1

Introduction to PowerShell

Welcome to Windows PowerShell, the new object-based command-line interface shell and scripting language built on top of .NET. PowerShell provides improved control and automation of IT administration tasks for the Windows platform. It is designed to make IT professionals and developers more productive.

Several books that introduce end-user IT professionals to Windows PowerShell are already available, but PowerShell development from the perspective of cmdlet, provider, and host developers has gone largely unmentioned. This book attempts to fill that gap by introducing the reader to the concepts, components, and development techniques behind building software packages that leverage Windows PowerShell. This book is written for developers who want to extend the functionality of Windows PowerShell and extend their applications using PowerShell.

Traditionally, when developers write a command-line utility, they have to write code for parsing the parameters, binding the argument values to parameters during runtime. In addition, they have to write code for formatting the output generated by the command. Windows PowerShell makes that easy by providing a runtime engine with its own parser. It also provides functionality that enables developers to add custom formatting when their objects are displayed. By performing the common tasks associated with writing command-line utilities, Windows PowerShell enables developers to focus on the business logic of their application, rather than spend development time solving universal problems.

Windows PowerShell Design Principles

Windows PowerShell was designed in response to years of customer feedback about the administrative experience on Microsoft Windows. Early on, many users asked why some of the traditional Unix shells weren't licensed and included in Windows, and it became apparent that the right answer was to produce a whole new kind of shell that would leave these legacy technologies behind. This thinking was distilled into four guiding principles that provided the foundation for PowerShell's design effort.

Preserve the Customer's Existing Investment

When a new technology is rolled out, it takes time for the technology to be adopted. Moreover, customers are likely to have already invested a lot in existing technologies. It's unreasonable to expect people to throw out their existing investments, which is why PowerShell was designed from the ground up to be compatible with existing Windows Management technologies.

In fact, PowerShell runs existing commands and scripts seamlessly. You can make use of PowerShell's integration with COM, WMI, and ADSI technologies alongside its tight integration with .NET. Indeed, PowerShell is the only technology that enables you to create and work with objects from these various technologies in one environment. You can see examples of this and other design principles in a quick tour of PowerShell later in the chapter.

Provide a Powerful, Object-Oriented Shell

`CMD.exe` and other traditional shells are text-based, meaning that the commands in these shells take text as input and produce text as output. Even if these commands convert the text internally into objects, when they produce output they convert it back to text. In traditional shells, when you want to put together simple commands in the pipeline, a lot of text processing is done between commands to produce desired output. Tools such as SED, AWK, and Perl became popular among command-line scripters because of their powerful text-processing capabilities.

PowerShell is built on top of .NET and is an object-based shell and scripting language. When you pipe commands, PowerShell passes objects between commands in the pipeline. This enables objects to be manipulated directly and to be passed to other tools. PowerShell's tight integration with .NET brings the functionality and consistency of .NET to IT professionals without requiring them to master a high-level programming language such as C# or VB.NET.

Extensibility, Extensibility, Extensibility

This design principle aims to make the IT administrator more productive by providing greater control over the Windows environment and accelerating the automation of system administration. Administrators can start PowerShell and use it immediately without having to learn anything because it runs existing commands and scripts, and is therefore easy to adopt. It is an easy to use shell and language for administrators.

All commands in PowerShell are called *cmdlets* (pronounced ''commandlet''), and they use verb-noun syntax — for example, `Start-Service`, `Stop-Service` or `Get-Process`, `Get-WMIObject`, and so on. The intuitive nature of verb-noun syntax makes learning commands easy for administrators. PowerShell includes more than 100 commands and utilities that are admin focused. In addition, PowerShell provides a powerful scripting language that supports a wide range of scripting styles, from simple to sophisticated. This enables administrators to write simple scripts and learn the language as they go. With this combined functionality and ease of use, PowerShell provides a powerful environment for administrators to perform their daily tasks.

