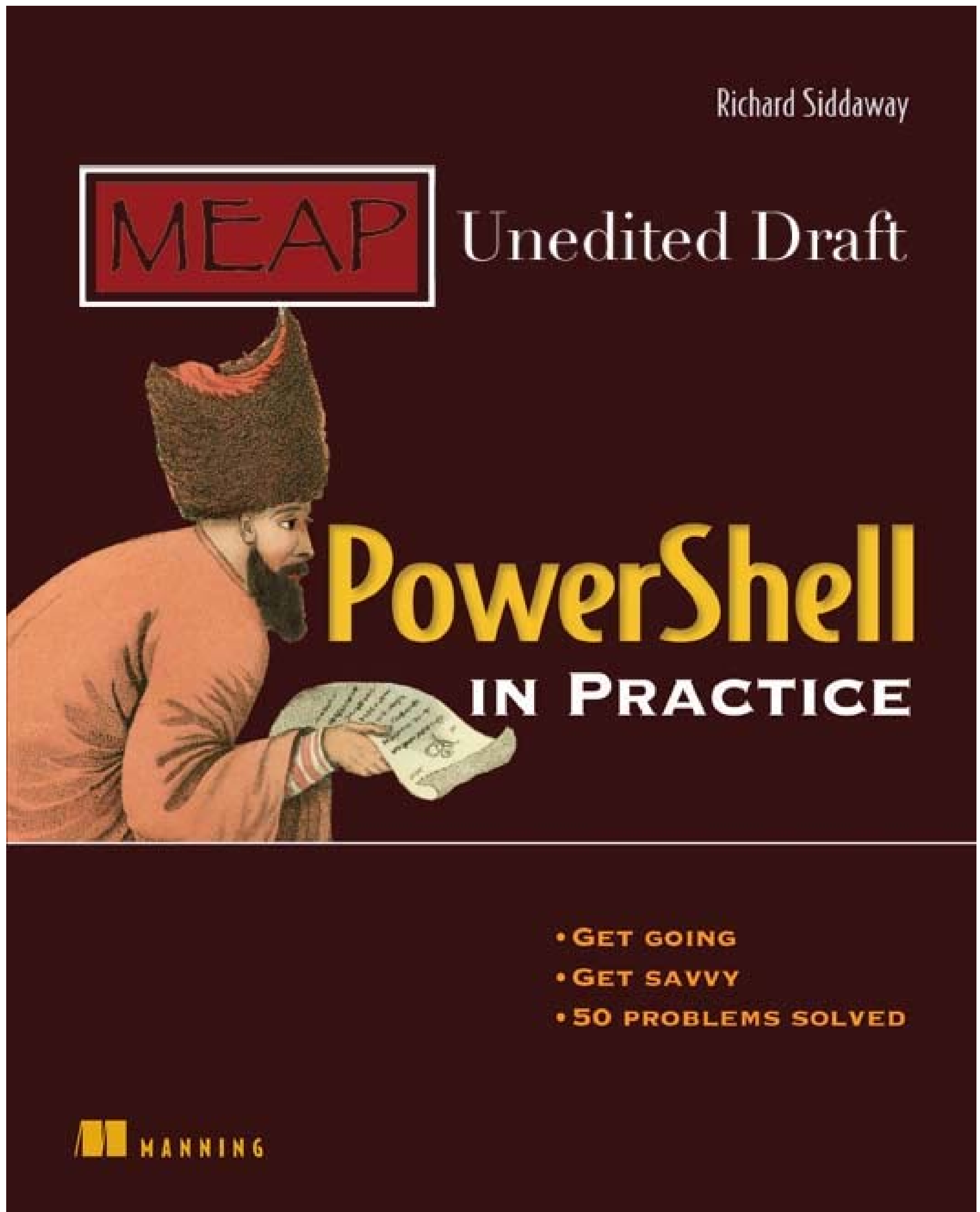

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

MEAP Edition
Manning Early Access Program

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

Part 1: Getting Started with PowerShell

1. PowerShell Fundamentals
2. Learning PowerShell
3. Using other technologies with PowerShell
4. Automating administration

Part 2: Working with People

5. User Accounts
6. Mailboxes
7. Desktop

Part 3: Working with Servers

8. Windows Servers
9. DNS
10. Active Directory Structure and Administration
11. Active Directory - Topology
12. Exchange Server 2007
13. IIS 7
14. SQL Server

Appendices

A. PowerShell Additions
B. PowerShell Reference
C. PowerShell – More Information
- **a. Books**
- **b. Websites**
- **c. Blogs**

D. Other things you can do with PowerShell

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

Part 1

Getting started with PowerShell

Welcome to PowerShell in Practice. PowerShell is the new command shell and scripting language from Microsoft. This book will enable you to use Windows PowerShell to administer your Windows servers and applications such as SQL Server, IIS 7, Exchange 2007 and Active Directory from the command line. PowerShell provides a more efficient and powerful mechanism for administration that will save you time and effort in your daily job. If you are a PowerShell novice or a more experienced user there will be something for you in the many examples used to illustrate PowerShell based administration.

The book is divided into three sections. Section 1 covers the fundamentals of working with PowerShell including an explanation of what it is and how it works. Installation and configuration of PowerShell are followed by a look at the unique features of PowerShell.

Chapter 2 will show how to learn to use PowerShell with practical examples to speed the process. Chapter 3 covers the other technologies that are required to work with PowerShell namely .NET, COM, ADSI and WMI. The final chapter in this section, Chapter 4 is concerned with the process of automation and best practice around writing scripts.

Section 2 shows how to perform administrative tasks that are concerned with people. These tasks include managing user accounts in Active Directory and on local systems, managing Exchange mailboxes and the user's desktop.

Section 3 looks at working with servers starting with Windows, including the new Server Core install option in Windows Server 2008. Subsequent chapters consider Exchange 2007, SQL Server, IIS 7, DNS and Active Directory including the new features in Windows Server 2008.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

1
PowerShell Fundamentals

Microsoft seems to be always talking about PowerShell. Listen to a talk about Exchange Server 2007 or Windows server 2008 or even SQL Server 2008 and PowerShell will be mentioned. PowerShell gets its own section on the Microsoft scripting center and there are a whole stack of books on the subject. So what is PowerShell and why are so many people excited about it? This chapter introduces Windows PowerShell and answers some of those basic questions. In this chapter you will discover

- What PowerShell is and why so many people are excited about it
- Why you should use it
- The benefits of learning to use PowerShell
- The major features of PowerShell that make it stand out from other automation tools in the Windows arena
- The things that PowerShell is really good at and the odd area where you shouldn't use it
- What changes you can expect with version 2 of PowerShell

©Manning Publications Co. Please post comments or corrections to the Author Online forum: http://www.manning-sandbox.com/forum.jspa?forumID=499

PowerShell is a tool that is designed by administrators for administrators. It is a way to automate your day-to-day administration work that is fully supported by Microsoft as well as being adopted by an increasing number of other vendors. PowerShell is very powerful compared to other scripting tools - 86 lines of VBScript code can be condensed to a single line of PowerShell.

Microsoft is building PowerShell into all of their major products. This will give a consistent and coherent way to manage windows and the services such as Exchange and SQL Server that are installed on it. It will save you time and administrative effort across your Windows based servers and will amply repay the time spent learning it.

PowerShell was released in November 2006 as a free download from the Microsoft web site. It is also being incorporated into Microsoft products such as Exchange Server 2007 and SQL Server 2008. PowerShell is built on .NET but you do not have to be a .NET programmer to use it.

PowerShell has a number of unique features such as cmdlets and providers. These features form the fundamentals of PowerShell. Understanding these features is essential to understanding PowerShell and its terminology. They will be explained with examples.

Scripting languages need to be able to perform utility functions such as sorting, grouping and comparing. PowerShell has a number of utility cmdlets to perform these roles. Their use is fundamental to getting the most from the PowerShell pipeline. We will discover how to use these cmdlets with some practical examples relating to tasks that Windows administrators will need to perform. In this chapter, and throughout the book, the examples will be drawn from practical administrative tasks rather than demonstrating PowerShell as a programming language.

The PowerShell team has not stopped working on the product. The first Community Technology Preview (CTP) for PowerShell version 2 became available approximately one year after version 1 was released! While writing about any product that is still under development is dangerous as changes can occur in the final version there are a number of new features in version 2 that need to be mentioned to ensure a full understanding of version 1.

PowerShell, like any tool, has a learning curve. It seems to be very steep when you are first introduced to it but this chapter and the next three will lay the foundations for us to dive into using PowerShell to make our day-to-day administrative tasks quicker and easier. This will enable us to spend more time on other, potentially more interesting tasks. At the end of the chapter you will understand what PowerShell is and more importantly what it isn't; what the major features are and how they work and have an understanding of the utility commands within PowerShell.

NOTE

A number of PowerShell commands will be used in this chapter including Get-Member, Get-Command, Get-Help and Get-PSDrive. This chapter will provide sufficient information to explain the particular example. A full explanation of these commands will have to wait until Chapter 2.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

1.1 Open the box

PowerShell is a download or optional install so there isn't a box! Newcomers to PowerShell usually ask a couple of questions. The first question is always "What is PowerShell?" This section will answer that question. The second question is "What can I do with it?" The answer to that question takes the rest of the book.

A simple answer to the question "What is PowerShell" would be that it is the new scripting language and command line shell for Microsoft products. It is better described as the automation engine that Microsoft is building into all major products as Figure 1.1. shows.

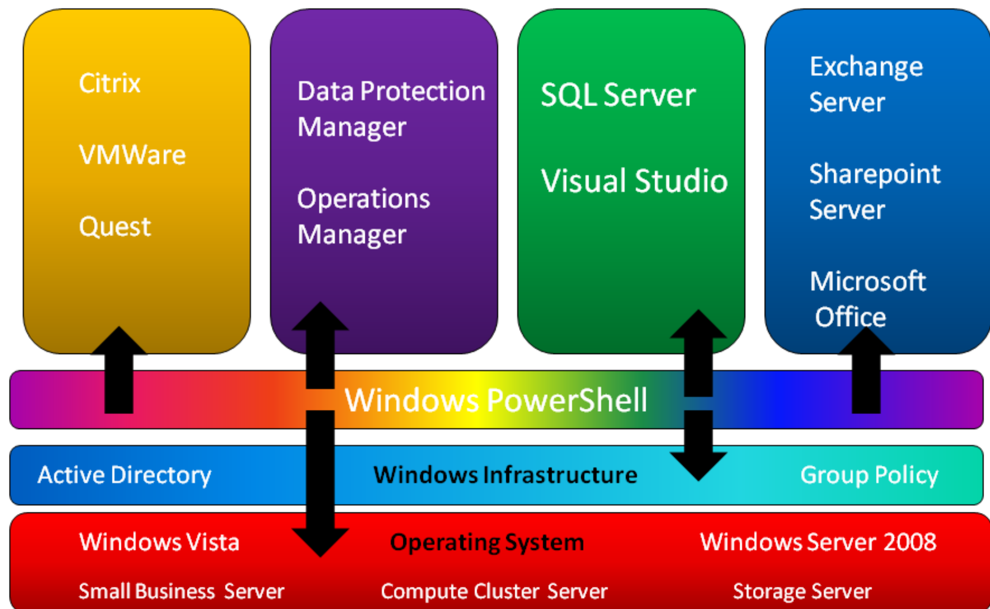

Figure 1.1 PowerShell is the automation and integration layer in a Microsoft environment. It can be used to administer Windows systems as well as an increasing number of Microsoft and third part applications.

The shell and scripting language is the most visible implementation of PowerShell but it can also be hosted in .NET applications. That aspect of PowerShell is outside the scope of this book (thankfully). We will be concentrating on using PowerShell at the command line and in scripts for administering Windows based systems.

Windows administration is often viewed as a GUI based occupation. One of the major failings of the Windows Operating System, at least according to UNIX and Linux administrators, is the inability to perform the powerful shell based, command line administration activities they traditionally use. PowerShell addresses that failing and provides a first class command line experience that makes administrator's life easier. It is so good

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

that there is an open source project to port PowerShell to the UNIX\Linux platform! This book will show you how to get the most out of PowerShell.

1.1.2 .NET - not necessarily

PowerShell is .NET based and enables the .NET framework to be used in scripts and from the command line. This mixture of interactive and scripting use makes it easy to start using and building on what you already know. Great scripts from little cmdlets grow.

ADMINISTRATORS PLEASE READ THIS!

You do NOT have to become a .NET programmer to be able to use PowerShell. It is perfectly possible to work with PowerShell and never use any .NET code. There are a lot of examples of using .NET code within PowerShell that can be downloaded and reused.

PowerShell uses a syntax that is similar to C#. If you do any C# programming you will find it close enough to be confusing sometimes. It is not necessary to use a semi-colon at the end of each line though one can be used as a line separator if multiple PowerShell lines are combined. PowerShell is not case sensitive like C#.

PowerShell commands produce .NET objects rather than the text output produced by other shells. The objects may not be "pure" .NET objects in that PowerShell creates a wrapper around the object. This wrapper controls the methods and properties in the output object. Extra properties called noteproperty and scriptproperty may be added to a PowerShell output object. A noteproperty enables a new piece of data to be attached to the object while a scriptproperty is property whose value is determined a PowerShell script block.

The relationship between PowerShell and .NET together with how to use the .NET framework are covered in Chapter 3.

Now we have an idea of what PowerShell is we will consider why it is worthwhile learning.

1.2 Why PowerShell?

After asking "What is PowerShell?" the next question is often "Why should I bother with PowerShell"? I am assuming that if you are reading this book you are interested in using PowerShell. There are many parts to the answer to "Why PowerShell"? I think the following sections answer the question. Learning every new technology has some "Eureka" moments where everything suddenly clicks. I will share a few of those moments as we progress through the book.

PowerShell is not the answer to every problem. There are a number of situations where PowerShell is difficult to use or cannot be used:

- Windows 2008 Server Core (but see Chapter 8)
- Logon scripts
- WinPe environments because .NET is not loaded

This still leaves the vast majority of the Windows environment for PowerShell

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

1.2.1 Eureka 1

I was once asked to look through a 12,000 seat Active Directory to find all of the users that did not have Outlook Web Access enabled. Not the sort of task to perform through the GUI tools! I wrote a script that has been reused several times since. It took less time to write and test the script than it would have done to perform the process manually. That time can be spent on other, more interesting tasks.

