO` parameter).

When you make changes to these properties, the Windows Script Host saves your settings in a new file that has the same name as the script file, except with the `.wsh` (Windows Script Host Settings) extension. For example, if the script file is `MyScript.vbs`, the settings are stored in `MyScript.wsh`. These `.wsh` files are text files organized into sections, much like `.ini` files. Here's an example:

```
[ScriptFile]
Path=C:\Users\Paul\Documents\Scripts\Popup1.vbs
[Options]
Timeout=0
DisplayLogo=1
```

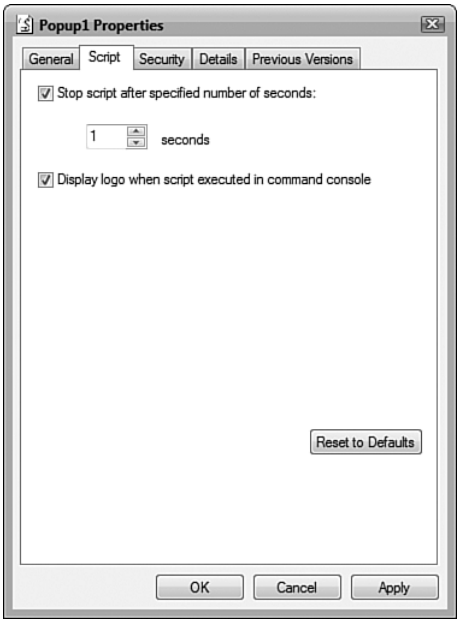

FIGURE 12.1 In a script file's properties sheet, use the Script tab to set some default options for the script.

To use these settings when running the script, use either WScript or CScript and specify the name of the `.wsh` file:

```
wscript myscript.wsh
```

NOTE

Rather than setting properties for individual scripts, you might prefer to set global properties that apply to the WScript host itself. Those global settings then apply to every script that runs using the WScript host. To do this, run `WScript.exe` without any parameters. This displays the properties sheet for WScript, which contains only the Script tab shown in Figure 12.1. The settings you choose in the properties sheet are stored in the following Registry key:

HKLM\Software\Microsoft\Windows Script Host\Settings

Programming Objects

Although this chapter isn't a programming primer per se, I'd like to take some time now to run through a few quick notes about programming objects. This will serve you well throughout the rest of the chapter as I take you on a tour of the Windows Script Host object model.

The dictionary definition of an object is "anything perceptible by one or more of the senses, especially something that can be seen and felt." In scripting, an **object** is an application element that exposes an interface to the programmer, who can then perform the programming equivalent of seeing and feeling:

- You can make changes to the object's *properties* (this is the seeing part).
- You can make the object perform a task by activating a *method* associated with the object (this is the feeling part).

Working with Object Properties

Every programmable object has a defining set of characteristics. These characteristics are the object's **properties**, and they control the appearance and position of the object. For example, the `WScript` object (the top-level Windows Script Host object) has an `Interactive` property that determines whether the script runs in interactive mode or batch mode.

When you refer to a property, you use the following syntax:

```
Object.Property
```

`Object`	The name of the object
`Property`	The name of the property with which you want to work

For example, the following expression refers to the `Interactive` property of the `WScript` object:

```
WScript.Interactive
```

Setting the Value of a Property

To set a property to a certain value, you use the following syntax:

```
Object.Property = value
```

Here, *`value`* is an expression that specifies the value to which you want to set the property. As such, it can be any of the scripting language's recognized data types, which usually include the following:

- A numeric value
- A string value, enclosed in double quotation marks (such as `"My Script Application"`)
- A logical value (in VBScript: `True` or `False`; in JavaScript: `true` or `false`)

For example, the following VBScript statement tells the Windows Script Host to run the script using interactive mode:

```
WScript.Interactive = True
```


Returning the Value of a Property

Sometimes you need to know the current setting of a property before changing the property or performing some other action. You can find out the current value of a property by using the following syntax:

```
variable = Object.Property
```

Here, *variable* is a variable name or another property. For example, the following statement stores the current script mode (batch or interactive) in a variable named `currentMode`:

```
currentMode = WScript.Interactive
```

Working with Object Methods

An object's properties describe what the object is, whereas its **methods** describe what the object *does*. For example, the `WScript` object has a `Quit` method that enables you to stop the execution of a script.

How you refer to a method depends on whether the method requires any arguments. If it doesn't, the syntax is similar to that of properties:

```
Object.Method
```

Object	The name of the object
Method	The name of the method you want to run

For example, the following statement shuts down a script:

```
WScript.Quit
```

If the method requires arguments, you use the following syntax:

```
Object.Method(Argument1, Argument2, ...)
```

NOTE

In VBScript, the parentheses around the argument list are necessary only if you'll be storing the result of the method in a variable or object property. In JavaScript, the parentheses are always required.

For example, the `WshShell` object has a `RegWrite` method that you use to write a key or value to the Registry. (I discuss this object and method in detail later in this chapter; see "Working with Registry Entries.") Here's the syntax:

```
WshShell.RegWrite strName, anyValue[, strType]
```

strName	The name of the Registry key or value
anyValue	The value to write, if *strName* is a Registry value
strType	The data type of the value

Argument Naming Conventions

When presenting method arguments in this chapter, I'll follow Microsoft's naming conventions, including the use of the following prefixes for the argument names:

Prefix	*Data Type*
`any`	Any type
`b`	Boolean
`int`	Integer
`nat`	Natural numbers
`obj`	Object
`str`	String

For many object methods, not all the arguments are required. In the `RegWrite` method, for example, the *strName* and *anyValue* arguments are required, but the *strType* argument is not. Throughout this chapter, I differentiate between required and optional arguments by surrounding the optional arguments with square brackets—for example, [*strType*].

