

APPENDIX A

Basic PowerShell

IN THIS APPENDIX YOU WILL LEARN TO:

► DISCOVER POWERSHELL'S ORIGINS	2
► DEVELOP YOUR UNDERSTANDING OF OBJECTS	4
What Is an Object?.....	4
The PowerCLI Objects	5
Creating Your Own Objects	6
► UNCOVER CMDLETS AND GET HELP	8
What Is a Cmdlet?	8
Tab Completion	10
Built-In Help.....	11
The <i>Get-Member</i> Cmdlet	12
► UTILIZE THE POWERSHELL PIPELINE	17
Using the Pipeline in PowerShell	17
Bringing It All Together	18
► BUILD YOUR TOOLKIT OF BASIC CMDLETS	21
Reporting	22
Enhancing Script Writing	23

PowerCLI is a toolkit in its own right for managing a vSphere infrastructure. However, since PowerCLI is a snap-in for Microsoft's Windows PowerShell, a good understanding of basic PowerShell will not only help you with learning PowerCLI, but it will also enable you to improve command-line management and script writing. In this appendix, we will examine some of the basic concepts and set you on the right road to becoming both a PowerShell and PowerCLI expert!

Discover PowerShell's Origins

No system administrator who manages Unix- or Linux-based systems needs to be convinced of the importance of command-line management. Historically, though, the Windows-based operating systems have been primarily managed via graphical user interfaces (GUIs)—it's been all about mouse clicking. While GUIs are great for an instant visualization of a management issue or for configuring a single task, they do not lend themselves to bulk management. The larger an IT environment becomes, the more necessary it is to have efficient, repeatable, and timesaving processes in place to ensure that administrators are spending their valuable time doing the most useful work.

If you work through a wizard and create a new virtual machine (VM), you have nothing to show for it at the end of that process other than one job off your list for that day. If you create a VM with a set of commands and save those commands into a script, you have a repeatable and consistent process for the next VM. The next time you're asked to create 100 VMs at 4:55 p.m. on a Friday afternoon, you'll be the one at home in time to stoke up the BBQ for a family evening with the spouse and kids, while your colleagues are still at work considering making a doctor's appointment from the repetitive strain injury they have caused themselves from mouse-clicking through the New Virtual Machine wizard 100 times.

Thankfully for Windows-based system administrators, back in the early 2000s Microsoft decided they needed a command-line shell to complement the success they had achieved with GUIs. Everybody knows they had `cmd.exe`, which was fine for running simple batch files, but it was not a tool for managing large IT environments. While different product teams would write different command tools that could be run from `cmd.exe` (such as Netsh or Robocopy), no global standards existed; therefore, the overall experience would differ for each set of commands. For instance, if you went from one command-line tool to another, there would be inconsistent parameter styles; is it `command -a`, or `command -A`, or `command /a`?

Of course, some Windows administrators were able to take advantage of VBScript, which was present on Windows systems. Although VBScript was originally designed with developers in mind, it was possible to use it for scripting many administrative tasks. However, as the .NET Framework was released and implemented into Windows, VBScript no longer had access. A new scripting technology was needed to take advantage of this framework.

What initially came from this need was a project known as Monad. Urban rumor has it that Monad was named after now legendary PowerShell guru /\/o\/\ (real name Marc van Orsouw), but it's more likely influence was one of the works of Gottfried Wilhelm Leibniz (1646–1716), *The Monadology*. Pioneering the effort at Microsoft was Jeffrey Snover. Snover eventually managed to convince enough important people within the company that this project was necessary and would be a success. From that evolved PowerShell 1.0, released in November 2006 at IT Forum in Barcelona.

Eighteenth-century English writer Charles Caleb Colton once said, “Imitation is the sincerest form of flattery.” Microsoft sensibly looked at the success of Unix shells such as Bash and Korn, examined the success of other programming languages such as Ruby and Python, and using these influences and their own talents, put together their own shell and scripting language in PowerShell.

So, is it a shell or a scripting language? The simple answer is both! PowerShell encompasses both an interactive command-line shell and a fully functional scripting language. The same commands used in the shell to carry out interactive administration can be packaged up into a script and executed from the shell. This enables you to create repeatable processes from everyday administration tasks.

