Part I

Getting Started with Windows Scripting

Part I of the *PowerShell, VBScript, and JScript Bible* introduces you to the powerful administrative tool that is Windows scripting. You'll get an overview of Windows scripting and its potential, and an introduction to three technologies you can use for Windows scripting: VBScript, JScript, and PowerShell.

IN THIS PART

Chapter 1

Introducing Windows Scripting

Windows scripting gives everyday users and administrators the ability to automate repetitive tasks, complete activities while away from the computer, and perform many other time-saving activities. Windows scripting accomplishes all of this by enabling you to create tools to automate tasks that would otherwise be handled manually, such as creating user accounts, generating log files, managing print queues, or examining system information. By eliminating manual processes, you can double, triple, or even quadruple your productivity and become more effective and efficient at your job. Best of all, scripts are easy to create and you can rapidly develop prototypes of applications, procedures, and utilities; and then enhance these prototypes to get exactly what you need, or just throw them away and begin again. This ease of use gives you the flexibility to create the kinds of tools you need without a lot of fuss.

IN THIS CHAPTER

Introducing Windows Scripting

You've heard the claims about scripting and now you're thinking, so what? What's in it for me? You may be an administrator rather than a developer. Or maybe you're a power user who helps other users from time to time. Either way, scripting will prove useful to your situation and needs. So in answer to the question, "What's in it for me?" consider the following:

- **Would you like to have more free time?** Windows scripting frees you from mundane and repetitive tasks, enabling you to focus on more interesting and challenging tasks.
- **Would you like to be able to analyze trends and be proactive rather than reactive?** You can use Windows scripting to extract and manipulate huge quantities of information and turn out easy-to-use reports.
- **Would you like to be able to seize opportunities before they disappear?** Windows scripting enables you to take advantage of opportunities and be more effective. You can solve problems quickly and efficiently.
- **Would you like to be a top performer and receive the praise you deserve?** Windows scripting enables you to accomplish in hours or days what would otherwise take weeks or months with traditional techniques. You'll be more successful and more productive at work.
- **Would you like to be able to integrate activities and applications?** Windows scripting enables you to integrate information from existing systems and applications, allowing you to kick off a series of tasks simply by starting a script.
- **Would you like to have fun at work?** Windows scripting can be fun, challenging, and rewarding. Give it a try and you'll see!

If Windows scripting can do so much, it must be terribly complex, right? On the contrary—it is its simplicity that enables you to do so much, not complexity. Many Windows scripts are only a few lines long and you can create them in a few minutes!

Taking a look at Windows Scripting

Two different architectures are used for scripting in Windows. The older one uses the Windows Script Host and the newer one uses PowerShell. A lot of the tasks that can be carried out using the VBScript in the Windows Scripting Host can be transferred to PowerShell. However not all the tasks that can be run in PowerShell can be transferred to Windows Script Host scripts so easily. For a lot of organizations using various derivatives of Visual Basic—in Web pages, Office applications, Windows forms applications—makes a de-facto standard.

Windows Script Host Architecture

Windows Script Host (WSH) has been part of Windows since Windows NT4. Windows Script Host provides architecture for building dynamic scripts that consist of a core object model, scripting hosts, and scripting engines—each of which is discussed in the sections that follow.

Getting Started with Windows Script Host

Windows Script Host is a core component of the Windows operating system and, as such, is installed by default when you install Windows. Like other components, Windows Script Host can be uninstalled. It can also be upgraded through downloads or by installing service packs. To

ensure that Windows Script Host is installed on your system, type **cscript** at a command prompt. You should see version information for Windows Script Host as well as usage details. If you don't see this information, Windows Script Host may not be installed and you'll need to install it as you would any other Windows component.

The key components of Windows Script Host are as follows:

- **WScript:** A Windows executable for the scripting host that is used when you execute scripts from the desktop. This executable has GUI controls for displaying output in pop-up dialog boxes.
- **CScript:** A command-line executable for the scripting host that is used when you execute scripts from the command line. This executable displays standard output at the command line.
- **WSH ActiveX Control:** An ActiveX control that provides the core object model for the scripting host.
- **Scripting Engines:** Scripting engines provide the core functions, objects, and methods for a particular scripting language. VBScript and JScript scripting engines are installed by default on Windows.

A Windows script is a text file containing a series of commands. Unlike shell scripts, Windows script commands don't resemble commands that you'd type in at the keyboard. Instead, they follow the syntax for the scripting language you are using, such as VBScript or JScript.