For example, the following statement creates a new value named `Test` and sets it equal to `Foo`:

```
WshShell.RegWrite "HKCU\Software\Microsoft\Windows Script Host\Test",
➥"Foo", "REG_SZ"
```

Assigning an Object to a Variable

If you're using JavaScript, you assign an object to a variable using a standard variable assignment:

```
var variableName = ObjectName
```

variableName	The name of the variable
ObjectName	The object you want to assign to the variable

In VBScript, you assign an object to a variable by using the `Set` statement. `Set` has the following syntax:

```
Set variableName = ObjectName
```

variableName	The name of the variable
ObjectName	The object you want to assign to the variable

You'll see later on that you must often use Automation to access external objects. For example, if you want to work with files and folders in your script, you must access the scripting engine object named `FileSystemObject`. To get this access, you use the `CreateObject` method and store the resulting object in a variable, like so:

```
Set fs = CreateObject("Scripting.FileSystemObject")
```

Working with Object Collections

A **collection** is a set of similar objects. For example, `WScript.Arguments` is the set of all the arguments specified on the script's command line. Collections are objects, too, so they have their own properties and methods, and you can use these properties and methods to manipulate one or more objects in the collection.

The members of a collection are **elements**. You can refer to individual elements by using an **index**. For example, the following statement refers to the first command-line argument (collection indexes always begin at 0):

```
WScript.Arguments(0)
```

If you don't specify an element, the Windows Script Host assumes that you want to work with the entire collection.

VBScript: Using `For Each...Next` Loops for Collections

As you might know, VBScript provides the `For...Next` loop that enables you to cycle through a chunk of code a specified number of times. For example, the following code loops 10 times:

```
For counter = 1 To 10
    Code entered here is repeated 10 times
Next counter
```

A useful variation on this theme is the `For Each...Next` loop, which operates on a collection of objects. You don't need a loop counter because VBScript loops through the individual elements in the collection and performs on each element whatever operations are inside the loop. Here's the structure of the basic `For Each...Next` loop:

```
For Each element In collection
    [statements]
Next
```

element	A variable used to hold the name of each element in the collection
collection	The name of the collection
statements	The statements to execute for each element in the collection

The following code loops through all the arguments specified on the script's command line and displays each one:

```
For Each arg In WScript.Arguments
    WScript.Echo arg
Next
```

JavaScript: Using Enumerators and `for` Loops for Collections

To iterate through a collection in JavaScript, you must do two things: create a new `Enumerator` object and use a `for` loop to cycle through the enumerated collection.

To create a new `Enumerator` object, use the `new` keyword to set up an object variable (where *`collection`* is the name of the collection you want to work with):

```
var enum = new Enumerator(collection)
```

Then set up a special `for` loop:

```
for (; !enumerator.atEnd(); enumerator.moveNext())
{
    [statements];
}
```

enumerator	The `Enumerator` object you created
statements	The statements to execute for each element in the collection

The `Enumerator` object's `moveNext` method runs through the elements in the collection, whereas the `atEnd` method shuts down the loop after the last item has been processed. The following code loops through all the arguments specified on the script's command line and displays each one:

```
var args = new Enumerator(WScript.Arguments);
for (; !args.atEnd(); args.moveNext())
{
   WScript.Echo(args.item());
}
```

Programming the `WScript` Object

The `WScript` object represents the Windows Script Host applications (`WScript.exe` and `CScript.exe`). You use this object to get and set certain properties of the scripting host, as well as to access two other objects: `WshArguments` (the `WScript` object's `Arguments` prop-

erty) and WshScriptEngine (accessed via the WScript object's GetScriptEngine method). WScript also contains the powerful CreateObject and GetObject methods, which enable you to work with Automation-enabled applications.

Displaying Text to the User

The WScript object method that you'll use most often is the Echo method, which displays text to the user. Here's the syntax:

```
WScript.Echo [Argument1, Argument2,...]
```

Here, *Argument1*, *Argument2*, and so on, are any number of text or numeric values that represent the information you want to display to the user. In the Windows-based host (WScript.exe), the information displays in a dialog box; in the command-line host (CScript.exe), the information displays at the command prompt (much like the command-line ECHO utility).

Shutting Down a Script

You use the WScript object's Quit method to shut down the script. You can also use Quit to have your script return an error code by using the following syntax:

```
WScript.Quit [intErrorCode]
```

intErrorCode	An integer value that represents the error code you want to return

You could then call the script from a batch file and use the ERRORLEVEL environment variable to deal with the return code in some way. (See Appendix C for more information on ERRORLEVEL.)

Scripting and Automation

Applications such as Internet Explorer and Word come with (or **expose**, in the jargon) a set of objects that define various aspects of the program. For example, Internet Explorer has an Application object that represents the program as a whole. Similarly, Word has a Document object that represents a Word document. By using the properties and methods that come with these objects, it's possible to programmatically query and manipulate the applications. With Internet Explorer, for example, you can use the Application object's Navigate method to send the browser to a specified web page. With Word, you can read a Document object's Saved property to see whether the document has unsaved changes.

This is powerful stuff, but how do you get at the objects that these applications expose? You do that by using a technology called **Automation**. Applications that support Automation implement object libraries that expose the application's native objects to Automation-aware programming languages. Such applications are **Automation servers**, and the applications that manipulate the server's objects are **Automation controllers.** The Windows Script Host is an Automation controller that enables you to write script code to control any server's objects.

This means that you can use an application's exposed objects more or less as you use the Windows Script Host objects. With just a minimum of preparation, your script code can refer to and work with the Internet Explorer `Application` object, or the Microsoft Word `Document` object, or any of the hundreds of other objects exposed by the applications on your system. (Note, however, that not all applications expose objects. Windows Mail and most of the built-in Windows Vista programs—such as WordPad and Paint—do not expose objects.)