Tear Down the Barriers to Development

Another design principle of PowerShell is to make it easy for developers to create command-line tools and utilities. It provides common argument parsing code, parameter binding code that enables

developers to write code only for the admin functionality they are providing. The PowerShell development model separates the processing of objects from formatting and outputting. PowerShell provides a set of cmdlets for manipulating objects, formatting objects, and outputting objects. This eliminates the need for developers to write this code. PowerShell leverages the power of .NET, which enables developers to take advantage of the vast library of this framework. It provides common functionality for logging, error handling, and debugging and tracing capabilities.

A Quick Tour of Windows PowerShell

This section presents a quick tour of Windows PowerShell. We'll start with a brief look at installing the program, and then move right into a discussion of cmdlets.

You start Windows PowerShell either by clicking the Windows PowerShell shortcut link or by typing **PowerShell** in the Run dialog box (see Figure 1-1).

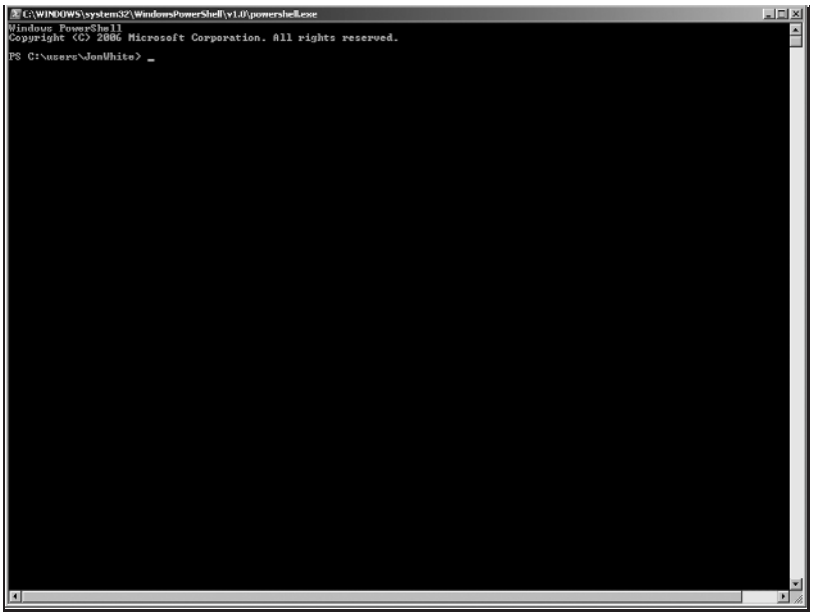

Figure 1-1: Click the shortcut link and you'll get the prompt shown here.

Cmdlets

Windows PowerShell enables access to several types of commands, including functions, filters, scripts, aliases, cmdlets, and executables (applications). PowerShell's native command type is the cmdlet. A *cmdlet* is a simple command used for interacting with any management entity, including the operating system. You can think of a cmdlet as equivalent to a built-in command in another shell. The traditional shell generally processes commands as separate executables, but a cmdlet is an instance of a .NET class, and runs within PowerShell's process.

Chapter 1: Introduction to PowerShell

Windows PowerShell provides a rich set of cmdlets, including several that enhance the discoverability of the shell's features. We begin our tour of Windows PowerShell by learning about a few cmdlets that will help you get started in this environment. The first cmdlet you need to know about is `get-help`:

```
PS C:\> get-help
TOPIC
    Get-Help


SHORT DESCRIPTION
    Displays help about PowerShell cmdlets and concepts.


LONG DESCRIPTION


SYNTAX
    get-help {<CmdletName> | <TopicName>}
    help {<CmdletName> | <TopicName>}
    <CmdletName> -?


    "Get-help" and "-?" display help on one page.
    "Help" displays help on multiple pages.

    Examples:
      get-help get-process   : Displays help about the get-process cmdlet.
      get-help about-signing : Displays help about the signing concept.
      help where-object      : Displays help about the where-object cmdlet.
      help about_foreach     : Displays help about foreach loops in PowerShell.
      match-string -?        : Displays help about the match-string cmdlet.


    You can use wildcard characters in the help commands (not with -?).
    If multiple help topics match, PowerShell displays a list of matching
    topics. If only one help topic matches, PowerShell displays the topic.


    Examples:
      get-help *      : Displays all help topics.
      get-help get-*  : Displays topics that begin with get-.
      help *object*   : Displays topics with "object" in the name.
      get-help about* : Displays all conceptual topics.




    For information about wildcards, type:
      get-help about_wildcard



REMARKS
```

```
    To learn about PowerShell, read the following help topics:
      get-command  : Displays a list of cmdlets.
      about_object : Explains the use of objects in PowerShell.
      get-member   : Displays the properties of an object.


    Conceptual help files are named "about_<topic>", such as:
      about_regular_expression.

    The help commands also display the aliases on the system.
    For information about aliases, type:

      get-help about_alias


PS C:\>
```