Depending on your background, this may be a completely new approach based on your experiences with other programming languages or maybe, as a system administrator, this is the first language you have used. The hurdles to clear will be different depending on your experience. If you have a Unix background, then working with shells will be familiar, as will writing scripts. However, you will have been accustomed to manipulating text, not dealing with objects. If you have a Windows system administrator background, then your scripting experience may have been with VBScript. (You'll be pleased to know that PowerShell was designed for the system administrator, to make your life easier.) While some very smart people were able to do some very clever things with VBScript, PowerShell is a language made for you.

If you're a new PowerShell user, start with the shell. Perhaps at first you replace tasks you accomplished using `cmd.exe`—the vast majority of built-in commands in `cmd.exe` function exactly the same in PowerShell. Moving on from there, you can begin to use some of the native PowerShell cmdlets such as `Get-Service`,

Start-Service, and Stop-Service or Get-Process, Start-Process, and Stop-Process. Once you become familiar with these cmdlets, you can move on to saving a combination of these commands into a script to create a repeatable process.

In reality, once you begin to build your PowerShell experience, you will be just as comfortable working within the interactive shell as you are with a scripting integrated development environment (IDE).

Develop Your Understanding of Objects

PowerShell is an object-oriented programming language, and possibly the key to grasping the entire concept of PowerShell is to understand the implication that you are always working with objects.

What Is an Object?

Everything in PowerShell is an object; understand that concept and you will be well on your way. Whether you are dealing with a text string, a Windows COM object, some .NET code, or PowerCLI, you are using some kind of object. Typically, objects have both properties and methods. Let's take a look at an abstract example to help explain this concept.

Imagine you have just purchased a new car. The car has both characteristics (properties) and possible actions (methods). Let's take a look at some examples. Among all of its characteristics your new car might have the following:

- ▶ Color: Red
- ▶ Automatic: True
- ▶ 4 Wheel Drive: Enabled

Notice how these properties differ slightly; the Color is a label, Automatic is True or False, and 4 Wheel Drive is Enabled or Disabled. These properties help to describe the object and are different pieces of information you might need to build a picture of this object.

Now think about the possible actions that could be taken with a car:

- ▶ Drive Forward (Fast)
- ▶ Drive Backward (Slow)

- ▶ Open the Sunroof (Halfway)

- ▶ Open the Hood

Some of these actions have parameters (think options), so for instance when we Open the Sunroof, we can specify that we only want to open it Halfway. Some of them, like Open the Hood, have no possible options.

Let's have a look at a more IT-related example. Think about a service on a Windows server. In PowerShell terms, as an object, it will have both properties and methods. For example, it would have properties like

- ▶ DisplayName: Print Spooler

- ▶ Status: Running

- ▶ ServicesDependedOn: HTTP, RPCSS

Again, you have slightly different types of properties; the first two are single valued and the third has the possibility to be multivalued.

This service also has methods, which are actions you can carry out on that service:

- ▶ Start

- ▶ Stop

- ▶ WaitForStatus

With these methods, you could manipulate the Print Spooler service. You could stop and start it, or wait for it to have a particular status.

The PowerCLI Objects

PowerCLI is no different from standard PowerShell when it comes to objects. Everything you deal with is an object, whether it's a VM, an ESX(i) host, or a virtual switch. Similarly, with PowerCLI objects you have both properties and methods.

Let's look at some of the properties of a VM:

- ▶ Name: Server01

- ▶ NumCPU: 2

- ▶ MemoryMB: 4068

- ▶ PowerState: PoweredOn

These properties help to describe the VM and could be useful for creating reports or helping to search for a certain group of VMs.

The virtual machine object also has methods available to you:

- ▶ `CreateSnapshot_Task`: (Name, Description, Memory?, Quiesce?)
- ▶ `Destroy_Task`
- ▶ `PowerOffVM_Task`
- ▶ `Rename_Task`: (NewName)

Notice again how some of these methods require you to supply parameters in order to run them.



TIP To use methods with PowerCLI objects, you need to work with managed objects rather than the standard object that is returned. We'll talk more about this later in this appendix in the section "The Get-Member Cmdlet."

Creating Your Own Objects

So far, we have thought about objects provided to us by default, either from Windows PowerShell or from vSphere PowerCLI. However, what can you do if it is not possible to retrieve the object you need from that which is easily supplied to you? Within Windows PowerShell, you can create a new object of your own; you can do this either with .NET or a COM object.