Creating an Automation Object with the `CreateObject` Method

The `WScript` object's `CreateObject` method creates an Automation object (specifically, what programmers call an **instance** of the object). Here's the syntax:

```
WScript.CreateObject(strProgID)
```

strProgID	A string that specifies the Automation server application and the type of object to create. This string is a **programmatic identifier**, which is a label that uniquely specifies an application and one of its objects. The programmatic identifier always takes the following form: *AppName.ObjectType* Here, *AppName* is the Automation name of the application and *ObjectType* is the object class type (as defined in the Registry's `HKEY_CLASSES_ROOT` key). For example, here's the programmatic ID for Word: `Word.Application`

Note that you normally use `CreateObject` within a `Set` statement, and that the function serves to create a new instance of the specified Automation object. For example, you could use the following statement to create a new instance of Word's `Application` object:

```
Set objWord = CreateObject("Word.Application")
```

You need to do nothing else to use the Automation object. With your variable declared and an instance of the object created, you can use that object's properties and methods directly. Listing 12.1 shows a VBScript example (you must have Word installed for this to work).

LISTING 12.1 A VBScript Example That Creates and Manipulates a Word `Application` Object

```
' Create the Word Application object
'
Set objWord = WScript.CreateObject("Word.Application")
'
' Create a new document
'
objWord.Documents.Add
'
' Add some text
'
objWord.ActiveDocument.Paragraphs(1).Range.InsertBefore "Automation test."
'
' Save the document
'
objWord.ActiveDocument.Save
'
' We're done, so quit Word
'
objWord.Quit
```


This script creates and saves a new Word document by working with Word's `Application` object via Automation. The script begins by using the `CreateObject` method to create a new Word `Application` object, and the object is stored in the `objWord` variable. From there, you can wield the `objWord` variable just as though it were the Word `Application` object.

For example, the `objWord.Documents.Add` statement uses the `Documents` collection's `Add` method to create a new Word document, and the `InsertBefore` method adds some text to the document. The `Save` method then displays the Save As dialog box so that you can save the new file. With the Word-related chores complete, the `Application` object's `Quit` method runs to shut down Word. For comparison, Listing 12.2 shows a JavaScript procedure that performs the same tasks.

LISTING 12.2 A JavaScript Example That Creates and Manipulates a Word `Application` Object

```
// Create the Word Application object
//
var objWord = WScript.CreateObject("Word.Application");
//
// Create a new document
//
objWord.Documents.Add();
//
// Add some text
//
objWord.ActiveDocument.Paragraphs(1).Range.InsertBefore("Automation test.");
//
// Save the document
//
objWord.ActiveDocument.Save();
//
// We're done, so quit Word
//
objWord.Quit();
```

Making The Automation Server Visible

The `CreateObject` method loads the object, but doesn't display the Automation server unless user interaction is required. For example, you see Word's Save As dialog box when you run the `Save` method on a new document (as in Listings 12.1 and 12.2). Not seeing the Automation server is the desired behavior in most Automation situations. However, if you *do* want to see what the Automation server is up to, set the `Application` object's `Visible` property to `True`, as in this example:

```
objWord.Visible = True
```

Working with an Existing Object Using the `GetObject` Method

If you know that the object you want to work with already exists or is already open, the `CreateObject` method isn't the best choice. In the example in the previous section, if Word is already running, the code will start a second copy of Word, which is a waste of resources. For these situations, it's better to work directly with the existing object. To do that, use the `GetObject` method:

```
WScript.GetObject(strPathname, [strProgID])
```

`strPathname`	The pathname (drive, folder, and filename) of the file you want to work with (or the file that contains the object you want to work with). If you omit this argument, you have to specify the *`strProgID`* argument.

strProgID	The programmatic identifier that specifies the Automation server application and the type of object to work with (that is, the *AppName.ObjectType* class syntax).

Listing 12.3 shows a VBScript procedure that puts the GetObject method to work.

LISTING 12.3 A VBScript Example That Uses the GetObject Method to Work with an Existing Instance of a Word Document Object

```
' Get the Word Document object
'
Set objDoc = WScript.GetObject("C:\GetObject.doc", "Word.Document")
'
' Get the word count
'
WScript.Echo objDoc.Name & " has " & objDoc.Words.Count & " words."
'
' We're done, so quit Word
'
objDoc.Application.Quit
```

The GetObject method assigns the Word Document object named GetObject.doc to the objDoc variable. After you've set up this reference, you can use the object's properties and methods directly. For example, the Echo method uses objDoc.Name to return the filename and objDoc.Words.Count to determine the number of words in the document.

Note that although you're working with a Document object, you still have access to Word's Application object. That's because most objects have an Application property that refers to the Application object. In the script in Listing 12.3, for example, the following statement uses the Application property to quit Word:

```
objDoc.Application.Quit
```

Exposing VBScript and JavaScript Objects

One of the most powerful uses for scripted Automation is accessing the object models exposed by the VBScript and JavaScript engines. These models expose a number of objects, including the local file system. This enables you to create scripts that work with files, folders, and disk drives, read and write text files, and more. You use the following syntax to refer to these objects:

```
Scripting.ObjectType
```

Scripting is the Automation name of the scripting engine, and *ObjectType* is the class type of the object.

NOTE

This section just gives you a brief explanation of the objects associated with the VBScript and JavaScript engines. For the complete list of object properties and methods, please see the following site: msdn.microsoft.com/scripting.

Programming the `FileSystemObject`

`FileSystemObject` is the top-level file system object. For all your file system scripts, you begin by creating a new instance of `FileSystemObject`:

In VBScript:

```
Set fs = WScript.CreateObject("Scripting.FileSystemObject")
```

In JavaScript:

```
var fs = WScript.CreateObject("Scripting.FileSystemObject");
```

Here's a summary of the file system objects you can access via Automation and the top-level `FileSystemObject`:

- `Drive`—This object enables you to access the properties of a specified disk drive or UNC network path. To reference a `Drive` object, use either the `Drives` collection (discussed next) or the `FileSystemObject` object's `GetDrive` method. For example, the following VBScript statement references drive C:

  ```
  Set objFS = WScript.CreateObject("Scripting.FileSystemObject")
  Set objDrive = objFS.GetDrive("C:")
  ```