As you can see, `get-help` provides information about how to get help on PowerShell cmdlets and concepts. This is all well and good, but you also need to be able to determine what commands are available for use. The `get-command` cmdlet helps you with that:

```
PS C:\> get-command

CommandType     Name                              Definition
-----------     ----                              ----------
Cmdlet          Add-Content                       Add-Content [-P
Cmdlet          Add-History                       Add-History [[-
Cmdlet          Add-Member                        Add-Member [-Me
Cmdlet          Add-PSSnapin                      Add-PSSnapin [-
Cmdlet          Clear-Content                     Clear-Content [
Cmdlet          Clear-Item                        Clear-Item [-Pa
Cmdlet          Clear-ItemProperty                Clear-ItemPrope
Cmdlet          Clear-Variable                    Clear-Variable
Cmdlet          Compare-Object                    Compare-Object
Cmdlet          ConvertFrom-SecureString          ConvertFrom-Sec
Cmdlet          Convert-Path                      Convert-Path [-
Cmdlet          ConvertTo-Html                    ConvertTo-Html
Cmdlet          ConvertTo-SecureString            ConvertTo-Secur
Cmdlet          Copy-Item                         Copy-Item [-Pat

...
```

As shown in the preceding output, `get-command` returns all the available commands. You can also find cmdlets with a specific verb or noun:

```
PS C:\> get-command -verb get

CommandType     Name                              Definition
-----------     ----                              ----------
Cmdlet          Get-Acl                           Get-Acl [[-Path]
Cmdlet          Get-Alias                         Get-Alias [[-Nam
Cmdlet          Get-AuthenticodeSignature         Get-Authenticode
Cmdlet          Get-ChildItem                     Get-ChildItem [[
Cmdlet          Get-Command                       Get-Command [[-A
```

```
Cmdlet          Get-Content                    Get-Content [-Pa
Cmdlet          Get-Credential                 Get-Credential [
Cmdlet          Get-Culture                    Get-Culture [-Ve
Cmdlet          Get-Date                       Get-Date [[-Date
Cmdlet          Get-EventLog                   Get-EventLog [-L
Cmdlet          Get-ExecutionPolicy            Get-ExecutionPol
Cmdlet          Get-Help                       Get-Help [[-Name
Cmdlet          Get-History                    Get-History [[-I
Cmdlet          Get-Host                       Get-Host [-Verbo
Cmdlet          Get-Item                       Get-Item [-Path]
Cmdlet          Get-ItemProperty               Get-ItemProperty
Cmdlet          Get-Location                   Get-Location [-P
Cmdlet          Get-Member                     Get-Member [[-Na
Cmdlet          Get-PfxCertificate             Get-PfxCertifica
Cmdlet          Get-Process                    Get-Process [[-N
Cmdlet          Get-PSDrive                    Get-PSDrive [[-N
Cmdlet          Get-PSProvider                 Get-PSProvider [
Cmdlet          Get-PSSnapin                   Get-PSSnapin [[-
Cmdlet          Get-Runspace                   Get-Runspace [[-
Cmdlet          Get-Service                    Get-Service [[-N
Cmdlet          Get-TraceSource                Get-TraceSource
Cmdlet          Get-UICulture                  Get-UICulture [-
Cmdlet          Get-Unique                     Get-Unique [-Inp
Cmdlet          Get-Variable                   Get-Variable [[-
Cmdlet          Get-WmiObject                  Get-WmiObject [-
```