The `New-Object` cmdlet was designed for this purpose. For example, if you wanted to create a new `DateTime` object to work with, you could use the following:

```
$ChristmasDay = New-Object -TypeName System.DateTime 2010,12,25
```

By storing this newly created object into a `$ChristmasDay` variable, you could now access properties such as:

- ▶ `DayOfWeek`: Saturday
- ▶ `Month`: 12

You could also use some of the available methods:

- ▶ `AddDays`: (20)
- ▶ `ToShortDateString`

So, you could carry out date calculations like finding out what is the date 20 days after Christmas Day or convert your `DateTime` result to a `ShortDateString`, 12/25/2010.

Similarly, you can also create new PowerCLI objects with the `New-Object` cmdlet. Sometimes, it is not possible to change a setting on a VM with the cmdlets supplied with PowerCLI. However, if you create a new object of the type `VMware.Vim.VirtualMachineConfigSpec`, you would have access via the API to change as many of the virtual machine's properties as are available via the API.

```
$NewConfigSpec = New-Object -TypeName
VMware.Vim.VirtualMachineConfigSpec
```

(See Chapter 19, “The SDK,” for more detailed information on how to work with your own PowerCLI objects.)

At this point, it is also worth mentioning that it is possible to add your own properties and methods to existing objects using the `Add-Member` cmdlet. Let's have a look at an example using a file on a Windows system:

1. First, retrieve the details of a file and store it in a variable:

```
$file = Get-Childitem test.txt
```

2. Use the `Add-Member` cmdlet to add a `NoteProperty` to the file system object:

```
$file | Add-Member -MemberType noteproperty -Name Migrate
-value Complete
```

3. Now, examine that new property:

```
$file.Migrate
Complete
```

Note, the property only persists for this session.

Similarly, with PowerCLI you can add new properties to objects you want to work with. With vSphere 4.1, VMware makes this even simpler for you by providing a new cmdlet, `New-VIProperty`. Say you want to add a new property when returning ESX(i) host objects to retrieve the CPU model of the host. You can use the `New-VIProperty` cmdlet to create this custom property:

```
New-VIProperty -ObjectType VMHost -Name CPUModel
-ValueFromExtensionProperty 'Summary.Hardware.CPUModel' -Force
```

Now when you use the `Get-VMHost` cmdlet, you can specify the custom `CPUModel` property:

```
Get-VMHost VIRTUESX2* | Select Name,CPUModel
```

Name	CPUModel
----	-----
VIRTUESX2.virtu-al.local	Intel(R) Xeon(R) CPU E5450 @ 3.00GHz

Uncover Cmdlets and Get Help

Toward the end of the previous section we started to use some PowerShell cmdlets; before we go too much further, let's look at what a cmdlet is and how you can get help with them.

What Is a Cmdlet?

A *cmdlet* is a lightweight command that performs an action and typically returns a .NET object. Cmdlets differ from other shell-type commands in that they are not stand-alone executables; rather, the PowerShell runtime invokes them. The name *cmdlet* lends itself to this definition of a lightweight command, but it also shows a significant amount of forethought on behalf of the Windows PowerShell team. Try searching the Web for the word “command”; no doubt you will return millions of different results from different shells and programming languages. Now try the same search with the word *cmdlet*; notice how, while there are still hundreds of thousands of results, they are exclusively PowerShell based—at least it gives you a decent start in the search you are making!

Over 200 cmdlets are included as part of the basic installation of Windows PowerShell version 2. Since PowerShell is fully extensible, it is simple for other Microsoft product teams to ship cmdlets for their own products, such as Exchange or SharePoint. Third-party vendors, like VMware, are able to do so as well. PowerCLI, in fact, offers another 200+ cmdlets of its own on top of the default PowerShell cmdlets.