Windows scripts can be created in Notepad. When you finish creating the script, save it with an extension appropriate for the scripting language (.vbs for VBScript, .js for JScript, or .wsf for batch scripts that combine scripts with markup). Once you create a Windows script, you run it with WScript or CScript.

Using and running scripts

Windows scripts can be run with either WScript or CScript, and most of the time the application you use depends on your personal preference. However, you'll find that WScript works best for scripts that interact with users, especially if the script displays results as standard text output. For tasks that you want to automate or run behind the scenes, you'll probably prefer CScript, with which you can suppress output and prompts for batch processing.

You can use WScript and CScript with scripts in several different ways. The easiest way is to set WScript as the default application for scripts and then run scripts by clicking their file name in Windows Explorer. Don't worry—you don't have to do anything fancy to set WScript as the default. The first time you click a Windows script, you'll be asked if you'd like to associate the file extension with WScript. Click Yes. Alternatively, you may see an Open With dialog box that asks which program you would like to use to open the file. Choose WScript, and then check the "Always use this program to open this file" checkbox.

You can also set CScript as the default interface. When you do this, clicking a Windows script runs CScript instead of WScript. Or, you could run scripts from the Run prompt just as you could when WScript was the default. To run scripts with CScript from the command line, enter **cscript** followed by the pathname of the script you want to execute. For now, don't worry about the details; you'll find detailed instructions in Chapter 4.

Core object model

The core object model and scripting hosts are packaged with WSH for Windows. The core object model is implemented in the WSH.ocx ActiveX control. WSH.ocx provides the key functionality necessary for scripts to interact with the operating system. In WSH, objects are simply named containers that you'll use to interact with operating system components. For example, you'll use the `WshNetwork` object to access and configure network resources, such as printers and drives.

Each object has properties and methods that are used to perform certain types of tasks. Properties are attributes of an object that you can access. Methods are procedures that you'll use to perform operations. As with other object-based programming languages, you can work with objects in a variety of ways. You can use built-in objects, create new objects based on the built-in objects, or define your own objects using unique methods and properties.

Table 1-1 provides a summary of the WSH object model. The WSH object hierarchy can be broken down into two broad categories: exposed objects and non-exposed objects. Exposed objects, such as `WScript`, are the ones you'll work with in your scripts. Non-exposed objects, such as `WshCollection`, are accessed through the methods or properties of other objects. These objects do the behind-the-scenes work.

TABLE 1-1

Core WSH Objects

Object Type	Object	Description
Exposed Object	`Script.Signer`	An object that allows you to sign scripts with a digital signature and to verify signed scripts
	`WScript`	Top-level object that provides access to core objects and other functionality such as object creation
	`WScript.WshNetwork`	Automation object used to access and configure network resources, such as printers and drives, also provides user, domain, and computer information
	`WScript.WshShell`	Automation object that provides access to the environment and file folders

Object Type	Object	Description
	WshController	Automation object that provides the control functions necessary for creating a remote script process
Non-exposed Object	WshArguments	Accessed through the WScript.Arguments property, obtains command-line arguments
	WshCollection	Accessed through WshNetwork.EnumNetwork Drives or WshNetwork.EnumPrinter Collection, used for iteration through a group of items, such as printers or drives
	WshEnvironment	Accessed through the WshShell.Environment property, allows you to work with environment variables
	WshNamed	Accessed through the WScript.Arguments .Named property, allows you to work with named arguments passed to a script
	WshRemote	Accessed through the WshController .WshRemote method, allows you to start, stop, and track the status of remote scripts
	WshRemote.Error	Accessed through the WshRemote.Error property, used to track runtime errors related to remote scripts
	WshScriptExec	Accessed through the WshShell.Exec method, allows you to track the status of program or scripts started with the WshShell.Exec method, also provides access to the related input, output, and error streams
	WshShortcut	Accessed through the WshShell.CreateShortcut method, used to create and manage file shortcuts
	WshSpecialFolders	Accessed through the WshShell.SpecialFolders property, used to work with file folders
	WshUnnamed	Accessed through the WScript.Arguments .Unnamed property, allows you to work with unnamed arguments passed to a script
	WshUrlShortcut	Accessed through the WshShell.CreateShortcut method, used to create and manage URL shortcuts

NOTE **With the JScript scripting engine, the letter case for object, method, and property names is important. The JScript engine doesn't recognize an object unless you reference it properly. For example, with `WScript`, the JScript engine does not recognize `Wscript`. Because VBScript really doesn't care about letter case, either `Wscript` or `WScript` works just fine.**

More on scripting hosts

To execute Windows scripts, you'll use one of the two scripting hosts available, either WScript or CScript. WScript has GUI controls for displaying output in pop-up dialog boxes and is used primarily when you execute scripts from the desktop. CScript is the command-line executable for the scripting host that is used when you execute scripts from the command line. Although you can work with both of these hosts in much the same way, there are some features specific to each, which we discuss later in Chapter 4. For now, let's focus on how the scripting hosts work.