When commands are executed, their output is returned to the shell in the form of .NET objects. (In the case of native commands, the text output of the command is converted to .NET string objects before being returned.) These objects can be directly queried and manipulated by using the object's properties and methods. Fortunately, you don't have to know the properties and methods of each object in order to manipulate it. If you're unfamiliar with an object's type, you can use the `get-member` cmdlet to examine its members:

```
PS C:\> "Hello"  |  get-member

   TypeName: System.String

Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone()
CompareTo        Method               System.Int32 CompareTo(Object value),...
Contains         Method               System.Boolean Contains(String value)
CopyTo           Method               System.Void CopyTo(Int32 sourceIndex,...
EndsWith         Method               System.Boolean EndsWith(String value)...
Equals           Method               System.Boolean Equals(Object obj), Sy...
GetEnumerator    Method               System.CharEnumerator GetEnumerator()
GetHashCode      Method               System.Int32 GetHashCode()
GetType          Method               System.Type GetType()
GetTypeCode      Method               System.TypeCode GetTypeCode()
get_Chars        Method               System.Char get_Chars(Int32 index)
get_Length       Method               System.Int32 get_Length()
IndexOf          Method               System.Int32 IndexOf(Char value, Int3...
```

```
IndexOfAny       Method                 System.Int32 IndexOfAny(Char[] anyOf,...
Insert           Method                 System.String Insert(Int32 startIndex...
IsNormalized     Method                 System.Boolean IsNormalized(), System...
LastIndexOf      Method                 System.Int32 LastIndexOf(Char value, ...
LastIndexOfAny   Method                 System.Int32 LastIndexOfAny(Char[] an...
Normalize        Method                 System.String Normalize(), System.Str...
PadLeft          Method                 System.String PadLeft(Int32 totalWidt...
PadRight         Method                 System.String PadRight(Int32 totalWid...
Remove           Method                 System.String Remove(Int32 startIndex...
Replace          Method                 System.String Replace(Char oldChar, C...
Split            Method                 System.String[] Split(Params Char[] s...
StartsWith       Method                 System.Boolean StartsWith(String valu...
Substring        Method                 System.String Substring(Int32 startIn...
ToCharArray      Method                 System.Char[] ToCharArray(), System.C...
ToLower          Method                 System.String ToLower(), System.Strin...
ToLowerInvariant Method                 System.String ToLowerInvariant()
ToString         Method                 System.String ToString(), System.Stri...
ToUpper          Method                 System.String ToUpper(), System.Strin...
ToUpperInvariant Method                 System.String ToUpperInvariant()
Trim             Method                 System.String Trim(Params Char[] trim...
TrimEnd          Method                 System.String TrimEnd(Params Char[] t...
TrimStart        Method                 System.String TrimStart(Params Char[]...
Chars            ParameterizedProperty System.Char Chars(Int32 index) {get;}
Length           Property               System.Int32 Length {get;}
```

Windows PowerShell also enables you to execute existing native operating system commands and scripts. The following example executes the ipconfig.exe command to find out about network settings:

```
PS C:\> ipconfig
Windows IP Configuration
Wireless LAN adapter Wireless Network Connection:
   Connection-specific DNS Suffix  . : ARULHOMELAN
   Link-local IPv6 Address . . . . . : fe80::c4e0:69b3:5d35:9b4b%9
   IPv4 Address. . . . . . . . . . . : 192.168.1.13
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
```

In a traditional shell, when you want to get the IP address output by the `IPConfig.exe` utility, you have to perform text parsing. For example, you might do something like get the ninth line of text from the output and then get the characters starting from the thirty-ninth character until the end of the line to get the IP address. PowerShell enables you to perform this style of traditional text processing, as shown here:

```
PS C:\Users\arulk> $a = ipconfig
PS C:\Users\arulk> $a[8]
   IPv4 Address. . . . . . . . . . . : 192.168.1.13
PS C:\Users\arulk> $a[10].Substring(39)
192.168.1.13
```