- `Drives`—This object is the collection of all available drives. To reference this collection, use the `FileSystemObject` object's `Drives` property:

  ```
  Set objFS = WScript.CreateObject("Scripting.FileSystemObject")
  Set objDrives = objFS.Drives
  ```

- `Folder`—This object enables you to access the properties of a specified folder. To reference a `Folder` object, use either the `Folders` collection (discussed next) or the `FileSystemObject` object's `GetFolder` method:

  ```
  Set objFS = WScript.CreateObject("Scripting.FileSystemObject")
  Set objFolder = objFS.GetFolder("C:\My Documents")
  ```

- `Folders`—This object is the collection of subfolders within a specified folder. To reference this collection, use the `Folder` object's `Subfolders` property:

  ```
  Set objFS = WScript.CreateObject("Scripting.FileSystemObject")
  Set objFolder = objFS.GetFolder("C:\Windows")
  Set objSubfolders = objFolder.Subfolders
  ```

- File—This object enables you to access the properties of a specified file. To reference a File object, use either the Files collection (discussed next) or the FileSystemObject object's GetFile method:

```
Set objFS = WScript.CreateObject("Scripting.FileSystemObject")
Set objFile = objFS.GetFile("c:\autoexec.bat")
```

- Files—This object is the collection of files within a specified folder. To reference this collection, use the Folder object's Files property:

```
Set objFS = WScript.CreateObject("Scripting.FileSystemObject")
Set objFolder = objFS.GetFolder("C:\Windows")
Set objFiles = objFolder.Files
```

- TextStream—This object enables you to use sequential access to work with a text file. To open a text file, use the FileSystemObject object's OpenTextFile method:

```
Set objFS = WScript.CreateObject("Scripting.FileSystemObject")
Set objTS= objFS.OpenTextFile("C:\Boot.ini")
```

  Alternatively, you can create a new text file by using the FileSystemObject object's CreateTextFile method:

```
Set objFS = WScript.CreateObject("Scripting.FileSystemObject")
Set objTS= objFS.CreateTextFile("C:\Boot.ini")
```

  Either way, you end up with a TextStream object, which has various methods for reading data from the file and writing data to the file. For example, the following script reads and displays the text from C:\Boot.ini:

```
Set objFS = WScript.CreateObject("Scripting.FileSystemObject")
Set objTS = objFS.OpenTextFile("C:\Boot.ini")
strContents = objTS.ReadAll
WScript.Echo strContents
objTS.Close
```

Programming the WshShell Object

WshShell is a generic name for a powerful object that enables you to query and interact with various aspects of the Windows shell. You can display information to the user, run applications, create shortcuts, work with the Registry, and control Windows' environment variables. The next few sections discuss each of those useful tasks.

Referencing the WshShell Object

WshShell refers to the Shell object exposed via the Automation interface of WScript. Therefore, you must use CreateObject to return this object:

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

From here, you can use the objWshShell variable to access the object's properties and methods.

Displaying Information to the User

You saw earlier that the WScript object's Echo method is useful for displaying simple text messages to the user. You can gain more control over the displayed message by using the WshShell object's Popup method. This method is similar to the MsgBox function used in Visual Basic and VBA in that it enables you to control both the dialog box title and the buttons displayed, as well as to determine which of those buttons the user pressed. Here's the syntax:

```
WshShell.Popup(strText, [nSecondsToWait], [strTitle], [intType])
```

WshShell	The WshShell object.
strText	The message you want to display in the dialog box. You can enter a string up to 1,024 characters long.
nSecondsToWait	The maximum number of seconds the dialog box will be displayed.
strTitle	The text that appears in the dialog box title bar. If you omit this value, Windows Script Host appears in the title bar.
intType	A number or constant that specifies, among other things, the command buttons that appear in the dialog box (see the next section). The default value is 0.

For example, the following statements display the dialog box shown in Figure 12.2:

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
objWshShell.Popup "Couldn't find Memo.doc!", , "Warning"
```

TIP

For long messages, VBScript wraps the text inside the dialog box. If you prefer to create your own line breaks, use VBScript's Chr function and the carriage return character (ASCII 13) between each line:

```
WshShell.Popup "First line" & Chr(13) & "Second line"
```

For JavaScript, use \n instead:

```
WshShell.Popup("First line" + "\n" + "Second line");
```

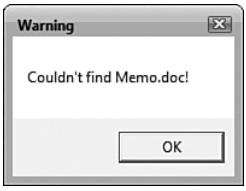

FIGURE 12.2 A simple message dialog box produced by the Popup method.

Setting the Style of the Message

The default Popup dialog box displays only an OK button. You can include other buttons and icons in the dialog box by using different values for the *intType* parameter. Table 12.1 lists the available options.

TABLE 12.1 The Popup Method's *intType* Parameter Options

VBScript Constant	Value	Description
Buttons		
vbOKOnly	0	Displays only an OK button. This is the default.
vbOKCancel	1	Displays the OK and Cancel buttons.
vbAbortRetryIgnore	2	Displays the Abort, Retry, and Ignore buttons.
vbYesNoCancel	3	Displays the Yes, No, and Cancel buttons.
vbYesNo	4	Displays the Yes and No buttons.
vbRetryCancel	5	Displays the Retry and Cancel buttons.
Icons		
vbCritical	16	Displays the Critical Message icon.
vbQuestion	32	Displays the Warning Query icon.
vbExclamation	48	Displays the Warning Message icon.
vbInformation	64	Displays the Information Message icon.
Default Buttons		
vbDefaultButton1	0	The first button is the default (that is, the button selected when the user presses Enter).
vbDefaultButton2	256	The second button is the default.
vbDefaultButton3	512	The third button is the default.

You derive the *intType* argument in one of two ways:

- By adding the values for each option
- By using the VBScript constants separated by plus signs (+)

The script in Listing 12.4 shows an example and Figure 12.3 shows the resulting dialog box.

LISTING 12.4 A VBScript Example That Uses the `Popup` Method to Display the Dialog Box Shown in Figure 12.3