One of the key design aims of PowerShell is to make it discoverable. It should be easy for the user to figure out how it works. The recommended naming convention

for PowerShell cmdlets is in the form *Verb-Noun*, for instance `Get-VM`. Here are some typical common verbs for PowerShell cmdlets:

- ▶ Get
- ▶ Set
- ▶ New
- ▶ Add
- ▶ Remove
- ▶ Invoke



TIP You'll find a full list of common verbs for PowerShell cmdlets at this website:

<http://msdn.microsoft.com/en-us/library/ms714395%28VS.85%29.aspx>

Consequently, if you know what `Get-VM` does, then it does not take a genius to figure out the action for `Set-VM`, `New-VM`, `Remove-VM`, and so on. Both Microsoft and VMware are very keen for you to adapt these automation technologies. Consequently, making the barriers to entry as small as possible is an important factor.

Each cmdlet has its own set of parameters dependent on the actions it needs to carry out. In addition to this, however, each cmdlet should feature common parameters, again making the language discoverable and as easy as possible to learn. These could include the following:

- ▶ Debug
- ▶ ErrorAction
- ▶ Confirm
- ▶ Whatif



TIP A full list appears at this website:

<http://msdn.microsoft.com/en-us/library/dd901844%28VS.85%29.aspx>

For example, by using the `Whatif` parameter when running a cmdlet, you can see what would have happened if you had run the cmdlet. This is obviously useful for

cmdlets that make changes. A good use for this is with the `Shutdown-VMGuest` cmdlet. If you apply the `Whatif` parameter, you can see what would have happened if you had run it:

```
Get-VM PrintServer01 | Shutdown-VMGuest -WhatIf
What if: Performing operation "Shutdown VM guest." on VM
"PrintServer01".
```

As you can see, it would have carried out the operation `Shutdown VM guest` on the virtual machine `PrintServer01`.

Tab Completion

Another discoverable feature of Windows PowerShell when working within the command shell is tab completion. Using the Tab key, you can cycle through and select options for both cmdlets and their parameters. Continue pressing the Tab key until you find the desired cmdlet or parameter, and then press the spacebar to move on.

Cmdlets

Try typing **Get-** and then hitting the Tab key. Now, you can cycle through the list of available cmdlets beginning with `Get-`. Tab completion also functions the more text you type in. Imagine you know the cmdlet you need is `Get-VMsomething`, but you can't quite remember what the *something* is. You can continue pressing the Tab key until you find the *something* you are looking for.

Parameters

The tab completion functionality also follows through to cmdlet parameters. No longer do you have to remember if it is `cmdlet -a` or `cmdlet -A` or `cmdlet /a`—it will always be `cmdlet -something`. Again, you can press the Tab key and cycle through the available parameters until you find the one you want.

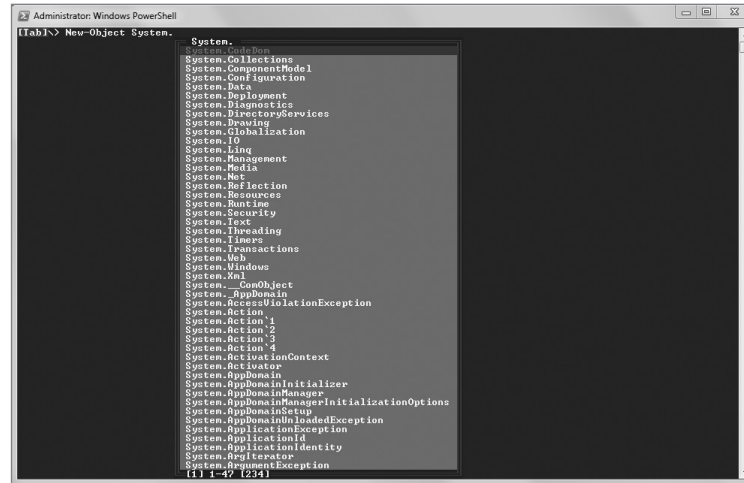
Tab completion is an essential feature for scripting IDEs, and it is found in all of the commonly used PowerShell script editors, such as PowerGUI, PowerShell Plus, and the built-in PowerShell ISE.

In addition to the built-in tab completion features of PowerShell and those in scripting IDEs, it is worth considering the open source project PowerTab. This project, originally started by legendary PowerShell community contributor `/v/o/` and now available on the CodePlex website, extends the tab completion functionality even further and gives you an IntelliSense drop-down list of values to choose from. Visit the CodePlex site at

<http://powertab.codeplex.com/>

In addition to tab completion for cmdlets and parameters, PowerTab offers tab completion for items such as WMI and .NET Framework classes. Figure A.1 demonstrates the .NET tab completion from PowerTab.