Several file extensions are mapped for use with the scripting hosts. These file extensions are:

- .js: Designates scripts written in JScript
- .vbs: Designates scripts written in VBScript
- .wsf: Designates a Windows script file
- .wsh: Designates a WSH properties file

A limitation of .js and .vbs files is that they can contain only JScript or VBScript statements, respectively, and you cannot mix and match. This is where .wsf files come into the picture. You can use .wsf files to create WSH jobs, or what I call ***batch*** scripts. These batch scripts can combine multiple types of scripts and can also include type libraries containing constants.

Batch scripts contain markup tags that identify elements within the batch, such as individual jobs and the scripting language being used. These markup tags are defined as XML (Extensible Markup Language) elements. XML is structured much like HTML and uses plain-text characters. You can use any text editor to create batch scripts and, because batch scripts contain XML, you can also use an XML editor.

Windows scripts can also use .wsh files. These files contain default settings for scripts, such as timeout values and script paths. Because of the introduction of .wsf files and direct in-script support for most script properties, .wsh files are rarely needed.

More on scripting engines

Scripting engines provide the core language functionality for Windows scripts and are packaged separately from the Windows Script Host itself. You can obtain scripting engines for JScript, VBScript, Perl, TCL, Python, and more. The official Microsoft scripting engines for VBScript and JScript are standard components on Windows and are the focus of this book.

With Windows scripting, many of the features available for scripting with Internet Explorer and the Web aren't available. Functions needed for Web scripting simply aren't needed for Windows

scripting and vice versa. For example, in JScript, none of the window-related objects are available in WSH because, in Windows, you normally don't need to access documents, forms, frames, applets, plug-ins, or any of those other browser-related features. The exception to this is if you create a script that starts a browser session; within the browser session, you can use the browser-related objects as much as you want.

Right now, you may be wondering what exactly is and isn't supported by Windows scripts. In a nutshell, the scripting engines support core language and language runtime environments. The core language includes operators, statements, built-in objects, and built-in functions. Operators are used to perform arithmetic, comparisons, and more. Statements are used to make assignments, to conditionally execute code, and to control the flow within a script. For example, you can use `for` looping to execute a section of code for a specific count. These types of statements are all defined in the core language. Beyond this, the core language also defines the core functions and objects that perform common operations such as evaluating expressions, manipulating strings, and managing data.

The runtime environment adds objects to the core object model. These objects are used to work with the operating system and are available only with Windows Scripting. Table 1-2 provides a complete list of the available VBScript objects. The list is organized according to where the objects originate, either in the runtime environment or the core object model.

TABLE 1-2

VBScript Objects for Windows Scripting

Runtime Objects	Core Objects
`Dictionary` object	`Class` object
`Drive` object	`Debug` object
`Drives` collection	`Dictionary` object
`File` object	`Err` object
`Files` collection	`FileSystemObject` object
`FileSystemObject` object	`Match` object
`Folder` object	`Matches` collection
`Folders` collection	`RegExp` object
`TextStream` object	`SubMatches` collection

Table 1-3 provides a complete list of available JScript objects. Again, the list is organized according to where the objects originate.

TABLE 1-3

JScript Objects for Windows Scripting

Runtime Objects	Core Objects
`Arguments` Object	`ActiveXObject` Object
`Dictionary` object	`Array` object
`Drive` object	`Boolean` object
`Drives` collection	`Date` object
`File` object	`Debug` object
`Files` collection	`Dictionary` object
`FileSystemObject` object	`Enumerator` object
`Folder` object	`Error` object
`Folders` collection	`FileSystemObject` object
`TextStream` object	`Function` object
	`Global` object
	`Math` object
	`Number` object
	`Object` object
	`RegExp` object
	`Regular Expression` object
	`String` object
	`VBArray` object

Windows PowerShell Architecture

The name "PowerShell" explains the key architectural difference from the Windows Scripting Host. PowerShell began life as a command-line shell—like Windows CMD.EXE, and you can interact with it—so where VBScript or JScript programs are written in Notepad and run using the appropriate language inside the scripting host, the lines of a PowerShell script might be tested at a command prompt one by one and then gathered into a script.