The original script was written in VBScript as that was all I had available at the time. The script occupied 86 lines of code and took me about a day to conceive, write and test.

When PowerShell became available in Exchange Server 2007 I converted the code to PowerShell. It took me about 30 minutes, most of which was starting the virtual machine (this was when Exchange Server 2007 was in beta) and looking up the appropriate cmdlets. Those 86 lines of VBScript condensed to one line of PowerShell that consisted of three cmdlets linked on the pipeline.

That was when I had the realization as to just how powerful PowerShell was and how much coding it was going to save me. Eureka - PowerShell Rocks!

1.2.2 Importance to you

PowerShell is an important technology to you the administrator. It is a small download but it has a large impact on the administration of a Windows environment. The way things are going in the Microsoft world if you can't do things at the command line that is through PowerShell you will be stuck with the mundane jobs. It is being built into all of the major Microsoft products, either as part of the product or as an optional download including:

- Windows Server 2008
- Exchange Server 2007
- SQL Server 2008
- IIS 7
- Members of the System Center family
- Small Business Server 2008 and Windows Essential Business Server 2008.

Microsoft's Common Engineering Criteria for 2009 include PowerShell. The one major omission from the list appears to be SharePoint but it is possible to use the .NET APIs for SharePoint within PowerShell.

Using the same automation engine across all Microsoft products enables the transfer of skills across products. The MMC GUI tools have a (more or less) common look and feel. This has accelerated learning as the tools are navigated and used in the same way. PowerShell brings this same concept to the command line. Product specific add-ins building on a common language base mean that only the new commands need to be learnt rather than a whole new language. PowerShell also provides the common administration tools that VBScript has never had.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

As PowerShell appears in more Microsoft, and third party, products it will be the best way to automate the administration of your Windows systems. PowerShell is already incorporated into products from Quest, IBM, Citrix, VMWare, Special Operations Software and SDM Software for example. Some of these we will be meeting in later chapters. The ability to use the same basic language makes PowerShell the only way to integrate administration using these products.

An open source project is working to port it to the Linux platform. This holds the promise of being able to administer Windows and Linux from the same shell. That will be an obvious benefit to those working in heterogeneous environments.

1.2.2 Designed for you

PowerShell has been designed from the very beginning for administrators. It has built in access to a number of the commonest things that administrators are interested in including:

- Processes - what is running on the machine?
- Services
- Event logs - what is happening on the machine?
- ACLs for security
- WMI - oh so much easier than VBScript
- File system.

One of the points that brings this home is that PowerShell understands GB, MB and KB as Gigabyte, Megabyte and Kilobyte respectively. PowerShell is case insensitive so gb, mb and kb or any combination of case are equally understood.

Listing 1.1 Use of GB, MB and KB

```
PS> 1kb
1024
PS> 1mb
1048576
PS> 1gb
1073741824
PS> (1024*1024)/1MB
1
```

The terms can be used in a standalone manner or can be used in calculations as shown.

PowerShell can access the full range of .NET as well as older COM interfaces. This allows the administrator to continue to work with known tools.

1.2.4 Quicker and more powerful

There is a perception that the only way to administer Windows based systems is through the GUI tools. In fact Microsoft has been increasing the support for command line administration through the various versions of Windows since Windows 2000. The use of command line

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

tools was emphasized at many technical events after the launch of Windows 2000. With each subsequent release more command lien tools have been added. Microsoft has also promoted the use of scripting tools much more over the last five years or so.

If you need to perform an administrative action on a single user in Active Directory it may be as fast to use the GUI as to use a script. If you have to perform that same action on 100 users it will definitely be quicker, and easier, to use a script. Once the script is written it can be saved and used for the one user or 100 user scenarios. The return on the time spent writing the script is paid back every time you use it plus it makes you look good. If you can script it you must really understand this stuff. Right?

The venerable command file could be regarded as the first, if very limited, scripting language on Windows. Command files have very limited functionality and rely to a large degree on command line tools to perform most tasks. These tools cannot be integrated and only pass text between them making processing very difficult.

VBScript was introduced early in the life time of Windows NT. At that time scripting was not regarded as a main stream activity by Windows administrators. That perception is slowly changing but the majority of Windows administrators, in my experience, still prefer not to write scripts.

NOTE

I have found the UNIX administrators that become involved in administering Windows often adopt PowerShell much quicker than administrators who have always worked with Windows.

VBScript is COM based. This gives it access to a wide range of interfaces for administration. Unfortunately they are often very different in the way the work and the way they are used. This makes VBScript difficult to use. There are gaps in the products that can be administered through VBScript which reduces its potential.

PowerShell can be used interactively at the command line as well as in a script which makes testing, and development, much easier. This is not possible with VBScript. The VBScript commands have to be in a file which is then executed making testing and development a slower and more difficult task.

1.2.5 Extensible and flexible

PowerShell is easily extensible. Writing cmdlets is fairly straight forward and while providers may be more complicated there are examples available. There are many commercial and open source PowerShell extensions available. Some of these extensions will be covered in Chapter 4.

PowerShell is a very flexible system. There are often a number of ways of achieving the same task. This allows administrators to find a method with which they feel comfortable. It also means that it is more likely that someone will have a found a solution to your problem and posted the script on a blog or forum.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

This flexibility can be a disadvantage. Many people have commented that it is a weakness of PowerShell that there can be multiple methods of achieving the same end. I disagree that it is a weakness, however it can make life much more difficult for a new comer. He has a problem to solve so searches the Internet for a script to copy or alter. Our administrator may find three scripts that say they do they same thing but seem to be very different - which one should he use. This can be a difficulty but the idea of this book is to present the administrator with the information required to make that choice or better still be able to write the script himself and share it with the wider PowerShell community. No doubt some people looking at the examples will say "Oh he should have done it this way...." The examples I use are those that seem to me to be the most straightforward to use and learn. When it comes to PowerShell the old saying "if you have three techies in a room there at least four opinions on how to do something" was never truer. All of those opinions will be good though.

It is time to start learning about PowerShell and we will start with the major features of PowerShell. These are the things that stick in your mind and make you realize it is different.

1.3 Major Features

PowerShell has a number of features that combine to make it unique and to make it such a powerful tool. We will examine the language in more detail in the next chapter but for now the most obvious features will be covered. These include:

- Cmdlets
- Pipeline
- Providers
- Help system

Putting these things together will give us the basics of PowerShell that we can take into the rest of the book. I will be concentrating on the needs of the administrator wanting to know how to use these features rather than looking at it purely from a programming view point.

One of the great strengths of PowerShell is that it can be used interactively as well as in scripts. The same commands should, and usually do, work equally well from the command line and in scripts.

1.3.1 Cmdlets

Cmdlets are probably the most obviously different feature when comparing PowerShell to other scripting languages. A cmdlet (I always pronounce it as command-let) is a small, self contained piece of functionality that does one specific job. A cmdlet is analogous to a cmd shell command such as ping.exe. PowerShell version 1 has 129 cmdlets. Version 2 adds a further 50 or so to that total. One of the nice things about PowerShell is that it is very easy to discover information like this using PowerShell itself. In this case I used the following code.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

```
 (Get-Command | Where-Object {$_.PSSnapin -like
"Microsoft.P*"}).Count
```

Get-Command generates a list of PowerShell commands. That list is piped into a filter (Where-Object) that only passes those commands installed by a PowerShell snapin (method of extending PowerShell) whose names start "Microsoft.P". We then count the number of commands in the filtered list as shown in Fig. 1.2.

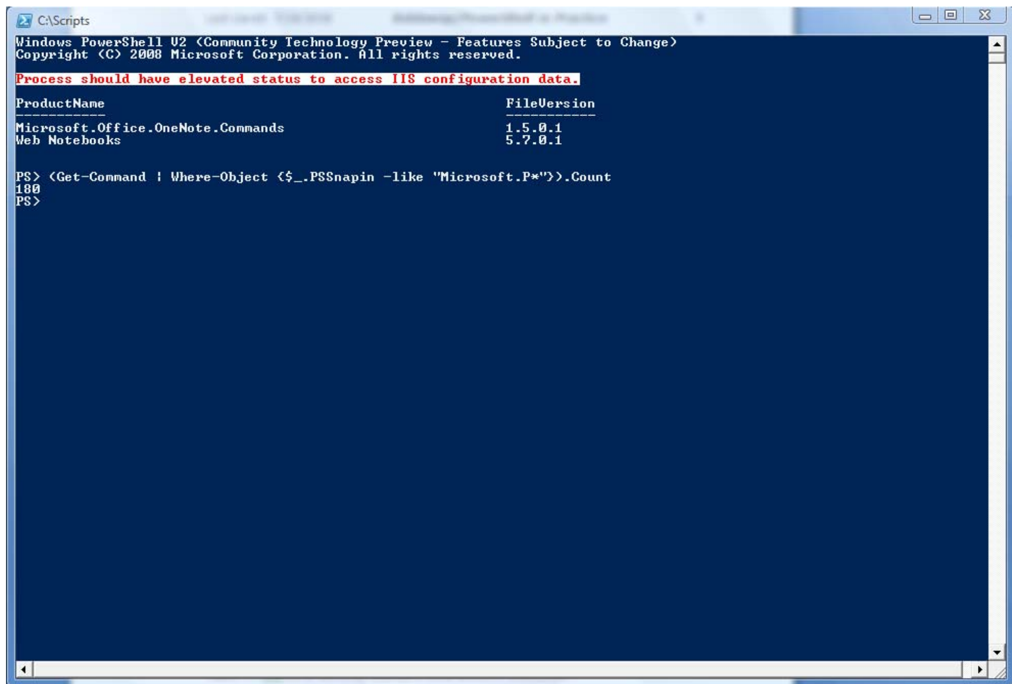

Figure 1.2 PowerShell shell used to count number of cmdlets

NOTE

PowerShell is case insensitive. The code in the listing above could have been written in all lowercase, all upper case or any random combination of the two. I will follow the style of PowerShell itself when capitalizing cmdlet names, properties or methods.

OPERATORS

The operator -like is used to perform the comparison in the example above. PowerShell operators are detailed in Appendix A.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

It seems simple enough but this one line of code demonstrates a number of PowerShell features. It starts with a cmdlet Get-Command. This, like all cmdlets, has a verb-noun syntax. It starts with a verb. The PowerShell team maintains a list of approved verbs. Their aim to ensure consistency for example any time you have a command that is fetching information the verb to use is get. The second part of the name is a noun that describes what the verb is acting on in this case the commands within PowerShell. Get-Command retrieves information regarding the installed cmdlets. We will be learning much more about Get-Command in the next chapter. Having generated a list of cmdlets we pass that list onto the pipeline. I will cover the pipeline in much greater detail in the next section.

The second cmdlet, Where-Object which is one of the utility cmdlets , functions as a filter acting on the information moving along the pipeline in this case on each command. The filter determines if the PSSnapin property is like the string "Microsoft.P*" where * is the usual wildcard character. Note the use of {} to enclose the filter. By wrapping the cmdlets in () we can treat the results as a collection and use the Count property to determine the number of cmdlets present that match the filter.

The common standard verbs used in PowerShell are shown in table 1.1

Table 1.1 The set of common standard verbs used in PowerShell.

Add	Clear	Compare	Convert
ConvertFrom	ConvertTo	Copy	Export
ForEach	Format	Get	Group
Import	Invoke	Join	Measure
Move	New	Out	Pop
Push	Read	Remove	Rename
Resolve	Restart	Resume	Select
Set	Sort	Split	Start
Stop	Suspend	Tee	Test
Trace	Update	Where	Write

One thing to remember with all cmdlet names is that they should always be singular so use Get-Service rather than Get-Services. This is one of the commonest mistakes when writing PowerShell commands and to prove that PowerShell was designed for you it has a solution for this problem. Tab completion (and the intellisense functionality built into the editors covered in Chapter 4) makes entering PowerShell commands quicker, easier and less error prone.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

TAB COMPLETION

When working at the command line PowerShell demonstrates another feature that aids productivity namely tab completion. If you type Get- at the command line and then press the Tab key the PowerShell engine will complete the command with the first cmdlet that matches what has been typed so far. In this case usually Get-Acl. If the Tab key is pressed again the next Get- cmdlet will be displayed and repeated pressing of the Tab key enables cycling through the list of relevant cmdlets. Tab completion can be invoked from any relevant part of the cmdlet name so for instance Get-C followed by Tab starts cycling through the Get cmdlets whose noun part starts with C.

Tab completion also applies to parameters in that typing - followed by the Tab key enables cycling through the parameter list. As with the cmdlet names the more of the parameter name you give the sooner the process brings the required parameter.

While the in-built Tab completion works well there are alternatives including one from the PowerShell Guy (/\/\o\/\/) at http://thepowershellguy.com/blogs/posh/archive/2007/06/14/powertab-0-93-powershell-tab-expansion-scripts-library.aspx and the PowerShell Community Extensions from http://www.codeplex.com/Wiki/View.aspx?ProjectName=PowerShellCX

ALIASES

As an alternative to typing the full name of a cmdlet or parameter it is possible to use an alias. An alias is shorthand for the command. Aliases can be used at the command line as well as in scripts. The use of aliases saves on typing but at the expense of readability. The list of standard aliases is provided in Appendix A. It is also possible to create your own aliases using the Set-Alias cmdlet.

NOTE

The set of standard set of aliases contains a number of aliases corresponding to traditional commands from the cmd shell including dir, cd, copy and del. There are also a number of aliases for standard UNIX commands including ls, lp, mv and cp. This is a deliberate action to present the administrator with familiar commands wherever possible. The ability to create additional aliases means that the command line tool set can be tailored to match the way you want to work rather than having to learn a new set of commands.