FIGURE A.1 .NET tab completion from PowerTab



Built-In Help

PowerShell has built-in help ready for every cmdlet; you can access it using `Get-Help` followed by the name of a cmdlet. Unix administrators will be familiar with *man pages*; PowerShell help is similar. Cmdlet developers, whether Microsoft or a third-party organization, are expected to include built-in help for every cmdlet they produce.

For example, to read the built-in help for the `Get-Service` cmdlet, use the following:

```
C:\Scripts> Get-Help Get-Service
```

The response will be a short summary of what the cmdlet does, available parameters, a fuller description, and a list of related cmdlets. This is a good place to start, but more detailed options are available. The `Get-Help` cmdlet has parameters. Here are some of the most useful:

Detailed The `Detailed` parameter brings up a more detailed description of the cmdlet, an explanation of each parameter, and some examples.

Full The `Full` parameter allows you to access detailed help plus more extensive information on each parameter. In particular, you can find out whether the parameter is mandatory, has a position, has a default value, and whether it accepts pipeline input or wildcard characters.

Examples Use the `Examples` parameter to view just the examples. This can be very useful when you are new to a cmdlet and looking for a quick view on how it works.

Online Add the `Online` parameter to view the online help. Typically, the web-based help is the same as the built-in help, but it is updated more frequently.

About It is also worth taking a look at the about pages within the built-in help. These are more topic based, sometimes containing groups of cmdlets (such as `remoting`) or more general PowerShell topics (like script signing). The list of available about pages can be found by using the wildcard character `*` in conjunction with `Get-Help`:

```
Get-Help about*
```

Once you find your the topic of interest, access via

```
Get-Help about_Windows_PowerShell_ISE
```

The *Get-Member* Cmdlet

As mentioned earlier in the appendix, PowerShell is all about objects. Whatever your programming background, the sooner this is understood, the sooner you will make progress learning PowerShell. The `Get-Member` cmdlet was provided in PowerShell to help you discover more about the objects you are working with. In particular, it reveals the properties and methods that are available to you. Finding out what possibilities you have with an object will lead you to greater exploration with that object and more uses for it. Let's look at an example to illustrate this.

Earlier in the appendix, we looked at the `Get-Service` cmdlet and some of its properties and methods. How did we find out which properties and methods were available? Well, we used the `Get-Member` cmdlet to reveal them:

```
Get-Service 'Print Spooler' | Get-Member
```

This revealed the type of the object and the properties and methods listed next. Take a look. You will see some of those we selected for discussion earlier in the appendix.

```
TypeName: System.ServiceProcess.ServiceController
Name                                     MemberType
----                                     -
Name                                     AliasProperty
RequiredServices                         AliasProperty
Disposed                                 Event
Close                                    Method
```

Continue	Method
CreateObjRef	Method
Dispose	Method
Equals	Method
ExecuteCommand	Method
GetHashCode	Method
GetLifetimeService	Method
GetType	Method
InitializeLifetimeService	Method
Pause	Method
Refresh	Method
Start	Method
Stop	Method
ToString	Method
WaitForStatus	Method
CanPauseAndContinue	Property
CanShutdown	Property
CanStop	Property
Container	Property
DependentServices	Property
DisplayName	Property
MachineName	Property
ServiceHandle	Property
ServiceName	Property
ServicesDependedOn	Property
ServiceType	Property
Site	Property
Status	Property

Notice that several member types are listed: property, method, event, and alias property. For a new PowerShell user, properties and methods are the most important member types to grasp initially. After you've gained a little experience, take some time to research the other member types. Look at the help for `Get-Member` to see what other member types are available. Here's a quick description of the other two member types in this example:

Alias Property An `AliasProperty` defines a new name for an existing property.

Event Event member types indicate that the object sends a message to indicate an action or a change in state.