```
' First, set up the message
'
strText = "Are you sure you want to copy" & Chr(13)
strText = strText & "the selected files to drive A?"
strTitle = "Copy Files"
intType = vbYesNoCancel + vbQuestion + vbDefaultButton2
'
' Now display it
'
Set objWshShell = WScript.CreateObject("WScript.Shell")
intResult = objWshShell.Popup(strText, ,strTitle, intType)
```

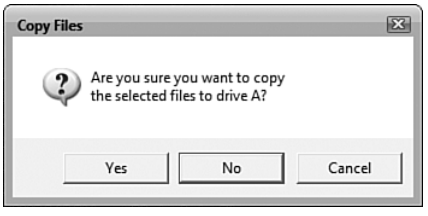

FIGURE 12.3 The dialog box that's displayed when you run the script.

Here, three variables—`strText`, `strTitle`, and `intType`—store the values for the `Popup` method's *`strText`*, *`strTitle`*, and *`intType`* arguments, respectively. In particular, the following statement derives the *`intType`* argument:

```
intType = vbYesNoCancel + vbQuestion + vbDefaultButton2
```

You also could derive the *`intType`* argument by adding up the values that these constants represent (`3`, `32`, and `256`, respectively), but the script becomes less readable that way.

Getting Return Values from the Message Dialog Box

A dialog box that displays only an OK button is straightforward. The user either clicks OK or presses Enter to remove the dialog from the screen. The multibutton styles are a little different, however; the user has a choice of buttons to select, and your script should have a way to find out which button the user chose, which enables it to decide what to do next, based on the user's selection. You do this by storing the `Popup` method's return value in a variable. Table 12.2 lists the seven possible return values.

TABLE 12.2 The Popup Method's Return Values

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

To process the return value, you can use an If...Then...Else or Select Case structure to test for the appropriate values. For example, the script shown earlier used a variable called intResult to store the return value of the Popup method. Listing 12.5 shows a revised version of the script that uses a VBScript Select Case statement to test for the three possible return values.

LISTING 12.5 A Script That Uses a Select Case Statement to Process the Popup Method's Return Value

```
' First, set up the message
'
strText = "Are you sure you want to copy" & Chr(13)
strText = strText & "the selected files to drive A?"
strTitle = "Copy Files"
intType = vbYesNoCancel + vbQuestion + vbDefaultButton2
'
' Now display it
'
Set objWshShell = WScript.CreateObject("WScript.Shell")
intResult = objWshShell.Popup(strText, ,strTitle, intType)
'
' Process the result
'
Select Case intResult
    Case vbYes
        WScript.Echo "You clicked ""Yes""!"
    Case vbNo
        WScript.Echo "You clicked ""No""!"
    Case vbCancel
        WScript.Echo "You clicked ""Cancel""!"
End Select
```

Running Applications

When you need your script to launch another application, use the `Run` method:

`WshShell``.Run` *`strCommand`*`, [`*`intWindowStyle`*`], [`*`bWaitOnReturn`*`]`

`WshShell`	The `WshShell` object.
`strCommand`	The name of the file that starts the application. Unless the file is in the Windows folder, you should include the drive and folder to make sure that the script can find the file.
`intWindowStyle`	A constant or number that specifies how the application window will appear:

`intWindowStyle`	Window Appearance
`0`	Hidden
`1`	Normal size with focus
`2`	Minimized with focus (this is the default)
`3`	Maximized with focus
`4`	Normal without focus
`6`	Minimized without focus

`bWaitOnReturn`	A logical value that determines whether the application runs asynchronously. If this value is `True`, the script halts execution until the user exits the launched application; if this value is `False`, the script continues running after it has launched the application.

Here's an example:

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
objWshShell.Run "Control.exe Inetcpl.cpl", 1, True
```

This `Run` method launches Control Panel's Internet Properties dialog box.

NOTE

To learn more about launching individual Control Panel icons using `Control.exe`, refer to "Operating Control Panel" in Chapter 10, "Using Control Panel and Group Policies."

Working with Shortcuts

The Windows Script Host enables your scripts to create and modify shortcut files. When writing scripts for other users, you might want to take advantage of this capability to display shortcuts for new network shares, Internet sites, instruction files, and so on.

Creating a Shortcut

To create a shortcut, use the `CreateShortcut` method:

```
WshShell.CreateShortcut(strPathname)
```

WshShell	The `WshShell` object.
strPathname	The full path and filename of the shortcut file you want to create. Use the `.lnk` extension for a file system (program, document, folder, and so on) shortcut; use the `.url` extension for an Internet shortcut.

The following example creates and saves a shortcut on a user's desktop:

```
Set WshShell = objWScript.CreateObject("WScript.Shell")
Set objShortcut = objWshShell.CreateShortcut("C:\Users\
➥Paul\Desktop\test.lnk")
objShortcut.Save
```

Programming the `WshShortcut` Object

The `CreateShortcut` method returns a `WshShortcut` object. You can use this object to manipulate various properties and methods associated with shortcut files.

This object contains the following properties:

- `Arguments`—Returns or sets a string that specifies the arguments used when launching the shortcut. For example, suppose that the shortcut's target is the following:

  ```
  C:\Windows\Notepad.exe C:\Boot.ini
  ```

  In other words, this shortcut launches Notepad and loads the `Boot.ini` file. In this case, the `Arguments` property would return the following string:

  ```
  C:\Boot.ini
  ```

- `Description`—Returns or sets a string description of the shortcut.
- `FullName`—Returns the full path and filename of the shortcut's target. This will be the same as the *strPathname* value used in the `CreateShortcut` method.
- `Hotkey`—Returns or sets the hotkey associated with the shortcut. To set this value, use the following syntax:

  ```
  WshShortcut.Hotkey = strHotKey
  ```

WshShortcut	The `WshShortcut` object.
strHotKey	A string value of the form *Modifier+Keyname*, where *Modifier* is any combination of Alt, Ctrl, and Shift, and *Keyname* is one of A through Z or 0 through 12.