These two scripts can be considered as examples of the use of aliases.

```
gwmi -cl win32_process
Get-WmiObject -Class Win32_Process


gps|?{$_.Handles-gt500}|%{$_.Name}
Get-Process | Where-Object{$_.Handles -gt 500} | ForEach-Object
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum: http://www.manning-sandbox.com/forum.jspa?forumID=499

```
{$_.Name}
```

The first example shows Get-WmiObject and one of its parameters being aliased. The second example shows a slightly contrived example of an aliased script. The use of % and ? makes this especially difficult to read. Heavily aliased scripts can be very off putting for newcomers to PowerShell and should be avoided apart from when working interactively. I would strongly advise against using aliases in scripts it makes them very difficult to understand when you come back to them several months later.

Parameters

PowerShell cmdlets have parameters to define the input and possibly output, or to select various options. An example of using a parameter can be seen in the previous listing. Parameters are always preceded by a hyphen. The parameters of a particular cmdlet can be viewed by using Get-Help. Using a command such as `Get-Help Get-WmiObject -full` will display the parameters of Get-WmiObject as well as the other help information. Typing `Get-Help Get-WmiObject -parameter *` will display only the parameters. As an example the Class parameter from Get-WmiObject can be considered.

```
-Class [<string>]
    Specifies the name of a WMI class. When this parameter is used,
the cmdlet retrieves instances of the WMI class.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       false
    Accept wildcard characters?  false
```

The parameter listing commences with the parameter name and the type of data that can be used with it. This is followed by a short description. The description may contain a list of acceptable values if the parameter will only accept a member of a restricted list of values as input. The Required? option indicates whether the parameter is considered mandatory for that cmdlet with the value given as true or false. If the parameter is mandatory and is not supplied PowerShell will prompt for the value. The Position? Option indicates if data can be passed to the cmdlet and be automatically allocated to the parameter. In this case the first argument passed to the cmdlet is assumed to be the WMI class to retrieve. If the data does not represent a valid wmi class an error will be thrown. If a value of "named" or 0 is given here it means that the parameter name must explicitly be used. Default value indicates if a default value has been set. If the data required by a parameter can be accepted from the pipeline Accept pipeline input? will be set to true. The Accept wildcard characters? option will be set to true if wildcards can be used in the input.

There are a number of common parameters defined for all cmdlets as listed in the following table.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

Table 1.2 Common Parameters

Parameter	Meaning
-Debug	Displays detailed information useful to programmers
-ErrorAction	Indicates how the cmdlet responds to a non-terminating error. Possible values are SilentlyContinue, Continue, Inquire, Stop
-ErrorVariable	Stores information about errors in the specified variable
-OutBuffer	Determines the number of objects to store before sending them onto the pipeline. This is usually omitted which means that objects are sent onto the pipeline inrnediately.
-OutVariable	Stores error messages in the specified variable
-Verbose	Displays detailed information about the operation

If a cmdlet will modify the system it has another two parameters.

Table 1.3 Safety parameters

Parameter	Meaning
-WhatIf	If present this parameter causes PowerShell to output a list of statements indicating what would have happened if the command had been executed without executing the command
-Confirm	Prompts the user for confirmation before performing any action

Further information can be found using Get-Help about_CommonParameters

Having looked at cmdlets and their parameters it is time to see how we can link them together using the PowerShell pipeline. The pipeline is what makes PowerShell a really powerful shell.

1.3.2 Pipeline

The ability to pipe data from one command to another has been a standard part of shells and command line utilities for many years. DOS, the command shell in later versions of Windows and most notably UNIX\Linux shells have all had this functionality. PowerShell also has this functionality as we have seen in some of the examples earlier in the chapter.

If shells are expected to have this functionality why is there such a fuss about the ability to pipe data from one command to the next in PowerShell? All the other shells pipe text data but PowerShell pipes .NET objects.

```
Get-Process | Where-Object {$_.Handles -gt 500}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

This example shows a Get-Process cmdlet passing data along the pipeline to a Where-Object cmdlet. The Get-Process cmdlet passes one .NET object for each process that is present on the machine. We can discover which particular .NET object is being passed by using Get-Member.

Listing 1.2 Using Get-Member to view the .NET type

```
PS> Get-Process | Get-Member


   TypeName: System.Diagnostics.Process

…Listing truncated for brevity
```

The use of Get-Member shows that the Get-Process cmdlet is producing, or emitting, .NET objects of type System.Diagnostics.Process. This .NET type has a property called Handles. The Where-Object cmdlet performs a filtering operation based on the value of the Handles property of each .NET object. Any object that has a value greater than 500 for the Handles property is passed. All other objects are filtered out.

The symbol $_ is used in PowerShell to refer to the object being passed along the pipeline.

NOTE

As was explained earlier the .NET objects emitted by PowerShell objects are not necessarily identical to an object of the same type produced by a .NET program. This can be seen if the output of Listing 1.2 is compared to the list of properties and methods for the System.Diagnostics.Process that can be found at http://msdn.microsoft.com/en-us/library/system.diagnostics.process.aspx. More information on working with .NET can be found in Chapter 3.

A number of cmdlets including the Format- cmdlets and the Write- cmdlets will terminate the pipeline. If a Foreach-Object cmdlet is used it is perfectly valid to create a pipeline within the loop produced by that cmdlet.

The data that Get-Process produces is as at the time of execution. When investigating a set of data such as that referring to the running processes it is sometimes necessary to ensure that all comparisons are performed on exactly the same data. Running variants of Listing 1.2 will not suffice as the data will change between runs. In this case we can make use of a variable as shown.

```
$proc = Get-Process

$proc | Where-Object{$_.Handles -gt 500}

$proc | Where-Object{$_.CPU -gt 100}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

```
$proc |Sort-Object -Property WS -Descending |Select-Object -First 5
```

In this example we start by setting a variable, $proc, equal to the output of the Get-Process cmdlet. A $ symbol is used in PowerShell to designate a variable. If Get-Member is used on $proc it will be seen that the variable is of type System.Diagnostics.Process. It is an array of such objects. When it is passed on to the pipeline the array elements are processed one at a time as they are passed along the pipeline.

The first use of $proc is a repeat of what we saw in Listing 1.2. The second is a variant using the CPU property instead of the Handles property.

The third is use is more interesting in that we are performing a sort of the data based on the WS (Working Space) property. The output of the sort is largest to smallest as designated by the use of the –Descending parameter. The first five objects in the sorted output are then displayed. Select-Object discards the other objects.

Most cmdlets will accept input from the pipeline. There are some exceptions where this is not possible. The help file for the cmdlet will show if this is the case. The fact that the command will error will show very quickly!

NOTE

For more information on the pipeline type Get-Help about_pipeline at the PowerShell prompt.

This concludes our look at the pipeline. There will be many more examples throughout the book. Next we will look at the utility cmdlets that have made brief appearances up to now.

1.3.3 Utility cmdlets

We have seen how cmdlets can be linked together on the pipeline and how .NET objects are passed along the pipeline. Utility cmdlets are used to supply the glue used to join together the cmdlets performing the processing. They supply utility actions such as sorting, selecting and filtering. Some of the utility cmdlets have been used in the previous examples. The utility cmdlets are listed in table 1.6.

Table 1.6 Utility cmdlets and their purpose.

Utility cmdlet	Purpose
Compare-Object	Compares two sets of objects.
ForEach-Object	Performs an operation against each of a set of input objects.
Group-Object	Groups objects that contain the same value for specified properties.
Measure-Object	Calculates the numeric properties of objects, and the characters, words,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

	and lines in string objects, such as files of text.
Select-Object	Selects specified properties of an object or set of objects. It can also select unique objects from an array of objects or it can select a specified number of objects from the beginning or end of an array of objects.
Sort-Object	Sorts objects by property values
Tee-Object	Saves command output in a file or variable and displays it in the shell.
Where-Object	Creates a filter that controls which objects will be passed along a command pipeline.

The best way to demonstrate the use of these cmdlets is with some examples. Full details on the syntax and use of these cmdlets can be found in the help system. Get-Help followed by the cmdlet name will supply the required information.

Comparing

At some time when working in Windows administration it will be necessary to compare two files. They may be two different versions of scripts or configuration files but is almost certain that you will spend a long time looking them to spot the differences. It never seems obvious until you have stared at them for a long time. The time to discover the differences can be shortened dramatically by using Compare-Object.

Listing 1.3 Using Compare-Object to compare two files

```
PS> Compare-Object -ReferenceObject chap01v1.txt -DifferenceObject
chap01v2.txt

InputObject                                                  SideIndicator
-----------                                                  -------------
chap01v2.txt                                                 =>
chap01v1.txt                                                 <=
```

Compare-object is used for comparisons. PowerShell really is self describing! In this case I have two text files that I want to compare. The ReferenceObject parameter supplies the object against which comparisons will be made. DifferenceObject supplies the object to be compared.

In this example the SideIndicator shows if an object appears in the reference object (<=) or the difference object (=>). We can see that there are differences between the files but we have no idea what differences and where they occur in the file. We need to modify our script slightly in order to discover that.

Listing 1.4 Comparing the content of two files

```
PS> Compare-Object -ReferenceObject $(Get-Content chap01v1.txt) `
>> -DifferenceObject $(Get-Content chap01v2.txt)
>>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

```
InputObject                                                 SideIndicator
-----------                                                 -------------
This is line 6a                                             =>
This is line 6                                              <=
```

Notice the use of a backtick ` character. This is the line continuation character. It is used here to split the line of code onto a continuation line to make it more readable. Here we have compared the individual lines within the files so we can see exactly where the differences occur. Of special interest is the use of $(Get-Content chap01v1.txt) and $(Get-Content chap01v2.txt) when supplying the objects to be compared. The structure $() tells PowerShell to evaluate what is between the parentheses and treat that as the variable to be used. All variables in PowerShell start with the $ symbol.

By default only data that is not equal is displayed in the output from Compare-Object. If matching data is required use the IncludeEqual parameter. It will generate a lot of output though.

GROUPING AND SORTING

Storage is relatively cheap but no organization can afford to have an infinite amount of disk space. In order to make better use of the space we need to know the distribution of files on the storage. Counting the number of files of each type can give a good indication of where the space is being used especially if they are files that shouldn't be there. How many organizations have server disk space taken up by downloaded music or video files? In this case we are grouping on the file type. Any suitable property can be used.

Listing 1.5 Use of Where-Object and Group-Object to count the number of files in a folder by file extension type

```
PS> Get-ChildItem -Path "c:\temp" | Where-Object {!$_.PSIsContainer} |
>> Group-Object -Property Extension | Sort-Object Count -Descending
>>

Count Name                      Group
----- ----                      -----
   92 .tmp                      {AD~5D8C.tmp, artD5CD.tmp, artD5DE.tmp...}
   61 .cvr                      {CVR1162.tmp.cvr, CVR2463.tmp.cvr...}
   60 .od                       {12623912.od, 13136469.od, 13819442.od...}
   33 .txt                      {dd_depcheck_VS_PRO_90.txt...}
   14 .log                      {java_install_reg.log, jusched.log...}
    2 .xml                      {setup.xml, tmp713D.tmp.xml}
    2 .exe                      {msxml6-KB927977-enu-x86.exe...}
    2 .dll                      {fxdecod1.dll...}
    1 .sqm                      {wmplog00.sqm}
    1 .msi                      {Virtual_PC_2007_Install.msi}
    1 .psc1                     {powergui.script.editor.psc1}
    1 .fzip                     {ImageDecoder_2.0.2008.523.fzip}
    1 .bmp                      {INT+rsiddaway.bmp}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

We will be meeting Get-ChildItem again in Chapter 8 when we examine the file system but for now it is the PowerShell equivalent of dir or ls (both of these exist in PowerShell as aliases of Get-ChildItem). The Path parameter tells Get-ChildItem the folder to examine. The output of Get-ChildItem is piped to Where-Object which applies a filter based on whether or not the object is a container (a folder in this case). PowerShell adds a property to the output of Get-ChildItem which indicates if the object is a folder. In this case we want those objects which are not folders that is just the files

The results of the filter are piped to Group-Object which groups the files by extension. Finally we use a Sort-Object to order the output by the number of files in each group. The output gives the number of files in each group, the Name of the group (in this case the file extension) and a partial list of the group membership.

NOTE

The PowerShell pipeline used in this example is actually a single line of code. It could be typed at the PowerShell prompt and allowed to wrap around. In order to make it more readable the input has been split across multiple lines. When entering code pressing the Enter key before the command is complete in this case immediately after the pipe symbol causes PowerShell to display a continuation line as shown. Once the extra code has been typed pressing Enter twice will run the code. Code can also be split in this manner before a closing bracket or closing quote for a string value.

This script could be modified to read a folder tree by adding the recurse parameter. The script would then start Get-ChildItem –Path "c:\temp" –recurse | etc.

Measure

We used Group-Object to determine the number of files of each type earlier in this section. We can use Measure-Object to determine statistics for those files including total number and the sum of their sizes.

Listing 1.6 Use of Where-Object and Measure-Object to produce statistics on file sizes in a folder

```
PS> Get-ChildItem -Path "c:\temp" | Where-Object {!$_.PSIsContainer} |
>> Measure-Object -Property Length -Average -Sum -Minimum -Maximum
>>


Count    : 272
Average  : 366154.713235294
Sum      : 99594082
Maximum  : 29440512
Minimum  : 0
Property : Length
```

As in the previous example we use Get-ChildItem and Where-Object to produce a set of objects representing the files in a folder. This time we pipe them into Measure-Object. If we

©Manning Publications Co. Please post comments or corrections to the Author Online forum: http://www.manning-sandbox.com/forum.jspa?forumID=499

just use Measure-Object without any parameters it will return just the number of files that is the Count. By telling the cmdlet which property to measure and selecting the measurements to make we can generate the average, minimum, maximum and sum of the file length (size in bytes). The parameters indicating which statistics to measure can be used in any combination that is require. A count of the total number of items will always be produced.

Measure-Object can be used with an array of numbers. It can be applied to any numeric property but only numeric properties.

FILTERING

The Where-Object cmdlet is used for filtering. We have seen it being used in a number of the previous examples. Correct use of filtering can have a beneficial impact on your scripts as they will run faster because less data is being processed. Filtering can also make the output easier to understand.

Think of the case where a problem has arisen on a server and you need to test if the relevant service is actually running. You could just run Get-Service but that involves reading through a lot of output. A better solution is to filter on just the service or services in which you are interested. Get-Service does have a certain level of built in filtering as the parameters for service names accept wildcards. In this example I will only filter using Where-Object as I want to show how to combine filters. PowerShell supports the full range of logical operators.

Listing 1.7 Using multiple filters in Where-Object

```
PS> Get-Service |
>> Where-Object {$_.Name -like "WM*" -and $_.Status -eq "Stopped"}
>>

Status   Name               DisplayName
------   ----               -----------
Stopped  wmiApSrv           WMI Performance Adapter
Stopped  WMPNetworkSvc      Windows Media Player Network Sharin...
Stopped  WMSvc              Web Management Service
```

Get-Service returns the list of Windows services installed on the system. This list is passed into Where-Object which performs a filter on the first part of the service name and on the status of the service. This could be extended by making the service partial name an argument which was passed into the script. This will be covered in Chapter 2.

An alternative form of filtering is supplied by the Select-Object cmdlet. This is used to limit the properties that are passed down the pipeline. It can also be used to add a calculated property or select a specific number of objects from the beginning or end of the list.

Listing 1.8 Using a calculated property in Select-Object

```
PS> $now = Get-Date
PS> Get-Process | Where-Object{$_.StartTime} |
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

```
>> Select-Object Name, @{Name="Run Time";
>> Expression={[int]($now - $_.StartTime).TotalMinutes}} |
>> Sort-Object "Run Time" -Descending | Format-Table -AutoSize
>>

Name                           Run Time
----                           --------
svchost                             909
smss                                909
csrss                               909

- - output truncated

WUDFHost                            207
WINWORD                             202
Quest.PowerGUI.ScriptEditor         190
PowerShell Assistant                187
Foxit Reader                        169
notepad                             119
powershell                           12
```

This is the most complicated example we have seen so far. We start by using the Get-Date cmdlet to record the current date and time in a variable. Remember that variables always start with the $ symbol.