As a shell, PowerShell can chain commands together using piping—that is, sending the output of one command into another using the | symbol. Often, development consists of running a command, checking its output, piping that output into something, checking that, and building up a long and complex line.

One of the important things that sets PowerShell apart from CMD.EXE is that where a command returns text to CMD, PowerShell's commands return objects. The properties and methods of those objects can be used by commands further along a pipeline.

Compared with the WSH languages, PowerShell's use of objects is both broader and deeper. Its use is deeper because .NET defines types such as text strings, and provides methods for working with them. PowerShell does not need to write a function for getting a substring from a bigger string—that's inherited from .NET, as is PowerShell's file handling, arithmetic, and so on (so PowerShell doesn't need to implement the core functions found in the WSH languages). PowerShell's use of objects is broader, because PowerShell has access to .NET objects, as well as COM ones and ready-made commands for getting to WMI and Active Directory objects.

WMI objects provide management, configuration, and performance information for many server applications and Windows components—indeed you could do a lot with just piping the output of PowerShell's `Get-WMIObject` command into its `Format-Table` command.

PowerShell was designed to be highly extensible. Not only can your own scripts become part of the working environment, but also developers can write ***snap-ins*** that extend the environment with compiled code. These add to the set of commands available inside PowerShell—the term "command" in PowerShell covers all the different things that can be invoked from the prompt: external programs, scripts, functions loaded from scripts, and what PowerShell terms "CMDlets" from the snap-ins. PowerShell provides five snap-ins by default.

TABLE 1-4

PowerShell Snap-ins

Snap-in	Functions
`Core`	Loads other snap-ins, provides access to command history, implements `for` loop, and `where` functionality
`Host`	Handles the console, manages transcripts
`Management`	Provides the commands to manage Windows components
`Security`	Handles credentials and secure strings
`Utility`	Provides the commands to format and output data

Other products that run on Windows can provide their own snap-ins—for example, Exchange2007, SQL Server 2008, and various members of the system center family provide their own snap-ins to

allow PowerShell to be used as the scripting environment to manage them. At the time of this writing, Windows Server 2008 R2 has only just been announced: It will include an updated version of PowerShell, and more Windows components will have snap-ins to manage them.

The CMDLets snap-ins can also implement *providers*. The Security snap-in loads a provider for the Certificate store, so you can browse through it as if it were a file system. The Core snap-in has one for the registry, so you can treat branches of the registry like folders on your hard disk. Again, additional snap-ins can add to the list of providers.

Although PowerShell is a shell, it is possible to use the engine from another program without loading the "host"—the command Window that is wrapped around the engine. Increasingly it is expected that management tools for Microsoft products will be written as PowerShell snap-ins and then the GUI management tools will invoke CMDlets in these. This allows you to carry out a task in the GUI, discover the script that would carry it out, and use that as the basis for your own scripts.

PowerShell scripts have a .ps1 file extension, but to avoid the dangers of PowerShell automatically running a malicious script, the file type is not tied to the PowerShell executable. You can run PowerShell.exe with a command line that is the name of a script. Or you can invoke the script inside the shell. There is no equivalent to the choice between CScript and WScript.

Is there any need to learn anything other than PowerShell? That's less of a point of argument between the contributors of this book than you might imagine. It's going to become harder to be a properly rounded IT professional in a Microsoft environment without PowerShell, but the other languages will be with us for many years. Few organizations will see sense in re-writing a perfectly good VB or JScript script as a PowerShell one, and there are libraries and code samples that exist only in those languages. Sometimes it will make sense to translate them into PowerShell (which requires the ability to understand the script) and sometimes it will make sense to adapt an existing script in its existing language.

Summary

Now that you have a taste of what Windows scripting is all about, it's time to go to the next level. Chapters 2, 3, and 4 provide essential scripting techniques for VBScript, JScript, and PowerShell, respectively. Carefully study these chapters before proceeding as they describe the core mechanics of scripting, covering variables, arrays, operators, conditional statements, control loops, procedures, and more. Once we have covered these core mechanics, we won't waste your time rehashing how these features work with every future scripting example. Instead, we will trust that you've reviewed and understand the core mechanics and want to focus on the new materials we are discussing in a particular chapter. Even if you know some scripting basics, we recommend that you use these chapters to brush up on your VBScript, JScript, and PowerShell knowledge.