For example, the following statement sets the hotkey to Ctrl+Alt+7:

```
objShortcut.Hotkey = "Ctrl+Alt+7"
```

- `IconLocation`—Returns or sets the icon used to display the shortcut. To set this value, use the following syntax:

  `WshShortcut``.IconLocation = `*`strIconLocation`*

`WshShortcut`	The `WshShortcut` object.
`strIconLocation`	A string value of the form *`Path`*`,`*`Index`*, where *`Path`* is the full pathname of the icon file and *`Index`* is the position of the icon within the file (where the first icon is `0`).

  Here's an example:

  ```
  objShortcut.IconLocation = "C:\Windows\System32\Shell32.dll,21"
  ```

- `TargetPath`—Returns or sets the path of the shortcut's target.
- `WindowStyle`—Returns or sets the window style used by the shortcut's target. Use the same values outlined earlier for the `Run` method's *`intWindowStyle`* argument.
- `WorkingDirectory`—Returns or sets the path of the shortcut's working directory.

NOTE

If you're working with Internet shortcuts, bear in mind that they support only two properties: `FullName` and `TargetPath` (the URL target).

The `WshShortcut` object also supports two methods:

- `Save`—Saves the shortcut file to disk.
- `Resolve`—Uses the shortcut's `TargetPath` property to look up the target file. Here's the syntax:

  ```
  WshShortcut.Resolve = intFlag
  ```

`WshShortcut`	The `WshShortcut` object.
`intFlag`	Determines what happens of the target file is not found:

`intFlag`	**What Happens**
1	Nothing
2	Windows continues to search subfolders for the target file
4	Updates the `TargetPath` property if the target file is found in a new location

Listing 12.6 shows a complete example of a script that creates a shortcut.

Listing 12.6 A Script That Creates a Shortcut File

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
Set objShortcut = objWshShell.CreateShortcut("C:\Users\Paul
➥\Desktop\Edit BOOT.INI.lnk")
With objShortcut
    .TargetPath = "C:\Windows\Notepad.exe "
    .Arguments = "C:\Boot.ini"
    .WorkingDirectory = "C:\"
    .Description = "Opens BOOT.INI in Notepad"
    .Hotkey = "Ctrl+Alt+7"
    .IconLocation = "C:\Windows\System32\Shell32.dll,21"
    .WindowStyle = 3
    .Save
End With
```


Working with Registry Entries

You've seen throughout this book that the Registry is one the most crucial data structures in Windows. However, the Registry isn't a tool that only Windows yields. Most 32-bit applications make use of the Registry as a place to store setup options, customization values the user selected, and much more. Interestingly, your scripts can get in on the act as well. Not only can your scripts read the current value of any Registry setting, but they can also use the Registry as a storage area. This enables you to keep track of user settings, recently used files, and any other configuration data that you'd like to save between sessions. This section shows you how to use the `WshShell` object to manipulate the Registry from within your scripts.

Reading Settings from the Registry

To read any value from the Registry, use the WshShell object's `RegRead` method:

```
WshShell.RegRead(strName)
```

WshShell — The `WshShell` object.

strName — The name of the Registry value or key that you want to read. If *strName* ends with a backslash (\), `RegRead` returns the default value for the key; otherwise, `RegRead` returns the data stored in the value. Note, too, that *strName* must begin with one of the following root key names:

Short Name	Long Name
HKCR	HKEY_CLASSES_ROOT
HKCU	HKEY_CURRENT_USER

Short Name	Long Name
HKLM	HKEY_LOCAL_MACHINE
N/A	HKEY_USERS
N/A	HKEY_CURRENT_CONFIG

The script in Listing 12.7 displays the name of the registered owner of this copy of Windows XP.

LISTING 12.7 A Script That Reads the RegisteredOwner Setting from the Registry

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
strSetting = "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\RegisteredOwner"
strRegisteredUser = objWshShell.RegRead(strSetting)
WScript.Echo strRegisteredUser
```

Storing Settings in the Registry

To store a setting in the Registry, use the WshShell object's RegWrite method:

```
WshShell.RegWrite strName, anyValue [, strType]
```

WshShell	The WshShell object.
strName	The name of the Registry value or key that you want to set. If *strName* ends with a backslash (\), RegWrite sets the default value for the key; otherwise, RegWrite sets the data for the value. *strName* must begin with one of the root key names detailed in the RegRead method.
anyValue	The value to be stored.
strType	The data type of the value, which must be one of the following: REG_SZ (the default), REG_EXPAND_SZ, REG_DWORD, or REG_BINARY.

The following statements create a new key named ScriptSettings in the HKEY_CURRENT_USER root:

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
objWshShell.RegWrite "HKCU\ScriptSettings\", ""
```

The following statements create a new value named NumberOfReboots in the HKEY_CURRENT_USER\ScriptSettings key, and set this value to 1:

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
objWshShell.RegWrite "HKCU\ScriptSettings\NumberOfReboots", 1, "REG_DWORD"
```

Deleting Settings from the Registry

If you no longer need to track a particular key or value setting, use the `RegDelete` method to remove the setting from the Registry:

```
WshShell.RegDelete(strName)
```

WshShell	The `WshShell` object.
strName	The name of the Registry value or key that you want to delete. If *strName* ends with a backslash (`\`), `RegDelete` deletes the key; otherwise, `RegDelete` deletes the value. *strName* must begin with one of the root key names detailed in the `RegRead` method.