Get-Process retrieves a list of the running processes on the system. We filter out those processes that do not report a start time. If they are left in the script will still work but we get error messages for those processes that do not have a start time recorded. Select-Object is used to filter the properties. We are only interested in the process name and calculating the running time.

The calculated property is a hash table (see arrays in Chapter 2). It is an array with two values separated by a semi colon. The first, known as the key, is the name of the property in this case "Run Time". The second item, known as the value, is an expression to calculate the property. This calculation will happen for every process coming along the pipeline.

Once the property is calculated it can be manipulated in the same way as any other property and we can use it in a sort operation. We can see the longest running processes by sorting in a descending direction.

The final step is to use the Format-Table to output the results to screen. The autosize parameter is used to control the formatting of the columns on screen.

NOTE

Hash tables are also known as associative arrays. Details can be found in Appendix A.

This concludes our look at the utility cmdlets (Tee-Object is not used very much and its use is self explanatory). They will appear in many more scripts throughout the book. You have now learnt enough about them to follow their use in future scripts. Having completed

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

learning about cmdlets and the pipeline it is time to turn our attention to another method of working in PowerShell namely the providers.

1.3.4 Providers

Have you ever wanted a consistent method of working with multiple data stores such as the file system, Active Directory, SQL Server, IIS and the Windows Registry? PowerShell can deliver a large part of that vision through the use of providers.

The provider feature in PowerShell gives us a way of treating data stores as if they were the file system. PowerShell demonstrations where we do a dir through Active Directory or the registry always go down well. The provider exposes a data store as just another drive on your system. The following shows how to view the installed providers and the associated drives. Notice that the cmdlet refers to them as PSDrives to differentiate them from physical drives.

Listing 1.9 Viewing the installed PowerShell Drives

```
PS> Get-PSDrive | Format-Table -AutoSize

Name     Provider          Root                CurrentLocation
----     --------          ----                ---------------
Alias    Alias
C        FileSystem        C:\                         Scripts
cert     Certificate       \
D        FileSystem        D:\
E        FileSystem        E:\
Env      Environment
F        FileSystem        F:\
Feed     FeedStore
Function Function
Gac      AssemblyCache     Gac
HKCU     Registry          HKEY_CURRENT_USER
HKLM     Registry          HKEY_LOCAL_MACHINE
IIS      WebAdministration \\PCRS2
OneNote  OneNote           OneNote
Variable Variable
```

NOTE

The following drives are not part of the standard PowerShell install: Feed, Gac, IIS and OneNote.

The list includes some drives that are specific to PowerShell such as Environment which exposes the environmental variables; Function which exposes the PowerShell functions (see Chapter 2) loaded into memory and Variable which contains the variables active in your session (mixture of system and user defined variables).

An alternative way of viewing the installed providers is to use Get-PSProvider which will display the providers, associated drives and some capabilities. Get-PSProvider will display all

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

providers installed in the PowerShell session but Get-PSDrives only shows the active providers. I have a provider for Active Directory installed on my laptop but it is only active when I am connected to the network and logged on to the Active Directory domain.

In theory providers should supply access to a common set of cmdlets that enable navigation through and interaction with the data exposed by the provider. The full list can be seen by typing `Get-Help about_Core_Commands` at a PowerShell prompt. The list includes cmdlets with the following nouns Item, ItemProperty, ChildItem, Content, Location, Path, PSDrive and PSProvider.

NOTE

Not all providers supply access to all of the core commands for example the SQL Server provider does not implement the New-Item cmdlet.

A provider is navigated in exactly the same way as the file system. The full cmdlet name is Set-Location but I expect that most people will be happier using the aliases cd or chdir depending on their background (it is also much less typing!). Aliases are good things when typing interactively but should be avoided in scripts.

The core commands have aliases corresponding to DOS or UNIX commands. As a demonstration of navigating a provider try typing these commands into PowerShell one at a time.

Listing 1.10 Navigating the Registry provider

```
cd HKLM:
ls
chdir software
dir
cd microsoft
ls
cd ..
dir
cd c:
```

NOTE

cd and chdir are aliases of Set-LocationB while ls and dir are aliases of Get-ChildItem

In this example we start by navigating into the HKLM: drive (HKEY_Local_Machine). A directory listing is then produced. This process is repeated to view the software and Microsoft keys respectively. It is also possible to work with the data exposed by a provider directly for example `dir HKLM:\software\Microsoft`.

That has covered the basics of providers. We will be working with the providers again when we examine the Registry, SQL Server and IIS in more depth later in the book. The last

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

feature I want to examine is something that has been mentioned several times already namely the help system.

1.3.5 Help system

PowerShell has a set of help files that are presented in the shell as text files by using get-Help. The help system will be covered in detail in Chapter 2 when we look at learning PowerShell.

We have completed our introduction to PowerShell all that remains is a look to the future by examining some of the new features we can expect in PowerShell version 2.

1.4 PowerShell V2

PowerShell version 2 is under development. As of the time of writing the second Community Technology Preview (CTP2) is available. PowerShell V2 introduces a number of new features that will extend its capabilities. A full description of the capabilities of version 2 cannot be supplied as it has not yet been declared feature complete. The following list of major new features is subject to change as PowerShell V2 progresses through the development process.

Table 1.4 New features in PowerShell version 2

Feature	Explanation
Remoting	Enables PowerShell on the local machine to issue commands that will be executed on a remote machine or machines. PowerShell remoting requires WinRm and PowerShell v2 to be installed on the local and remote machine(s). PowerShell must be started with administrative privileges to use for running remote commands.
Background jobs	Enables PowerShell to run commands asynchronously. This facility returns the PowerShell prompt immediately rather than waiting until the command has finished. The asynchronous command runs in the background. The status of the job can be viewed and the output data retrieved when the job has completed. Background jobs require WinRm to be loaded. Commands can be run against remote machines using background jobs.
Script cmdlets	Cmdlets can now be written in PowerShell instead of needing to use a .NET language such as C#. Script cmdlets accept parameters in the same way as a compiled cmdlet.
Modules	Modules are collections of functions contained in a .psm1 file that are loaded into PowerShell as a unit. Individual functions can be made visible to the shell or remain hidden and only be accessible from other functions within the module. The cmdlet Add-Module is used to add modules into PowerShell. It can also be used to load PowerShell snapins without registering them with PowerShell.
Transactions	When making a change via a provider the change can be wrapped within a transaction so that it can be rolled back in the event of an error. CTP2 only supports

	transactions on the Registry provider.
Eventing	PowerShell can access the eventing system to work with management and system events.
PowerShell startup parameters	New PowerShell startup parameters have been added to run script files via a File parameter; run PowerShell as a single threaded application and to pass complex commands into PowerShell that require using quotes or curly braces.
Try-Catch-Finally	PowerShell V1 uses the trap and throw commands to process .NET exceptions. CTP 2 adds a Try-Catch finally block to PowerShell to bring it in line with the .NET languages. The command that may throw an exception is put it a try{} block, any exceptions are caught in the catch{} block and a finally{} block executes irrespective of whether an exception occurred.
Steppable pipelines	Enables finer control on how a script block executes
Data langauge	Enables separation of data from code in a PowerShell script
Script Internationalization	Enables the internationalization of scripts by importing files of message strings into a data section. The file to be imported is controlled by the UI culture of the system
Script debugging	The debugging facilities have been enhanced with new debugging cmdlets
New operators and automatic variables	New operators for workign with strings and new automatic variables for working with the PowerShell system have been added
New cmdlets	New cmdlets have been added for Remoting, Adding or Converting types, ETW logs, script internationalization, modules, debugging, eventing, background jobs, transactions, wmi and some miscellaneous actions
Graphical PowerShell	Graphical PowerShell is a GUI based editor with an interactive shell. It requires .NET 3.0. Graphical PowerShell can support up to 8 PowerShell runspaces (instances) and can a whole script or on ly the highlighted part.
Out-Gridview	Displays output in an interactive table that can be searched and filtered. This feature requires .NET 3.5

Other new features include:

- New parameters for existing cmdlets
- Improved tab expansion function
- Improvements to [ADSI] accelerator (see Chapter 4) and introduction of [ADSISearcher] for searching Active Directory.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

The examples in this book will be based on PowerShell version 1 as PowerShell version 2 is still very early in the development cycle. If it is applicable reference will be made to PowerShell version 2 with an indication of the expected change.

1.5 Summary

This chapter has introduced PowerShell and explained some of the fundamental concepts that are required for the following chapters.

PowerShell is the new automation engine for the Windows platform. It is available as a download or installable feature (Windows Vista and Server 2008) that supplies a command shell and scripting language. It is .NET based and has been designed for administrators. The PowerShell language contains over 130 specific commands, known as cmdlets, that enable administrators to work with the file system, registry, event logs processes and services on a Windows machine.

PowerShell cmdlets can be linked via a pipeline. Unlike other shells the pipeline passes .NET objects rather than text between cmdlets. A number of utility cmdlets to perform actions such as sorting, grouping, filtering and measuring are available and provide the "glue" for joining cmdlets on the pipeline.

Shells can usually work with the file system. PowerShell extends the concept of drives to expose other data stores such as the Registry, Certificate Store and the environment as if they were the file system. This functionality is supplied by a provider. The core cmdlets work on these providers in the same way they work with the file system.

The PowerShell help system is text based similar to the "man" pages in UNIX. Help is supplied on the individual cmdlets and on PowerShell language and environmental features.

A Community Technology Preview (CTP) for the second version of PowerShell is available for download. The new features in PowerShell version 2 are briefly outlined.

The remainder of the section will look at learning PowerShell (Chapter 2), other technologies used with PowerShell including ADSI, .NET, COM and WMI (Chapter 3) and automation including best practices around writing scripts in Chapter 4. Sections 2 and 3 consider working with PowerShell to perform administration tasks in a Windows environment.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

2
Learning PowerShell

We will have created an automation toolkit by the time we reach the end of the book. The core of that toolkit is PowerShell itself. We have looked at the fundamentals of PowerShell now we learn how to use it. Think of it as unpacking a shiny new tool, putting the bits together and now we learn how to use it.

When I downloaded the first beta of PowerShell I remember installing it, clicking on the icon to start it and then having one of those "What on earth is this?" moments. I usually mention this when giving talks about PowerShell and often get someone coming up to me and saying they had the same problem. I spent a lot of time working through the documentation, searching the Internet and experimenting through trial and (lots of) error to find how this stuff worked. This chapter will be your short cut to that learning process as I will show you the self discovery mechanisms in PowerShell as well as providing usable examples of how it works.

This is where we learn how to use PowerShell. We will look at:

- Installation and Configuration
- Self discovery
- Language features
- Scripts

We start by discovering how to install and configure PowerShell. The main configuration item is the PowerShell profile, what to put in it and where to store it. It is possible to have four profiles, though not recommended, and I will explain which to use when. Once PowerShell is configured we can start learning how to use it. This is done by using PowerShell's self discovery mechanisms. We will find four new friends along this journey.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

The learning journey continues as we dig further into the language features such as loops, branches and variables. These are the things that give us the ability to write scripts in PowerShell as well as work interactively at the prompt.

The chapter will close by looking at script development. It sometimes becomes a bit hard to justify calling a piece of PowerShell a script when it is one line of code.

VBScript has been the main Windows based administration scripting language for the last ten years. There is a huge body of administrative scripting examples available on the Internet through sites such as the Microsoft TechNet Scripting Center which can be found at http://www.microsoft.com/technet/scriptcenter/default.mspx.

WARNING

As with any download from the Internet ensure that you understand what the script is doing before trying it in your production environment. Virtual machines are a very good place to experiment with downloaded scripts.

Many administrators have created a library of scripts in VBScript that are used to perform daily administration tasks. I will show how to convert VBScript into PowerShell by working through an example. It is not feasible to instantly convert to using PowerShell, especially if you rely on VBScript at the moment. I will show you how to incorporate, and run, VBScript code in your PowerShell scripts. This enables a phased approach to be used when converting to PowerShell.

It is time to start learning PowerShell.

2.1 Open the book - learn by doing

Everyone has different ways of learning a new subject. Learning a new technology is best achieved by a mixture of theory and practice. When I give talks about PowerShell they are always very heavy on demonstrations as I believe that seeing PowerShell working and solving problems explains more than a large set PowerPoint slides. There is also the excitement for the audience of waiting to see how my demo sessions go wrong - if it can go wrong I will find it!

This book will follow the same concept with lots of examples.

I would recommend that you type in the scripts that are given as examples in the book. Most of them are short and if you use one of the editors with Intellisense that are discussed in chapter 4 the code entry will not be an onerous task. The scripts will be available for download from the associated web site at www.xxxx.yyyy.com if you prefer. I have always found that typing in the examples helps me learn.