However, this kind of text processing is very brittle and error prone. If the output of `IPconfig.exe` changes, then the preceding script breaks. For example, because PowerShell converts to text output

by exes and scripts as `String` objects, it is possible to achieve better text processing. In the preceding example, we are looking for the line that contains IP in the text:

```
PS C:\> $match = @($a | select-string "IP")
PS C:\> $ipstring = $match[0].line
PS C:\> $ipstring
   IPv4 Address. . . . . . . . . . . : 192.168.1.13
PS C:\> $index = $ipstring.indexof(": ")
PS C:\> $ipstring.Substring($index+2)
PS C:\> $ipaddress = [net.ipaddress]$ipstring.Substring($index+2)
PS C:\> $ipaddress
```

In the preceding script, the first line searches for the string IP in the result variable `$a`. `@(...)` and converts the result of execution into an array. The reason we do this is because we will get multiple lines that match the IP in computers that have multiple network adapters. We are going to find out the `ipaddress` in the first adapter. The result returned by `select-string` is a `MatchInfo` object. This object contains a member `Line` that specifies the actual matching line. (I know this because I used `get-member` to find out.) This string contains the IP address after the characters `": "`. Because the `Line` property is a `String` object, you use the `String` object's `IndexOf` method (again, I used `get-member`) to determine the location where the IP address starts. You then use `Substring` with an index of `+ 2` (for `": "` characters) to get the IP address string. Next, you convert the IP address string into the .NET `IPAddress` object, which provides more type safety. As you can see, Windows PowerShell provides great functionality for doing traditional text processing.

Next, let's look at the COM support in PowerShell:

```
PS C:\> $ie = new-object -com internetexplorer.application
PS C:\> $ie.Navigate2("http://blogs.msdn.com/powershell")
PS C:\> $ie.visible = $true
PS C:\> $ie.Quit()
```

You can create COM objects using the `new-object` cmdlet, with the `-com` parameter specifying the programmatic ID of the COM class. In the preceding example, we create an Internet Explorer object and navigate to the blog of the Windows PowerShell team. As before, you can use `get-member` to find out all the properties and methods a COM object supports. Do you see a pattern here?

In addition to COM, PowerShell also has great support for WMI.:

```
PS C:\Users\arulk> $a = get-wmiobject win32_bios
PS C:\Users\arulk> $a


SMBIOSBIOSVersion : Version 1.50
Manufacturer      : TOSHIBA
Name              : v1.50V
SerialNumber      : 76047600H
Version           : TOSHIB - 970814
```

Using `get-wmiobject`, you can create any WMI object. The preceding example creates an instance of a `Win32_Bios` object.

Now that you've seen some of PowerShell's capabilities firsthand, let's take a look at what goes on under the hood while you're providing this functionality to the shell's user.

High-Level Architecture of Windows PowerShell

PowerShell has a modular architecture consisting of a central execution engine, a set of extensible cmdlets and providers, and a customizable user interface. PowerShell ships with numerous default implementations of the cmdlets, providers, and the user interface, and several third-party implementations are provided by other groups at Microsoft and by external companies.

The following sections provide details about each of the architectural elements illustrated in Figure 1-2.