To delete the `NumberOfReboots` value used in the previous example, you would use the following statements:

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
objWshShell.RegDelete "HKCU\ScriptSettings\NumberOfReboots"
```

Working with Environment Variables

Windows Vista keeps track of a number of **environment variables** that hold data such as the location of the Windows folder, the location of the temporary files folder, the command path, the primary drive, and much more. Why would you need such data? One example would be for accessing files or folders within the main Windows folder. Rather than guessing that this folder is `C:\Windows`, it would be much easier to just query the `%SystemRoot%` environment variable. Similarly, if you have a script that accesses files in a user's My Documents folder, hard-coding the username in the file path is inconvenient because it means creating custom scripts for every possible user. Instead, it would be much easier to create just a single script that references the `%UserProfile%` environment variable. This section shows you how to read environment variable data within your scripts.

The defined environment variables are stored in the `Environment` collection, which is a property of the `WshShell` object. Windows Vista environment variables are stored in the `"Process"` environment, so you reference this collection as follows:

```
WshShell.Environment("Process")
```

Listing 12.8 shows a script that runs through this collection, adds each variable to a string, and then displays the string.

LISTING 12.8 A Script That Displays the System's Environment Variables

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
'
' Run through the environment variables
'
strVariables = ""
For Each objEnvVar In objWshShell.Environment("Process")
    strVariables = strVariables & objEnvVar & Chr(13)
Next
WScript.Echo strVariables
```

Figure 12.4 shows the dialog box that appears (your mileage may vary).

FIGURE 12.4 A complete inventory of a system's environment variables.

If you want to use the value of a particular environment variable, use the following syntax:

```
WshShell.Environment("Process")("strName")
```

WshShell	The `WshShell` object
strName	The name of the environment variable

Listing 12.9 shows a revised version of the script from Listing 12.6 to create a shortcut. In this version, the `Environment` collection is used to return the value of the `%UserProfile%` variable, which is used to contrast the path to the current user's Desktop folder.

LISTING 12.9 A Script That Creates a Shortcut File Using an Environment Variable

```
Set objWshShell = WScript.CreateObject("WScript.Shell")
strUserProfile = objWshShell.Environment("Process")("UserProfile")
Set objShortcut = objWshShell.CreateShortcut(strUserProfile & _
                  "\Desktop\Edit BOOT.INI.lnk")
With objShortcut
    .TargetPath = "C:\Windows\Notepad.exe "
    .Arguments = "C:\Boot.ini"
    .WorkingDirectory = "C:\"
    .Description = "Opens BOOT.INI in Notepad"
    .Hotkey = "Ctrl+Alt+7"
    .IconLocation = "C:\Windows\System32\Shell32.dll,21"
    .WindowStyle = 3
    .Save
End With
```


Programming the `WshNetwork` Object

`WshNetwork` is a generic name for an object that enables you to work with various aspects of the Windows network environment. You can determine the computer name and username, you can enumerate the mapped network drives, you can map new network drives, and more. The next couple of sections show you how to work with this object.

Referencing the `WshNetwork` Object

`WshNetwork` refers to the `Network` object exposed via the Automation interface of WScript. This means you use `CreateObject` to return this object, as shown here:

```
Set objWshNetwork = WScript.CreateObject("WScript.Network")
```

From here, you use the `WshNetwork` variable to access the object's properties and methods.

`WshNetwork` Object Properties

The `WshNetwork` object supports three properties:

`ComputerName`	Returns the network name of the computer
`UserDomain`	Returns the network domain name of the current user
`UserName`	Returns the username of the current user

Mapping Network Printers

The WshNetwork object supports several methods for working with remote printers. For example, to map a network printer to a local printer resource, use the WshNetwork object's AddWindowsPrinterConnection method:

```
WshNetwork.AddPrinterConnection strPrinterPath
```

WshNetwork	The WshNetwork object
strPrinterPath	The UNC path to the network printer

Here's an example:

```
Set objWshNetwork = WScript.CreateObject("WScript.Network")
objWshNetwork.AddWindowsPrinterConnection "\\ZEUS\printer"
```

To remove a remote printer mapping, use the WshNetwork object's RemovePrinterConnection method:

```
WshNetwork.RemovePrinterConnection strPrinterPath [, bForce] [, bUpdateProfile]
```

WshNetwork	The WshNetwork object
strPrinterPath	The UNC path to the network printer
bForce	If True, the resource is removed even if it is currently being used
bUpdateProfile	If True, the printer mapping is removed from the user's profile

Here's an example:

```
Set objWshNetwork = WScript.CreateObject("WScript.Network")
objWshNetwork.RemovePrinterConnection "\\ZEUS\inkjet"
```

Mapping Network Drives

The WshNetwork object supports several methods for mapping network drives. To map a shared network folder to a local drive letter, use the WshNetwork object's MapNetworkDrive method:

```
WshNetwork.MapNetworkDrive strLocalName, strRemoteName,
➥[bUpdateProfile], [strUser], [strPassword]
```

WshNetwork	The WshNetwork object
strLocalName	The local drive letter to which the remote share will be mapped (for example, F:)
strRemoteName	The UNC path for the remote share

bUpdateProfile	If `True`, the drive mapping is stored in the user's profile
strUser	Use this value to enter a username that might be required to map the remote share (if you're logged on as a user who doesn't have the proper permissions, for example)
strPassword	Use this value to enter a password that might be required to map the remote drive

Here's an example:

```
Set objWshNetwork = WScript.CreateObject("WScript.Network")
objWshNetwork.MapNetworkDrive "Z:", "\\ZEUS\SharedDocs"
```

To remove a mapped network drive, use the `WshNetwork` object's `RemoveNetworkDrive`:

```
WshNetwork.RemoveNetworkDrive strName, [bForce], [bUpdateProfile]
```

WshNetwork	The `WshNetwork` object.
strName	The name of the mapped network drive you want removed. If you use a network path, all mappings to that path are removed; if you use a local drive letter, only that mapping is removed.
bForce	If `True`, the resource is removed even if it is currently being used.
bUpdateProfile	If `True`, the network drive mapping is removed from the user's profile.