The interactive nature of PowerShell means that we can experiment in the shell and then build what works into our scripts. One very useful trick that makes use of this ability is dot sourcing. When a script is executed in PowerShell all reference to the variables in use is lost when the script finishes. As explained in the potential issues section later in this chapter we need to give a reference to the folder by typing `.\script_name.ps1` when we want to run

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

the script. If we add an additional reference. that is we type `. .\script_name.ps1` and run the script we find that the variables are left in memory. This means we can continue to work with them interactively. I find this particularly useful when I am developing scripts and need to experiment to with how to perform a particular task or if I am trying to work out why a script does not work properly.

Having thought about how we are going to learn to use PowerShell it is time to start the learning process. Starting at the beginning we will install and configure PowerShell.

2.2 Installation and Configuration

PowerShell can be installed on most modern Windows Operating Systems:

- Windows 2003
- Windows XP Service Pack 2
- Windows Vista
- Windows Server 2008

Windows Server 2008 has PowerShell as an optional installable feature. There are downloadable versions for the other Operating Systems. PowerShell is available in 32 bit and 64 bit versions. It is not possible to install PowerShell on Windows 2000. In theory it is also not possible to install PowerShell on Windows Server 2008 Server Core but as we will see in Chapter 8 there is a method of performing the installation.

2.2.1 Installation

The only pre-requisite for installing PowerShell is that the .NET framework version 2.0 is installed. .NET 2.0 is either installed as an optional feature or the framework is downloaded and installed.

NOTE

Windows Vista and Windows Server 2008 have .NET 3.0 as the installable option. .NET 3.0 and 3.5 are supersets of .NET 2.0 so that it is automatically installed.

If required the 32bit version of .NET 2.0 can be downloaded from http://www.microsoft.com/downloads/details.aspx?FamilyID=0856EACB-4362-4B0D-8EDD-AAB15C5E04F5&displaylang=en or the 64bit version is downloadable from http://www.microsoft.com/downloads/details.aspx?familyid=B44A0000-ACF8-4FA1-AFFB-40E78D788B00&displaylang=en. In both cases the Windows Installer software must be at least version 3.0. Links to download the installer software are available on the appropriate download page. There are also links to the latest Service Pack for .NET 2.0.

PowerShell version 1 can be downloaded from the Microsoft web site at http://www.microsoft.com/windowsserver2003/technologies/management/powershell/download.mspx. Select the correct Operating System version.

©Manning Publications Co. Please post comments or corrections to the Author Online forum: http://www.manning-sandbox.com/forum.jspa?forumID=499

On Windows 2003, XP and Vista PowerShell is installed via the update mechanism. It shows in the list of updates installed on the system rather than as an individual program in its own right.

NOTE

If you install PowerShell and then install a Service Pack it may not be possible to remove PowerShell without uninstalling the Service Pack.

The actual install is a simple matter of double clicking on the installation file and following the wizard. The 32bit version of PowerShell is installed on 64bit systems as well as the 64bit version. Having successfully installed PowerShell we now need to configure it to meet our requirements.

2.2.2 Configuring PowerShell

There are a number of configuration items we can perform on PowerShell. The most common are covered in this section.

EXECUTIONPOLICY

The PowerShell execution policy determines whether scripts can be run and if they need to be digitally signed. A full description of how to set execution policy can be found under the Potential issues section. This one of the commonest issues raised by people starting to use PowerShell.

PROFILES

A PowerShell profile is a script that executes when PowerShell is first started. Profiles are used to configure PowerShell and load extra functionality automatically at start up. This could be done manually but profiles make the application of a standard, repeatable configuration much easier. It is possible to use four separate profiles in PowerShell. In order of loading the profiles are:

1. C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
2. C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
3. %userprofile\MyDocuments\ WindowsPowerShell\profile.ps1
4. %userprofile\MyDocuments\ WindowsPowerShell\ Microsoft.PowerShell_profile.ps1

NOTE

Profiles that are applied later in the sequence can override settings from earlier profiles. Avoid using multiple profiles if possible.

Profile 1 applies to all users and all shells. Most PowerShell functionality can be added into the base shell that is the standard install PowerShell. The second profile applies to all users but only to the Microsoft.PowerShell shell (an example of an alternative shell is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

PowerShell Analyzer - see the PowerShell Toolkit section in Chapter 4). Profile 3 applies to the individual user but across all shells and the final profile applies to the individual user but is restricted to the Microsoft.PowerShell shell. The rationale behind the naming and location of the profiles can be found on Lee Holmes' blog (Lee is a member of the PowerShell team) http://www.leeholmes.com/blog/TheStoryBehindTheNamingAndLocationOfPowerShellProfiles.aspx

RECOMMENDATION

I would recommend using number 3 if you are the only person using the machine for example your personal workstation. If you want the same settings to apply to all users then number 1 should be used for example on a server where several administrators have accounts and you want them all to have the same settings.

The profile locations are summarized in Table 2.1.

Table 2.1 Summary of PowerShell profile locations

	All shells	**Standard PowerShell**
All users	%Install folder%\v1.0\profile.ps1	%Install folder%\v1.0\ Microsoft.PowerShell_profile.ps1
Individual user	.. \MyDocuments\ WindowsPowerShell\profile.ps1	.. \MyDocuments\ WindowsPowerShell\ Microsoft.PowerShell_profile.ps1

%Install folder% is the path to the PowerShell installation folder. PowerShell does not create any profiles during the installation process. All profiles have to be created manually though it is possible to store the profile centrally and reference it from a profile on the local machine.

Profiles can perform several actions including:

- Loading additional functionality via PowerShell snapins
- Creating functions and storing in memory for future use
- Setting the PowerShell prompt
- Running scripts
- Setting environmental factors such as color schemes
- Change the current folder

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

A folder containing an example profile is created when PowerShell is installed. On 32bit machines it can be found at C:\Windows\System32\WindowsPowerShell\v1.0\Examples. The folder contains a file called profile.ps1.

NOTE

If you copy this file ensure that you delete the lines at the start of the file that start set-alias. These aliases are now automatically defined and attempting to redefine them in a profile will cause a lot of error messages.

A sample profile is provided below (part of my standard profile).

Listing 2.1 Sample Profile

```
$host.privatedata.ErrorBackgroundColor = "White"                              1

cd c:\scripts                                                                 2

Add-PSSnapin Pscx                                                             3
Add-PSSnapin Quest.ActiveRoles.ADManagement                                   3


function help                                                                 4
{
     clear-host
     get-help $args[0] -full
}

function man                                                                  4
{
    get-help $args[0] -full | out-host -paging
}

function mkdir                                                                4
{
    new-item -type directory -path $args
}

function md                                                                   4
{
    new-item -type directory -path $args
}


function prompt                                                               5
{

   $host.ui.RawUI.WindowTitle = $(get-location)
   "PS> "
}
```

1 Set error color
2 Change location
3 Add extra functionality

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

4 Create standard functions
5 Create prompt

The profile starts by setting the background color for error messages. 1 Typing `$host` at a PowerShell prompt displays a number of PowerShell configuration items. `$host.privatedata` displays all of the foreground and background colors. These items can be changed.

NOTE

Setting colors like this in your profile will generate an error when you open the Graphical PowerShell console in CTP 2. It does not stop anything working.

The folder location is then changed. 2 One of the main uses for a profile is to load additional functionality into PowerShell using the `Add-PSSnapin` cmdlet. 3 The functions shown at 4 create pre-defined commands, in this case mimicking standard commands from DOS and UNIX. The final function 5 performs two tasks. It defines the PowerShell prompt to be "PS> " rather than the full path of the current folder. It also sets the title of the shell window to be the full path to the current folder. This will change as the location is changed in the file system or any other provider. A quick search on the Internet will reveal other prompt functions.

CONSOLE FILES

An alternative method of configuring PowerShell is to use a console file. A number of products such as Exchange 2007, the IIS 7 PowerShell provider and the Quest Active Directory cmdlets create console files as part of the installation process. It is also possible to create a console file for a PowerShell session using the `Export-Console` cmdlet. An example console file is shown in Listing 2.2.

Listing 2.2 Sample PowerShell Console file

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>1.0</PSVersion>
  <PSSnapIns>
    <PSSnapIn Name="IIsProviderSnapIn" />
  </PSSnapIns>
</PSConsoleFile>
```

The working part of the console file is the line <PSSnapIn Name="IIsProviderSnapIn" /> which causes the snapin to be loaded. PowerShell can be started using a console file for configuration like this powershell.exe -PsConsoleFile MyConsoleFile.psc1.

If you install one of the above mentioned products check the console file to see what you should add to your profile.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

2.2.3 Extending PowerShell

PowerShell's extensibility is a major strength that helps enable third party vendors and the PowerShell community to thrive. There are an increasing number of PowerShell related products available commercially and through community based projects. It is preferable to have a single shell incorporating all of the functionality you require rather than spreading the functionality across a number of different PowerShell shells.

PowerShell is extendable in a number of ways:

- Create functions that are loaded at PowerShell start up
- Create new cmdlets in a PowerShell snapin
- Create a provider in a PowerShell snapin
- Create a new shell incorporating the extra functionality.

We saw an example of creating functions in the previous section. New cmdlets or providers are created as a PowerShell snapins in a .NET language and complied into a dll. An installation package is created to install and register the dll with PowerShell.

All registered snapins are visible by typing `Get-PSSnapin -Registered`. Snapins can be loaded using `Add-PSSnapin` as shown in the sample profile and removed using `Remove-PSSnapin`. To view the installed snapins use `Get-PSSnapin`.

2.2.4 Potential issues

You have now installed and configured PowerShell and want to start using it to solve your administration problems. You are sat at the keyboard ready to start working with PowerShell. What issues are we likely to meet?

EXECUTION POLICY

The first issue is that when you first install PowerShell you cannot run scripts. Even the profile scripts will not execute. You are permitted to work interactively at the PowerShell prompt. In some cases you may want PowerShell left in that state however most of the time it is required to run scripts. The ability to execute scripts is controlled by the PowerShell execution policy. This is a deliberate design decision by the PowerShell team. Their assumption is that 80% of PowerShell users will be working interactively rather than scripting.

The setting for the execution policy can be found in the Windows Registry key ExecutionPolicy at HKLM:\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell. When PowerShell is first installed it is set to Restricted. The possible values for ExecutionPolicy are shown in the table.

Table 2.2 Possible values for PowerShell Execution Policy

Value	Meaning

©Manning Publications Co. Please post comments or corrections to the Author Online forum: http://www.manning-sandbox.com/forum.jspa?forumID=499

Restricted	This is the default setting. It does not allow profile scripts to be loaded or scripts to run.
AllSigned	All scripts and profiles must be digitally signed using a code signing certificate from a trusted certificate authority. This also applies to scripts created and executed on the local system.
RemoteSigned	All scripts or profile files downloaded from the Internet must be digitally signed. This also applied to scripts from network shares.
Unrestricted	All profile files and scripts will be executed regardless of origin.

The ExecutionPolicy value can be changed by editing the Registry and changing the value to the appropriate setting. A better way is to perform this task from PowerShell.

NOTE

On Windows Vista and Windows Server 2008 PowerShell must be started using the "Run as Administrator", that is with elevated privileges, if you intend to change the execution policy. On Windows XP and 2003 the user that started PowerShell when changing the execution policy must have administrator privileges. PowerShell cannot request that an elevation of privileges is performed once it is running.

The current setting for the execution policy can be viewed using the Get-ExecutionPolicy cmdlet. Only the value of the setting is returned. The setting can be changed using Set-Execution Policy

Listing 2.3 Using Set-ExecutionPolicy