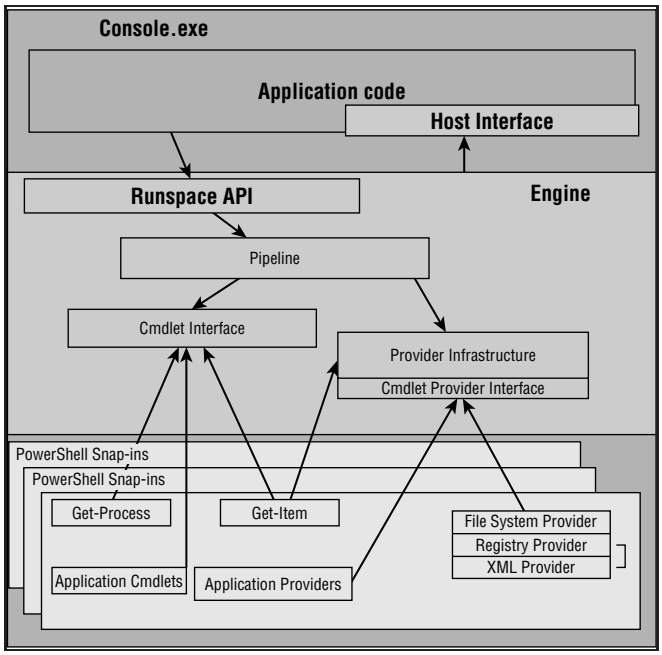

Figure 1-2: The high-level architecture of Windows PowerShell

Host Application

The Windows PowerShell engine is designed to be hostable in different application environments. In order to make use of PowerShell functionality, it needs to be hosted in an application that implements the Windows PowerShell host interface. The host interface is a set of interfaces that provides functionality enabling the engine to interact with the user. This includes but is not limited to the following:

- Getting input from users
- Reporting progress information
- Output and error reporting

The hosting application can be a console application, a windows application, or a Web application. Windows PowerShell includes a default hosting application called `PowerShell.exe`, which is console based. If you're like most developers, you'll rarely need to write your own host implementation. Instead,

you'll make use of PowerShell's host interface to interact with the engine. You only need to write a hosting application when you have application requirements for an interface that is richer than the interface provided by the default hosting application. Writing a hosting application involves implementing Windows PowerShell host interfaces and using the Windows PowerShell Runspace and Pipeline APIs to invoke commands. Together, these two interfaces enable communication between the application and the Windows PowerShell engine. You'll learn the details about writing a hosting application in Chapter 7.

Windows PowerShell Engine

The Windows PowerShell engine contains the core execution functionality and provides the execution environment for cmdlets, providers, functions, filters, scripts, and external executables. The engine exposes the functionality through the `Runspace` interface, which is used by the hosting application to interact with the engine. At a high level, the engine consists of a runspace, which is like an instance of the engine, and one or more *pipelines*, which are instances of command lines. These pipeline components interact with the cmdlets through the `cmdlet` interface. All cmdlets need to implement this interface to participate in the pipeline. Similarly, the pipeline interacts with the providers through a well-defined set of provider interfaces. We will delve into more details about the engine as we progress in the book.

Windows PowerShell Snap-ins

Windows PowerShell provides an extensible architecture for adding functionality to the shell by means of snap-ins. A *snap-in* is a .NET assembly or set of assemblies that contains cmdlets, providers, type extensions, and format metadata. All the commands and providers that ship as part of the Windows PowerShell product are implemented as a set of five snap-ins. You can view the list of snap-ins using the `get-pssnapin` cmdlet:

```
PS C:\> get-pssnapin

Name        : Microsoft.PowerShell.Core
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains Windows PowerShell manage-
ment cmdlets used to manage components of Windows PowerShell.

Name        : Microsoft.PowerShell.Host
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains cmdlets used by the Win-
dows PowerShell host.

Name        : Microsoft.PowerShell.Management
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains management cmdlets used to man-
age Windows components.

Name        : Microsoft.PowerShell.Security
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains cmdlets to manage Windows Pow-
erShell security.

Name        : Microsoft.PowerShell.Utility
```

```
PSVersion   : 1.0
Description : This Windows PowerShell snap-in contains utility Cmdlets used to  manip-
ulate data.
```

Summary

This chapter introduced you to some basic cmdlets to help with discoverability. It also described the high-level architecture of PowerShell. From here, we'll move on to the first step beyond the cmdlet level: learning how to develop a custom snap-in package. The techniques in the following chapter lay the foundation for creating your own cmdlets and providers. You'll also learn about PowerShell's model for distributing and deploying the code you write.