Here's an example:

```
Set objWshNetwork = WScript.CreateObject("WScript.Network")
objWshNetwork.RemoveNetworkDrive "Z:"
```

Example: Scripting Internet Explorer

To give you a taste of the power and flexibility of scripting—particularly Automation programming—I'll close this chapter by showing you how to program a specific Automation server: Internet Explorer. You'll see that your scripts can control just about everything associated with Internet Explorer:

- The position and dimensions of the window
- Whether the menu bar, toolbar, and status bar are displayed
- The current URL
- Sending the browser backward and forward between navigated URLs

Displaying a Web Page

To get started, I'll show you how to use the `InternetExplorer` object to display a specified URL. You use the `Navigate` method to do this, and this method uses the following syntax:

```
InternetExplorer.Navigate URL [, Flags,] [ TargetFramename] [, PostData]
[ ,Headers]
```

InternetExplorer	A reference to the `InternetExplorer` object with which you're working.
URL	The address of the web page you want to display.
Flags	One of (or the sum of two or more of) the following integers that control various aspects of the navigation: 1 Opens the *URL* in a new window 2 Prevents the *URL* from being added to the history list 4 Prevents the browser from reading the page from the disk cache 8 Prevents the *URL* from being added to the disk cache
TargetFrameName	The name of the frame in which to display the *URL*.
PostData	Specifies additional `POST` information that HTTP requires to resolve the hyperlink. The most common uses for this argument are to send a web server the contents of a form, the coordinates of an imagemap, or a search parameter for an ASP file. If you leave this argument blank, this method issues a `GET` call.
Headers	Specifies header data for the HTTP header.

Here's an example:

```
Set objIE = CreateObject("InternetExplorer.Application")
objIE.Navigate "http://www.microsoft.com/"
```

Navigating Pages

Displaying a specified web page isn't the only thing the `InternetExplorer` object can do. It also has quite a few methods that give you the ability to navigate backward and forward through visited web pages, refresh the current page, stop the current download, and more. Here's a summary of these methods:

`GoBack`	Navigates backward to a previously visited page
`GoForward`	Navigates forward to a previously visited page
`GoHome`	Navigates to Internet Explorer's default Home page

`GoSearch`	Navigates to Internet Explorer's default Search page
`Refresh`	Refreshes the current page
`Refresh2`	Refreshes the current page using the following syntax: `Refresh2(`*`Level`*`)`
	`Level` — A constant that determines how the page is refreshed:
	`0` — Refreshes the page with a cached copy
	`1` — Refreshes the page with a cached copy only if the page has expired
	`3` — Performs a full refresh (doesn't use a cached copy)
`Stop`	Cancels the current download or shuts down dynamic page objects, such as background sounds and animations.

Using the `InternetExplorer` Object's Properties

Here's a summary of many of the properties associated with the `InternetExplorer` object:

`Busy`	Returns `True` if the `InternetExplorer` object is in the process of downloading text or graphics. This property returns `False` when a download of the complete document has finished.
`FullScreen`	A Boolean value that toggles Internet Explorer between the normal window and a full-screen window in which the title bar, menu bar, toolbar, and status bar are hidden.
`Height`	Returns or sets the window height.
`Left`	Returns or sets the position of the left edge of the window.
`LocationName`	Returns the title of the current document.
`LocationURL`	Returns the URL of the current document.
`MenuBar`	A Boolean value that toggles the menu bar on and off.
`StatusBar`	A Boolean value that toggles the status bar on and off.
`StatusText`	Returns or sets the status bar text.
`ToolBar`	A Boolean value that toggles the toolbar on and off.
`Top`	Returns or sets the position of the top edge of the window.
`Type`	Returns the type of document currently loaded in the browser.
`Visible`	A Boolean value that toggles the object between hidden and visible.
`Width`	Returns or sets the window width.

Running Through a Sample Script

To put some of the properties and methods into practice, Listing 12.10 shows a sample script.

LISTING 12.10 A Script That Puts the `InternetExplorer` Object Through Its Paces

```
Option Explicit
Dim objIE, objWshShell, strMessage, intResult

' Set up the Automation objects
Set objIE = WScript.CreateObject("InternetExplorer.Application")
Set objWshShell = WScript.CreateObject("WScript.Shell")

' Navigate to a page and customize the browser window
objIE.Navigate "http://www.wordspy.com/"
objIE.Toolbar = False
objIE.StatusBar = False
objIE.MenuBar = False

' Twiddle thumbs while the page loads
Do While objIE.Busy
Loop

' Get the page info
strMessage = "Current URL:  " & objIE.LocationURL & vbCrLf & _
    "Current Title: " & objIE.LocationName & vbCrLf & _
    "Document Type: " & objIE.Type & vbCrLf & vbCrLf & _
    "Would you like to view this document?"

' Display the info
intResult = objWshShell.Popup(strMessage, , "Scripting IE", vbYesNo + vbQuestion)

' Check the result
If intResult = vbYes Then

    ' If Yes, make browser visible
    objIE.Visible = True
Else

    ' If no, bail out
    objIE.Quit
End If
Set objIE = Nothing
Set objWshShell = Nothing
```

The script begins by creating instances of the `InternetExplorer` and WScript `Shell` objects. The `Navigate` method displays a page, and then turns off the toolbar, status bar, and menu bar. A `Do...Loop` checks the `Busy` property and loops while it's `True`. In other words, this loop won't exit until the page is fully loaded. A string variable is used to store the URL, the title, and type of the page, and this string is then displayed in a `Popup` box, which also asks whether the user wants to see the page. If the user clicks the Yes button, the browser is made visible; if the user clicks the No button, the `Quit` method shuts down the browser.

From Here

Here are some sections of the book that contain information related to the scripting techniques you learned in this chapter:

- To learn how to run scripts when you start your Windows XP system, see the section "Specifying Startup and Logon Scripts" in Chapter 5, "Installing and Running Applications."
- To learn more about the Registry, see Chapter 11, "Getting to Know the Windows XP Registry."
- I show you a script that displays the available free space on all your drives in the "Checking Free Disk Space" section of Chapter 15, "Maintaining Your Windows XP System."
- For some examples of security-related scripts, see Chapter 21, "Implementing Windows XP's Internet Security and Privacy Features."
- You can also "program" Windows XP using batch files. See Appendix C, "Automating Windows XP with Batch Files."

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z