```
PS> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
PS> Get-ExecutionPolicy
RemoteSigned
PS>
```

Notice that nothing is returned to indicate a successful change of execution policy. The Get-ExecutionPolicy cmdlet has to be used to view the setting.

What is the recommended setting for execution policy? My recommendation would be to set the execution policy as high as possible without compromising the ability to perform the required administrative tasks. The UnRestricted setting should be avoided. Ideally AllSigned would be used but if you do not have access to a code signing certificate then use RemoteSigned. Restricted can be used if you want to allow a user access to run PowerShell interactively but not run scripts. Ensure the user does not have Administrator level privileges on the system otherwise they will be able to change the execution policy.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

Double click

Many Windows executables can be started by double clicking the file in Windows Explorer. This includes .exe files, batch files and VBScript files. If a PowerShell file is double clicked it will not execute. The default behavior is for the file to be opened in Notepad.

RECOMMENDATION

It is strongly recommended that this default behavior is NOT modified to allow a PowerShell script to be executed by double clicking.

This default behavior may be overridden by applications such as PowerGUI. The settings are changed so that the files are opened in the PowerGUI editor rather Notepad.

The blocking of script execution by double click is an intentional behavior that is viewed as a security feature in PowerShell.

Current folder

In PowerShell the current folder is not on the search path. The contents of the search path may be viewed by typing `$env:path` at the PowerShell prompt. This means that a PowerShell script file in the current folder cannot be executed simply be executed by typing the name of the script.

In order to execute a PowerShell script in the current folder use `.\script_name.ps1`. The option `./script_name.ps1` also works. This is the option I tend to use as it is quicker to type at the prompt.

NOTE

It is strongly recommended that the current folder is NOT added to the search path.

This is another deliberate configuration decision by the PowerShell team to help prevent the accidental or malicious running of scripts that could have an adverse effect on your system.

.NET

As stated earlier PowerShell is .NET based. It opens the whole of the .NET framework for use in your scripts or from the command line. One small issue is that the whole of the .NET framework is NOT loaded into PowerShell by default.

There are a number of methods of loading .NET assemblies into PowerShell. These methods will be explored in Chapter 3 when we look at .NET and PowerShell.

Nothing returned

Sometimes nothing appears to happen when you run a PowerShell cmdlet as seen in the earlier example using Set-ExecutionPolicy. This could be for two reasons. Firstly, there may not be any data to act upon so the cmdlet cannot perform the designated action. Secondly, the cmdlet may not return anything.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

The behavior can be very frustrating sometimes. If this happens and you are unsure as to what is happening then try and perform an independent check on the data to make sure that the cmdlet did perform as expected. Also, check if the cmdlet should actually return anything. This information can be found in the help file for the cmdlet.

This completes our look at configuring PowerShell. Using this information you should be able to install and configure PowerShell to meet your requirements. Time now to read the instructions and learn how to use our shiny new tool.

2.3 Your four best friends

We have looked at how to install and configure PowerShell and now we discover how to get over the "What on earth is this?" moment that I mentioned earlier. There is a certain amount of documentation delivered with PowerShell consisting of:

- Getting Started Guide
- Quick Reference Guide
- User Guide
- Release Notes

The documentation has some good examples and is well worth reading. If, like me, you want to use the product and then read about it then the PowerShell discovery mechanisms are designed just for that purpose.

PowerShell has an inbuilt discovery mechanism - you can use PowerShell to discover how to use PowerShell. The way that this sort of thing has been thought through and designed into the product is what makes a lot of people really think that PowerShell Rocks!

I have called this section "Your four best friends" and when using PowerShell there are four tools that you will come to rely on:

- Get-Help
- Get-Command
- Get-Member
- Get-PSDrive.

We will look at these in turn and discover how they help us learn PowerShell and how they continue to help use when we are working with PowerShell. When something is not working or you cannot work out how to do something the chances are that one of these tools will solve your problem. There are a number of code snippets in this section. They are there for you to try and look at the results. PowerShell is best learnt by doing. We will keep meeting these friends in later chapters.

We will visit each of our new friends in turn, starting with Get-Help.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

2.3.1 Get-Help

The first port of call when trying to discover how something works is the help system. PowerShell supplies help as files that are displayed as text in the shell. Help information is either stored in XML files related to the dll containing the PowerShell cmdlets or in text files stored with the PowerShell binaries. The PowerShell help system is text based and is analogous to the man files available on UNIX systems.

Two sets of help are available:

- Information specific to an individual cmdlet accessed by Get-Help followed by the cmdlet name
- PowerShell language and environment information accessed by Get-Help followed by about_topic_name.

Typing `Get-Help` returns the help file for Get-Help itself. If a cmdlet name is appended for example `Get-Help Get-Command` then a brief set of information is displayed including:

- Name of the cmdlet
- Synopsis - one sentence statement of what it does
- Syntax description - how we use it
- Detailed description
- Related Links pointing to related cmdlets or about files
- Remarks - usually a reminder about the -full and detailed parameters.

Using `Get-Help -detailed` adds information on the parameters available for the cmdlet and examples of how to use the cmdlet. The -full parameter returns more information on the cmdlet's parameters, some notes on the cmdlet and information about the input and output data.

To see the range of about files available type `Get-Help about` at the PowerShell prompt. The list of files includes language features such as the if statement and the looping commands as well as information about the PowerShell environment such as variables and the pipeline.

The sample profile discussed in Chapter 1 contains two functions that are worth copying into your profile. One defines a function called help and the other called man. The functions as presented are identical in that they take the output from a Get-Help call and pipe it into the Out-Host cmdlet which uses the paging parameter to generate a paged output. Again notice the use of DOS and UNIX commands used for the aliases.

NOTE

If you want to see the full help information you will need to change the code in these functions to read get-help $args[0] -full | out-host -paging. The example profile was produced during the beta process and not updated when the full and detailed parameters were added to Get-Help. Modified versions are shown in Listing 2.1

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

Using the help system can be a bit awkward when you are in the middle of working through a problem especially if you want to keep the screen relatively clear. One solution is to use two PowerShell shells. The first shell is your working prompt where you are running scripts or working interactively. A second PowerShell instance is used to display help information as required.

An alternative method of accessing the help information is to download the Graphical Help File from http://www.microsoft.com/technet/scriptcenter/topics/winpsh/pschm.mspx. Install the compiled help file into a folder of your choice and add the Get-GuiHelp function given on the web page to your profile. When you have restarted PowerShell you will then be able to type Get-GuiHelp Format-Table at the

PowerShell prompt and the graphical help system will open at the correct place as shown in Figure 2.1. Tab completion does work with the function or you can create an alias. If a non-existent command is entered the graphical help file will open and an error message will be provided. The help topics can be browsed using the controls in the left hand pane. The graphical help file contains the cmdlet and about file help documentation for PowerShell version 1. In addition there is a Beginner's Guide with articles from Microsoft's Scripting Guys plus information on converting VBScript to PowerShell and some of the early Weekly PowerShell Tips. Well worth the download time.

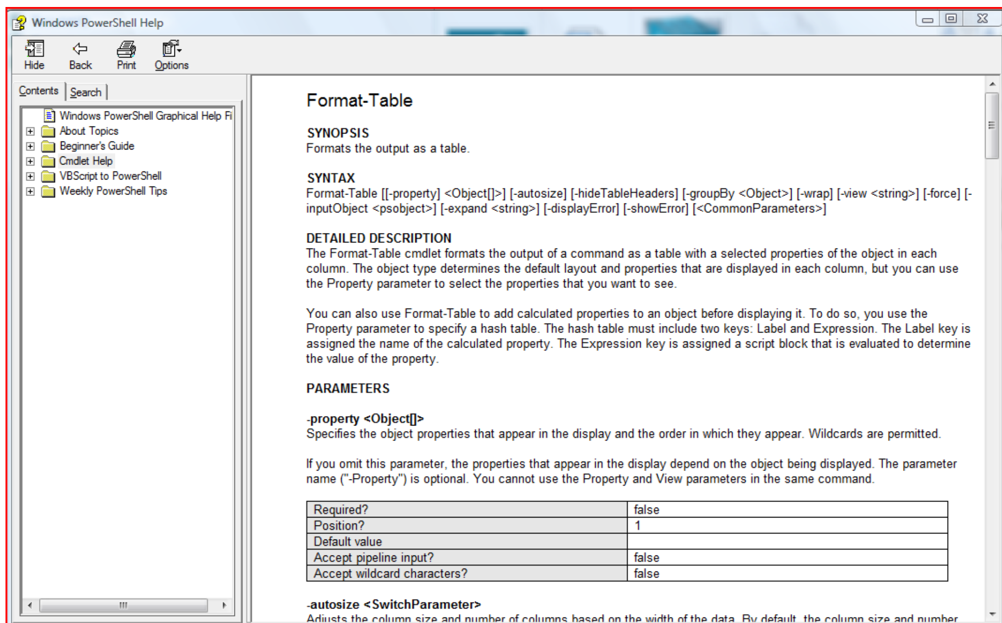

Figure 2.1. PowerShell Graphical Help File. The help topics are browsable using the tree control in the left

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

hand pane.

PowerShell version 2 supplies a similar sort of graphical help system through the graphical editor.

Get-Help gives us a lot of information about PowerShell and the cmdlets but it is not the whole story. The next friend we need to visit is Get-Command.

2.3.2 Get-Command

Get-Command is complementary to Get-Help. If we type `Get-Help Get-Command` into PowerShell prompt the synopsis states "Gets basic information about cmdlets and about other elements of Windows PowerShell commands". So how is that different from Get-Help? Get-Help gives help for those things that have help files in PowerShell. Get-Command gives data about those things that can be executed including cmdlets, functions, scripts and even Windows executables - try entering `Get-Command ipconfig.exe | Format-List` and see what is returned.

Get-Command is useful in a number of ways. You know you want to work with the processes on the computer but cannot remember the cmdlets then use

```
Get-Command *process
```

If you want to retrieve the names of the cmdlets in a particular snapin

```
Get-Command -PSSnapin IIsProviderSnapIn | Sort verb, noun
```

The last common use of Get-Member is to provide a quick reminder of the syntax used by a particular cmdlet

```
Get-Command Get-Process -Syntax
```

Get-help and Get-Command provide us with information about PowerShell and PowerShell commands. We are always dealing with objects in PowerShell. When we want to discover things about objects we turn to Get-Member.

2.3.3 Get-Member

Get-Member is best described as the "Swiss Army Knife" of PowerShell. It seems to be the one tool for which I am always reaching. It would be worth reading the help file for Get-Member. You need to type `Get-Help Get-Member` to display it. Get-Member gets information about objects. PowerShell cmdlets return objects so we can use Get-Member to view those objects.

If we use the Get-Process cmdlet to return information about the running processes whose names start with "c"

```
PS> Get-Process c*


Handles  NPM(K)   PM(K)      WS(K) VM(M)   CPU(s)     Id   ProcessName

-------  ------    -----      ----- -----   ------     -- -----------
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

```
    820       6     1772       5352     96              620 csrss
    613      14     3180      19288    193              684 csrss
    372      10    11452      17344     98     1.53    5400 cssauth
    136       7     3604       6580     77             1724 cvpnd
```

Notice that that only eight properties are shown for each process. If you have looked at processes in Task manager you will know there are a lot more properties available. We use Get-Member to view them.

Typing `Get-Process c* | Get-Member` will return more information than can easily be displayed on a page so I will leave that for you to try. We can use Get-Member to only display the property names as follows.

Listing 2.4 Using Get-Member to view object properties

```
PS> Get-Process c* | Get-Member -MemberType Property | Format-Wide
-Column 2


BasePriority                          Container
EnableRaisingEvents                   ExitCode
ExitTime                              Handle
HandleCount                           HasExited
Id                                    MachineName
MainModule                            MainWindowHandle
MainWindowTitle                       MaxWorkingSet
MinWorkingSet                         Modules
NonpagedSystemMemorySize              NonpagedSystemMemorySize64
PagedMemorySize                       PagedMemorySize64
PagedSystemMemorySize                 PagedSystemMemorySize64
PeakPagedMemorySize                   PeakPagedMemorySize64
PeakVirtualMemorySize                 PeakVirtualMemorySize64
PeakWorkingSet                        PeakWorkingSet64
PriorityBoostEnabled                  PriorityClass
PrivateMemorySize                     PrivateMemorySize64
PrivilegedProcessorTime               ProcessName
ProcessorAffinity                     Responding
SessionId                             Site
StandardError                         StandardInput
StandardOutput                        StartInfo
StartTime                             SynchronizingObject
Threads                               TotalProcessorTime
UserProcessorTime                     VirtualMemorySize
VirtualMemorySize64                   WorkingSet
WorkingSet64
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

Get-Member will also display the Methods on the object as well as properties specifically added by PowerShell.

There is a default formatter for each cmdlet that determines what will be displayed. The utility cmdlet Select-Object that we saw in Chapter 1 is used when non-default properties are to be displayed. Alternatively the Format- cmdlets can be use to select what will be displayed.

Last but not least is Get-PSDrive which gives us information about the PowerShell providers.

2.3.4 Get-PSDrive

Get-PSDrive displays information about the providers that are currently installed. Full details were supplied in Chapter 1 where Providers were discussed as part of the PowerShell feature set.

You can now install, configure and learn how to use PowerShell and you have learned to use it. Now you can start writing scripts. Oh you don't know the PowerShell language. Okay we will look at that next then.

2.4 Language features

PowerShell works as a scripting language as well as an interactive shell. So far we have been using PowerShell interactively. In this section we will look at the major features of the language.

NOTE

This is not a complete guide to the PowerShell language. The section will supply the information required to start scripting and to understand the scripts that will be presented later in the book. I would recommend Bruce Payette's PowerShell in Action for complete coverage of the language.

All scripting languages need to be able to perform a common set of actions:

- Store data – variables and arrays
- Repeat actions – loops
- Control logic flow – branches
- Reuse sections of code – functions
- Output results

They also need to be able to accept input. Typing at the PowerShell prompt has been covered and we will see lots of examples of reading files later in the book especially in Chapter 8. We will start by considering variables.

As we get into the loops and branching you will see that there is a common structure to many PowerShell commands.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

```
command(condition){script block of actions}
```

A command is followed by a condition (test of some kind) which causes a script block to execute.

2.4.1 Variables

A variable is a store for data that will be used during the execution of a script. What can a variable store? Variables store .NET objects or collections of objects.

PowerShell variables are typed that is they store a particular type of data based on the .NET object. Try the following code.

```
$a = 1
$a | Get-Member
$b = "1"
$b | Get-Member
```

We have defined two variables $a and $b. PowerShell variables are designated by a $ symbol. Get-Member gives us information about the object, in this case the variables. It will report that $a is TypeName: System.Int32 while $b is TypeName: System.String. PowerShell will assign the type based on the data you input. Probably the most common variables to deal with are integers and strings. If the type should change because of the code then PowerShell will perform the change if it can based on the .NET rules for changing (casting). For example

```
$a = $a * 2.35
$a | Get-Member
```

Results in $a being TypeName: System.Double.

NOTE

Instead of writing $a = $a * 2.35 we could use $a *= 2.35. There are similar operators for performing addition, subtraction and division. They are explained in the Operators section in Appendix A

PowerShell has a number of predefined variables. They can be viewed by using `Get-ChildItem variable:` (Yes, variables can be accessed as a drive). Help information on variables can be gained from:

- About_Automatic_variables - variables created and maintained by Powershell
- About_Environment_variables - Windows environment variables as used in batch files
- About_preference_variables - variables the customize PowerShell
- About_Shell_variable

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

One automatic variable that causes a lot of confusion is $_. It represents the object coming down the pipeline. We saw it in the Where-Object statements we used in Chapter 1. Another example would be

```
Get-ChildItem "c:\temp" | Where-Object {$_.Length -gt 1MB}
```

We use Get-ChildItem to generate a directory listing of the c:\temp folder. The output from Get-ChildItem is an object of the System.IO.FileInfo class (type) for each file in the folder. Where-Object is then used as filter to only pass those files whose length (size) is more than 1 megabyte using MB as discussed earlier.

Variable names can be made from letters, numbers and a few special characters including the underscore character _.

A variable stores a single value. Sometimes we want to store multiple values in which case we need an array which is next on the menu.

2.4.2 Arrays

An array is used to store a collection of objects. In an earlier example we saw an array created that contained System.IO.FileInfo objects (the directory listing). Arrays can be created directly as follows.

```
$a = 1,2,3,4,5
$b = 1..5
$c = "a,b,c,d,e"
$a = @(1,2,3,4,5)
```

The first example creates an array containing the integers 1-5. The second example does the same except we use the range operator instead of typing each value. In the third example we create an array of containing strings. The last example shows a variation on the first example in which @() is used to explicitly define the array. This leads to the possibility of creating an empty array and adding values to it

```
$a = @()
for($i=1,$i-le5;$i++){$a += $i}
```

Here we create an empty array and then use a loop to populate the elements. Note the use of += to add the value.

Array values can be accessed by the element number (index) which starts at 0 so the third element in our first example is $a[2].

NOTE

Startig at 0 can cause problems for users coming from scripting languages that start at 1. This is something to be aware of and check if you get unexpected problems with an array index.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

A second type of array is the associative array which is equivalent to the dictionary type object in VBscript. The associative array is normally referred to a hash table in PowerShell. Hash tables consist of key - value pairs for example

```
PS> $h =@{"one"=1;"two"=2;"three"=3}
PS> $h

Name                           Value
----                           -----
two                            2
three                          3
one                            1
```

Notice that when we dump the values they are not in the order in which they were input into the hash table. The hash table creates a hash index of the names so that it can search on the name.

```
PS> $h["two"]
2
```

This can make some activities such as sorting the hash table apparently impossible. In this case we need to use the GetEnumerator method to enable sorting to work.

```
$h.GetEnumerator() | sort value
```

This gives us enough information on arrays to start working with them. Further information can be found in the help files about_array and about_associative_array. Next on the menu is decision making and the branching statements in PowerShell.

2.4.3 Branches

Branching involves making a decision based on the relationship between two objects or an objects value. There are two PowerShell commands used for branching. The if statement is used for the situation where you want to create a test and perform actions based on the outcome of the test. The other branching command is the switch statement which is used where there are multiple outcomes usually based on the value of a variable.

IF STATEMENT

An if statement is the simplest kind of branch. A test is performed which is evaluated to true or false. If the result is true one set of actions is performed but if it is false another set of actions is performed. This script could be used to delete the largest files in a folder if you needed to create free space quickly.

Listing 2.5 Using if statement to test file size

```
Get-ChildItem "C:\Temp" | Where-Object{!$_.PsIsContainer} | ForEach-Object
{
    if ($_.Length -gt 1MB) {                                               1
           Remove-Item $_.Fullname -WhatIf                                 1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

```
    }                                                                      1
    elseif ($_.Length -gt 0.5MB){                                          2
            Write-Host $_.Name, $_.Length, " will be removed if more space
required" -ForegroundColor Yellow                                         2
    }                                                                     2
    else{                                                                 3
            if ($_.Length -gt 0.25MB){
                    Write-Host $_.Name, $_.Length, " will be removed in
third wave" -ForegroundColor Blue
            }
    }                                                                      3
}
```

1 test if greater than 1MB
2 test if greater than 0.5 MB
3 default test

We start by using Get-ChildItem to generate a file listing. We are not interested in the subfolders for this example. If you want to include the subfolders then use the -Recurse switch with Get-ChildItem. The directory listing is piped to Where-Object which filters out the containers. We are only concerned with files. We then use ForEach-Object to examine each file in the listing.

The first if statement 1 checks if the file is larger than 1MB. File size is stored in the length property. If the file is larger than 1MB then we delete it with Remove-Item. Notice that I have used the -whatif parameter. This tells which files will be removed without performing the action. That way I can keep using the files for other examples. Once you are happy with the way the script works then take off the -whatif. It could be replaced by a -confirm if you want to be prompted for each deletion.

Any file that is less than (or equal to) 1MB in size drops through to the elseif statement 2. The elseif tests if the file is more than half MB in size and prints a warning that we will get it next time round when we need more space.

Finally we drop into the else statement 3. This is the catchall for everything that doesn't get caught by the preceding tests. In this we nest another check for a possible third wave of deletions.

It is possible to have multiple elseif statements but it is usually better to use a switch statement instead which is what we will look at next.

SWITCH STATEMENT

The if statement that we used in the previous section is very good when you have a small number of decision points. When there are a large number of decision points and the syntax of the if statement would get cumbersome then we can use a switch statement. This listing shows a switch statement being used to check the size of a file so that the information can be printed to screen in the appropriate color.

Listing 2.6 Using a switch statement with color display of file size

```
$files = Get-ChildItem "C:\Temp" | Where-Object {!$_.PsIsContainer}       1
foreach ($file in $files){                                                2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

```
    switch ($file.Length){                                             3
            {$_ -gt 1MB}{Write-Host $file.Name, $file.Length           4
                           -ForegroundColor Red; break}
            {$_ -gt 0.5MB}{Write-Host $file.Name, $file.Length
                          -ForegroundColor Magenta; break}
            {$_ -ge 0.25MB}{Write-Host $file.Name, $file.Length
                          -ForegroundColor Cyan; break}
            #default {Write-Host $file.Name, $file.Length}             5
    }
}                                                                      2
```

1 Create array of file information
2 Loop through files
3 Start of switch
4 Check file size
5 Default

The output of Get-ChildItem (a directory listing) is stored in the variable $files 1. $files is a collection of objects (array) so we can iterate through them using a foreach loop 2. The switch statement uses a variable, in this case the file length property 3, as its test. We define a number of tests in this case:

- Greater than 1MB 4
- Greater than 0.5MB
- Greater than or equal to 0.25MB

When a file matches the particular criteria the associated script block is executed. We use Write-Host to write the file name and size to screen and set text color based on the size. Notice the ; symbol. This signifies a line break. We use it to concatenate multiple PowerShell commands on one line. The break command tells switch to terminate processing and not to perform any of the other tests.

A switch statement usually has a default clause to catch anything that hasn't been processed previously. In this case we don't want to see the bulk of the files displayed so the line is commented out by a # symbol.

We have mastered the use of variables and arrays and have seen a number of loops in use. It is time to consider PowerShell's looping commands in more detail.

2.4.4 Loops

We often need to repeat the same code. We often need to repeat the same code. That is bad enough to read (never mind write) so what would it be like if we had to use the same piece of code ten times or 50, 100 or more. This is where loops come to our rescue. There are a number of looping mechanisms in PowerShell:

- Foreach-Object cmdlet
- Foreach loop
- For loop
- Do loop

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

- While loop

They serve slightly different purposes and I will explain when it is best to use each one as we go through them. When using the foreach, do, for or while loop you can use the break command to jump out of the loop and the continue command to skip processing the script block and return to the start of the loop. Details in the about_break and about_continue help files.

NOTE

The PowerShell help system does NOT contain information on Do loops. The others are covered by about files.

Foreach-Object is the odd one out among the looping mechanisms because it is a cmdlet rather than a PowerShell language feature. Just to show that we don't that against it we will start with it.

Foreach-Object

Foreach-Object is used in a pipeline to perform some processing on each object coming along the pipeline. As with all things PowerShell the workings of this cmdlet are best shown by an example.

Listing 2.7 Using the Foreach-Object cmdlet to count file distributions

```
Get-ChildItem "c:\Temp" | ForEach-Object  `                    1
     -Begin {                                                   2
             $count = 0                                          2
             $total_size = 0                                     2
             $count_big = 0                                      2
             $total_size_big = 0                                 2
     } `                                                        3
     -Process {                                                 3
             if ($_.Length -gt 1MB) {                            4
                     $total_size_big += $_.Length     4
                     $count_big ++                     4
             }                                         4
             else {                                    4
                     $total_size += $_.Length           4
                     $count ++                         4
             }                                         4
     } `                                                 3
     -End {                                              5
             Write-Host $count, `
             " files are less than 1MB and occupy ", `
             $total_size_big, " bytes "
             Write-Host $count_big, `
             " files are more than 1MB and occupy ",`
             $total_size, " bytes "
     }                                                   5
```

1 Start of Foreach-Object loop
2 Begin script block
3 Start and end of process script block

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

4 If-else decision block
5 Start and and end of End script block

NOTE

The script is split across multiple lines by using the ` continuation character. This makes it more readable both in the book and when editing.

We start our script by using Get-ChildItem to generate a directory listing as in previous examples. 1 This is piped into the Foreach-Object cmdlet. The Begin, Process and End parameters are used to define script blocks which perform some actions before the first object is processed; as every object is processed and after the last object is processed respectively. Very often only a single script block is seen and that is, by default, the process block.

NOTE

A script block is some PowerShell code surrounded by {} that can be used in loops, if statements, assigned to variables, other cmdlets and even directly in the pipeline.

The -Begin script block is used to set the initial value of some variables. It is executed once when the first object enters the loop. In traditional scripting languages this would have to happen outside of the loop. Each object coming down the pipeline goes through the -Process script block. 3 In this case it goes through an if statement to decide if the file is bigger than 1MB. 4 Depending on that outcome the appropriate counter and total file size variables are updated.

After all of the objects have been processed the -End script block is executed. The results are written to screen using the Write-Host cmdlet which is described in more detail in the section dealing with output later in this chapter.

`Get-Help Foreach-Object -full` will supply the help information for this cmdlet.

Now we move on to the foreach loop which has a similar name and does similar things. Confused? You won't be after reading the next section.

FOREACH LOOP

A foreach loop is similar in concept to the Foreach-Object we have already looked at. So why do we need two things nearly the same? Foreach is part of the PowerShell language rather than being a cmdlet. Its purpose boils down to down a bunch of stuff in the following script block to every object in this collection of objects. This is best explained by an example. We could use the previous example and just rewrite it but we'll do something else to increase the examples. If you want try and rewrite the script in the previous example you can check your answer against the example on the associated web site.

NOTE

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

> One of the unwritten rules of PowerShell is that if you can get you code down to one line - the "one-liner" of much discussion on blogs and forums - then you are a real PowerShell user. This leads to some horrible examples of the over use of aliases and in some cases inefficiencies. I have produced a one line version of the following listing to demonstrate how it can be done. While it is does demonstrate the flexibility and strength of PowerShell in being able to use it as a "one-liner" it isn't always the best answer for administrative scripting as understanding and speed may be lost.

One important difference is that Foreach-Object will process objects as they come along the peipline but a foreach loop needs the objects to be already available in a collection. Testing has suggested that a foreach loop is faster than using the cmdlet version but if you have a huge number of objects the cmdlet's ability to proccess the objects as presented will be a bonus.

If you want try and rewrite the script in the previous example using a Foreach-Object cmdlet or as a "one-liner" you can check your answer against the example on the associated web site.

Listing 2.8 Using a foreach loop

```
$date = (Get-Date).Adddays(-10)
$files = Get-ChildItem "c:\Temp" | Where{!$_.PSIsContainer}
foreach($file in $files){                                        1
     if ($file.LastAccessTime -lt $date){
            Remove-Item -Path $file.FullName -WhatIf
     }
}                                                                2
```

1 Start loop
2 End loop

This script loops through all of the files in the C:\Temp folder and checks if they have been accessed in the last ten days. If they haven't been accessed then they are deleted. The script starts by using Get-Date to retrieve the current date. We then use the AddDays method to subtract ten days from the current date. Yes it does seem odd but the alternative is more complicated. The next line is interesting in that we create a variable called $files and set its value equal to the output from a pipeline consisting of `Get-ChildItem "c:\Temp" | Where{!$_.PSIsContainer}` which is capable of being run as a standalone piece of PowerShell. Try it. In the pipeline we start with Get-ChildItem producing a directory listing of the C:\Temp folder. That is piped into a Where-Object filter that looks to see if the object is a container that is it's a folder. Only objects that aren't folders are passed.

The $files variable contains all of the output from the pipeline so is actually a collection of objects rather than a single object. So we can use the foreach loop as shown. The $file variable is completely arbitrary it could have been $xxx. It is usual to make the collection's

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

name plural and the object singular. If the names reflect the objects being processed, in this case files, it aids understanding but is not mandatory.

Inside the loop we check the LastAccessTime of each file against the date we calculated at the beginning of the script. Note that we use less than (-lt) for the comparison. Dates are assumed to be increasing. If the file is older than 10 days we use Remove-Item to delete it. Notice I have a -Whatif parameter on the Remove-Item. This is safety mechanism while developing and testing the script to prevent accidents and to leave the files available for further tests. When you are ready to move the script into production remove the -WhatIf parameter. You could use the -Confirm parameter instead in production if you want to have a double check on the files before deletion.

The Foreach-Object that we used in the previous section is an alias for the foreach loop that can be used in a pipeline. Full details on using from `Get-Help about_foreach`.

Having looked at a couple of options for looping through a collection of objects we will now look at the straight forward coding loops.

FOR, WHILE AND DO LOOPS

The for loop is the classic "perform this set of code x number of times" type of loop that has been in programming languages since the year dot. The syntax borrows heavily from the C# language. While and do loops keep repeating a script block until some condition is met. The following example uses a simple for loop to create some test files.

Listing 2.9 Creating test folders and files with loops

```
$data = 1..57
$j = 1
while ($j -le 10){                              1
     $foldername = "Testfolder_$j"
     New-Item -ItemType directory -Name $foldername
      $j++

     for ($i=0; $i -le 10; $i++){               3
             $filename = "file_$i.txt"
             Set-Content -Path "$foldername\$filename" -Value $data
     }                                          4

}                                               2
```

1 Start while loop
2 End while loop
3 Start for loop
4 End for loop

We start by creating an array containing the values 1 to 57. This is just arbitrary data that is easy to create. A variable $j is created to be a counter. An outer loop, using a while command is used to create some test folders. 1 2 New-Item is the standard cmdlet for creating objects. It is one of the core cmdlets that should be enabled to work in all providers (the SQL Server provider doesn't enable this - see Chapter 14). All we need to do is pass it the type of object and the name of the object as shown. If you want to create the object

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

anywhere but the current folder then the path needs to be defined as well. After creating the folder we increment the counter by one.

We then use a for loop to create some test files. 3 4 The part in () sets the conditions and counters for the loop:

- Starting point - $i=0
- End Point - $i -le 10
- Counter increment - $i++

$i is an arbitrary counter. The fact that many people use $i is a hangover from FORTRAN where i was automatically defined as an integer and was often used as a counter in loops. The initial value of $i is 0. We increment $i by one on successive iterations of the loop until we get to a point where $i is equal to ten.

NOTE

In a for loop or a while loop the test occurs before the script block is executed. So in this case when $i is 10 the block is executed but at the next iteration $i has increased to 11 so the condition is exceeded and the loop stops.

The script block is executed during each loop. A file name is created by substituting the counter variable into the file name so we end up with file_0.txt, file_1.txt etc. We then use Set-Content to write the array into the file. One of the properties is Set-Content is that it will create a file that doesn't exist. We can exploit that mechanism here to create and write the content to the file in one line.

Time now to turn our attention to another loop type - the do loop.

The do loop suffers a little bit because there isn't any official documentation supplied for it. All the other looping mechanisms get an about file - not the poor do loop Another reason for its underuse is that it can be awkward to use sometimes.

```
$i = 1
do {
       $name = "Testfolder_$i"
       New-Item -ItemType directory -Name $name
       $i++
}
  while ($i -le 10)
```

We start our counter at 1 and create a new folder using the counter to supply a differentiator. New-Item is the standard cmdlet for creating objects. It is one of the core cmdlets that should be enabled to work in all providers (the SQL Server provider doesn't enable this - see Chapter 14). All we need to do is pass it the type of object and the name of the object as shown. If you want to create the object anywhere but the current folder then

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

the path needs to be defined as well. After creating the folder we increment the counter by one.

The script will create 10 folders named testfolder_1 to TestFolder_10.

As an alternative we could replace the last line of the example above with

```
until ($i -gt 10)
```

Notice how the operator changes because of the difference between while and until.

RECOMMENDATION

The do loop can cause problems in testing as we have seen. If you know the number of times you will traverse the loop use a for loop otherwise use a while loop.

That is enough on loops to make you an expert. We will continue our look at the language features by working with functions and then quickly looking how to output data before moving on to creating scripts.

2.4.5 Functions

The looping mechanisms that we have just looked at are one way to reuse code or at least be more efficient in the way we use code. There are times when we want to run a set of code and get a result returned or we want to separate off a piece of code to make the main part of the script easier to understand. That is when we want to use a function. PowerShell, unlike VBScript for instance, makes no distinction between functions that return a result and one that doesn't. They are all functions.

Let's take Listing 2.9 and use a function instead of the inner loop.

Listing 2.10 PowerShell script to create test folders and test files

```
function new-file {                                                  1
     param ($number, $foldername)                                     2
     for ($i=0; $i -le $number; $i++){                               3
             $name = "$foldername-file-$i.txt"
             Set-Content -Path "$foldername\$name" -Value $data
     }
}                                                                    1
$data = 1..57                                                        4
$i = 1
while ($i -le 10) {                                                  5
     $name = "Testfolder_$i"
     New-Item -ItemType directory -Name $name
     new-file $i $name                                               6
     $i++
}
```

1 Function definition
2 Input parameters for function
3 Loop to create files
4 Data to put into file
5 Loop to create folders
6 Call to function

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

The function has to be defined first and given a name. 1 It is considered best practice to use the PowerShell naming conventions. The param statement is used to accept arguments passed into the function. 2 A data type can be assigned to the parameter. It is possible to use the $args automatic variable to instead of the param statement. The param statement gives more flexibility and can be used to perform more checks on the parameters, including setting default values. The for loop we saw earlier is used to create the test files. 3 In this case the number of files to create is based on the number passed into the function as its first parameter. The folder name is used as part of the file name and to define the path to the file.

The main body of the script follows the function. The folders are created by the while loop used earlier. 5 The only difference being that a call to the function used to create the files is put into the loop. 6 We could have just nested the loops at this point instead of using a function but then I would have had to think of another example! Notice how the function is called. The arguments to pass into the function are listed after the function name rather than using () to surround them as in other scripting languages. More information can be found in the about_function help file.

Two things to notice about the way variables are used in this script. Firstly $i is used in the main body of the script and within the function as a counter. As far as PowerShell is concerned these are two totally independent variables even though they have the same name. Secondly, we define $data in the main body of the script but use it in the function. This is an example of variable scope which is not as complicated as it seems.

SCOPE

Scope defines how scripts and functions work with variables. When PowerShell starts it defines a top level or *global* scope. When a script, script block, or function is started a new scope is defined. A parent-child relationship is created between the original scope and the child scope. Similarly when a function is called within a script a scope is created that is a child of the script's scope.

Variables are defined with the scope of where they are created. In our function example $data is created in the scope of the script as is the $i used in the while loop. The $i used in the function is recreated every time the function is called and is created in the scope of the function. The function's scope is a child of the script's scope. The function can read and use the $data variable because it was created in the parent scope of the function.

PowerShell treats the $i independently because they are created in different scopes. The rule is that a variable can be read from a higher scope but not modified unless the scope of the variable is stated. A parent scope cannot modify variables in a child scope.

The important thing to remember is that scope exists and that child scopes can access variables from the parent scope. `Get-help about_scope` gives more examples.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

LIBRARY FILES

Functions can be loaded into memory at the start of a PowerShell session as we saw when looking at profiles. The profile can become too big to manage if there are many functions defined within it so the alternative is to use a library file. Collect the functions into a single file called LibraryXXXX.ps1 where XXXX is s suitably descriptive name. The library file can be dot sourced from your profile to make the functions available within the PowerShell session. Alternatively, dot source the library file from a script if you do not require the functions to be universally available.

The last of the language features is output. We have seen how to use variables and loops, how to make decisions and the working of functions. Administrative scripts normally produce output of some kind. We will close this section by quickly looking at the output methods in PowerShell.

2.4.6 Output

Most administrative scripts produce output even if it is only a message to say the script has completed. There are number of output methods available in PowerShell:

- Out-* - cmdlets that send the output to a specific destination
- Write-* - cmdlets that writes data to a specific destination
- Format-* - cmdlets that format the data and write it by default to the screen
- Export-* - cmdlets that export data to a file

Typing `Get-Command Out-*` or any of the other examples in the above list will generate a list of the cmdlets in that category. Get-Help can be used to view the help file. The commonly used output cmdlets are summarized in the following table.

Table 2.3 Common output cmdlets

Name	Purpose
Out-File	Sends output to a file. Data can be appended to a file.
Out-Null	Deletes output rather than displaying. Use it to suppress messages for instance when loading .NE T assemblies.
Out-Printer	Sends output to a printer. It does not format the data.
Out-String	Sends objects to the host as strings.
Write-Host	Writes output to screen. Can change colors of text.
Format-Table	Formats the data into a table and displays on screen.
Format-List	Formats the data into a list and displays on screen.
Format-Wide	Formats objects as a wide table that displays only one property of each object.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

Export-CSV	Exports objects to a csv file. Data cannot be appended to a file.
Export-CliXML	Exports objects to an xml file.

Examples of using these cmdlets will be seen though out the book.

Time to put everything together and start looking at scripts.

2.5 Scripts

Scripts represent a method of saving PowerShell code and reusing it at a future time either as a scheduled task or as *ad hoc* usage. Chapter 4 covers script development in detail. This section is concerned with converting to PowerShell from other langauages.

2.5.1 PowerShell Scripts

Scripts are text files containing PowerShell commands that are given a .ps1 extension. The same extension is used for scripts in PowerShell version 2. Any editor that can output text files can be used to create scripts ranging from notepad (still very useful to view scripts) to PowerShell specific editors such as PowerGUI, PowerShell+ or graphical PowerShell in version 2. The PowerShell toolkit in Chapter 4 has more information on these tools.

PowerShell scripts are used by typing the path and name of the script at a PowerShell prompt. If the script is in the current folder the path is .\ as in .\myscript.ps1.

One topic that frequently arises is how can I convert my existing scripts, written in VBScript, or another scripting language, into PowerShell scripts. If you have a large library of VbScripts that are in production use I would recommend that the you do not rush into converting them. Leave them in use and start developing new scripts in PowerShell. Convert the existing scripts as they need modification.

There will be the need in many circumstances for PowerShell scripts to either be converted into PowerShell or to inter-operate with PowerShell. We will look at how to do this in the next sections.

2.5.2 Converting from VBScript

VBScript has been in use on Windows systems for over ten years. There is a huge body of administration scripts available starting with for example Microsoft's Script Center http://www.microsoft.com/technet/scriptcenter/scripts/default.mspx?mfr=true. Some of the scripts have been converted to PowerShell but many haven't. So how do we go about converting a script.

A lot of the scripts use WMI in which case we can use Get-WmiObject as explained in the next chapter. For other scripts we start with the VBScript-to-Windows PowerShell Conversion Guide which is downloadable from http://www.microsoft.com/technet/scriptcenter/topics/winpsh/convert/default.mspx.

We will use this script to try the conversion.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

```
If objFSO.FileExists("C:\scripts\librarytime.ps1") Then
    Set objFolder = objFSO.GetFile("C:\scripts\librarytime.ps1")
Else
    Wscript.Echo "File does not exist."
End If
```

It uses the FileExists method of the FileSystemObject to test if a file exists. If the file exists a GetFile is performed so that the properties can be accessed.

In PowerShell this would become

```
if (Test-Path -Path "C:\Scripts\LibraryTime.ps1") {
 $file = Get-Item -Path "C:\Scripts\LibraryTime.ps1"
}
else
{
 Write-Host "File does not exist"
}
```

Test-Path returns true or false depending on whether or not the particular path exists. One advantage that PowerShell has is that this same code will work in other providers by just changing the object being tested.

Converting scripts is a big step. It may be better to start small and incorporate some VBScript into your early PowerShell scripts.

2.5.3VBScript in PowerShell

VBScript is based on COM while PowerShell is based on .NET. PowerShell can incorporate VBScript objects by using New-Object to create them. This will be covered in Chapter 4 in the section on COM.

One way of starting to work with PowerShell from a VBScript base is to incorporate VBScript into your PowerShell scripts so that you are only converting parts of the script. There are a couple of ways to use VBScript functionality in a PowerShell script.

We can start by really incorporating VBScript as this example demonstrates.

```
$sc = New-Object -ComObject ScriptControl
$sc.Language = "VBScript"
$sc.AddCode('
Set obj = CreateObject("WScript.Shell")
obj.popup "Popup from PowerShell via VBScript",,"PowerShell in
Practice",0
')
$sc.CodeObject
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

VBScript is COM based so we need to use the ComObject parameter of New-Object to define a COM ScriptControl object. There isn't a lot of documentation available on this object but a good starting point can be found at http://msdn.microsoft.com/en-us/library/aa227633(VS.60).aspx. We then define the language as VBScript (Jscript could be used - in fact it would be possible to use PowerShell, VBScript and Jscript in a single script). The AddCode method is used to define the script. There are some restrictions here in that it doesn't seem possible to use multiple objects in the same ScriptControl. The VBScript is very simple in that it creates a WScript.Shell object and then displays the popup.

This approach has its limitations especially around defining the VBScript code if multiple different objects are required in the code. We can move this on a little and use the COM objects directly in PowerShell.

```
$a = New-Object -ComObject WScript.Shell
$a.popup("Popup from PowerShell",$null,"PowerShell in Practice",0)
```

We define the WScipt.Shell object directly as a COM object in PowerShell and then call the popup method. It is important to note the difference in syntax around using the method. In the first example we used the VBScript syntax with the arguments listed after the method name and separated by commas. In the second example we used the .NET syntax with a comma delimited list of arguments surrounded by parentheses.

Having seen how to use VBScript inside PowerShell we can now quickly look at using PowerShell inside VBScript for completeness.

2.5.4 PowerShell in VBScript

SAPIEN have released a free control that enables us to use PowerShell inside VBScript. This is a valid way to start converting scripts by incorporating PowerShell into existing scripts. The control can be downloaded from http://blog.sapien.com/index.php/2008/06/25/activexposh-is-now-a-free-download/.

The control can be used like this.

```
set ap = CreateObject("SAPIEN.ActiveXPoSH")
ap.OutPutMode = 0
ap.Execute("Get-Service | Where-Object{$_.Status -e 'stopped'}")
```

This is standard VBScript in that we create the object and then call the Execute method to run the PowerShell code.

You will now have a good understanding of PowerShell and how it works. Congratulations, you are now ready to start building your PowerShell toolkit by looking at how PowerShell works with other technologies to access WMI, Active Directory and .NET.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=499

2.3.1 Get-Help 13
2.3.2 Get-Command 15
2.3.3 Get-Member 15
2.3.4 Get-PSDrive 17
2.4 Language features 17
2.4.1 Variables 18
2.4.2 Arrays 19
2.4.3 Branches 20
If statement 20
Switch statement 21
2.4.4 Loops 22
Foreach-Object 23
Foreach loop 24
For, while and do loops 26
2.4.5 Functions 28
scope 29
Library files 30
2.4.6 Output 30
2.5 Scripts 31
2.5.1 PowerShell Scripts 31
2.5.2 Converting from VBScript 31
2.5.3VBScript in PowerShell 32
2.5.4PowerShell in VBScript 33
2.6 Summary 34

2.3.3 Get-Member 15
2.3.4 Get-PSDrive 17
2.4 Language features 17
2.4.1 Variables 18
2.4.2 Arrays 19
2.4.3 Branches 20
If statement 20
Switch statement 21
2.4.4 Loops 22
Foreach-Object 23
Foreach loop 24
While and do loops 26
2.4.5 Functions 28
scope 29
Library files 30
2.4.6 Output 30
2.5 Scripts 31
2.5.1 PowerShell Scripts 31
2.5.2 Converting from VBScript 31
2.5.3 VBScript in PowerShell 32
2.5.4 PowerShell in VBScript 33
2.6 Summary 34

2.6 Summary

Learning PowerShell is done by using PowerShell. First we need to install and configure it so we can start learning. The PowerShell documentation is a good starting point and PowerShell supplies four built in learning tools Get-Help, Get-Command, Get-Member and Get-PSDrive. These are your four best friends as far as learning PowerShell is concerned. Use them to supplement the in built documentation.

PowerShell is a scripting language and command shell. The langauage has a number of features that need to be mastered including variables, arrays, loops, branching logic and functions.

VBScript has been available for a number of years. There is a guide available for converting scripts from VBScript into PowerShell. As a first step it is possible to embed VBScript in PowerShell and vice versa.

This chapter has supplied a grounding in installing, configuring and learning PowerShell. You are now ready to start looking beyond PowerShell to the other technologies that need to be mastered in order to successfully administer your Windows systems using PowerShell. In Chapter 3 we shall look at using .NET in PowerShell so we have access to a wider range of functionality; WMI so we can access existing Windows management technologies both on local and remote machines; COM so we can access existing applications such as Office and ADSI so we can work with Active Directory and local security systems.