To be slightly more focused with `Get-Member`, you can take advantage of the `MemberType` parameter, which allows you to request only the properties, methods, or other member types available for a particular object. For instance, to request the properties for the `Get-Service` cmdlet only, you would use the following:

```
Get-Service 'Print Spooler' | Get-Member -MemberType Property
```

Sometimes, however, it's useful to see the value of a property, in addition to the property name, to ensure you know what the property actually is. A lighter equivalent of `Get-Member -MemberType Property` uses the `Format-List` cmdlet with the wildcard character `*` to return all properties and their values. This can be a quick way to spot the property required, particularly if the value is already known.

```
Get-Service 'Print Spooler' | Format-List *
```

```
Name                : Spooler
RequiredServices     : {HTTP, RPCSS}
CanPauseAndContinue  : False
CanShutdown          : False
CanStop              : True
DisplayName           : Print Spooler
DependentServices    : {Fax}
MachineName          : .
ServiceName          : Spooler
ServicesDependedOn   : {HTTP, RPCSS}
ServiceHandle        :
Status               : Running
ServiceType          : Win32OwnProcess, InteractiveProcess
Site                 :
Container            :
```

It naturally follows that you can also use the `Get-Member` cmdlet with `PowerCLI` objects. For instance, take the example of the `Get-VM` cmdlet observed with `Get-Member` and the results it returns:

```
Get-VM PrintServer01 | Get-Member
```

```
TypeName: VMware.VimAutomation.ViCore.Impl.V1.Inventory.
VirtualMachineImpl
```

Name	MemberType
----	-----
ConvertToVersion	Method
Equals	Method
GetHashCode	Method
GetType	Method
IsConvertibleTo	Method
ToString	Method
CDDrives	Property
CustomFields	Property
DatastoreIdList	Property
Description	Property
DrsAutomationLevel	Property
ExtensionData	Property
FloppyDrives	Property
Folder	Property
FolderId	Property
Guest	Property
HAIsolationResponse	Property
HardDisks	Property
HARestartPriority	Property
Host	Property
HostId	Property
Id	Property
MemoryMB	Property
Name	Property
NetworkAdapters	Property
Notes	Property
NumCpu	Property
PersistentId	Property
PowerState	Property
ProvisionedSpaceGB	Property
ResourcePool	Property
ResourcePoolId	Property
Uid	Property
UsbDevices	Property
UsedSpaceGB	Property

VApp	Property
Version	Property
VMHost	Property
VMHostId	Property
VMResourceConfiguration	Property
VMSwapfilePolicy	Property

You should notice that although it returns lots of available properties, the list of methods is limited and not quite what you would expect to see. It does not show any actions to carry out in virtual machines. In this way, PowerCLI objects work slightly differently than the Service object we looked at in Windows. To retrieve the methods for a PowerCLI object, you need to use the `Get-View` cmdlet. (The results are cut short for brevity.) This time you see lots of actions that can be carried out on a virtual machine.

```
Get-VM PrintServer01 | Get-View | Get-Member -MemberType Method
```

TypeName: VMware.Vim.VirtualMachine

Name	MemberType
----	-----
AcquireMksTicket	Method
AcquireTicket	Method
AnswerVM	Method
CheckCustomizationSpec	Method
CloneVM	Method
CloneVM_Task	Method
CreateScreenshot	Method
CreateScreenshot_Task	Method
CreateSecondaryVM	Method
CreateSecondaryVM_Task	Method
CreateSnapshot	Method
CreateSnapshot_Task	Method
CustomizeVM	Method
CustomizeVM_Task	Method
DefragmentAllDisks	Method
Destroy	Method
Destroy_Task	Method
DisableSecondaryVM	Method

DisableSecondaryVM_Task	Method
EnableSecondaryVM	Method
EnableSecondaryVM_Task	Method

Utilize the PowerShell Pipeline

You will notice that in the `Get-Member` cmdlet example we used a technique known as pipelining; each PowerShell cmdlet was separated by a pipe (`|`) symbol. Think of it as a factory conveyor belt with different tasks carried out at each stage along the belt and the result of the previous stage being passed on to the next until a product is produced at the end.

This concept should be familiar in some form to Unix administrators. The key difference is that with PowerShell we are passing .NET objects down the pipeline, not text. This concept should be also be familiar to Windows administrators. Even in DOS, it was possible to use the pipeline to take the results of one command and send it to another.

Using the Pipeline in PowerShell

PowerShell significantly builds on this pipelining concept. It is fundamental to understanding some of the best ways to work at the command line and build scripts. Consequently, you may hear a lot about *one-liner* PowerShell commands. What this typically means is that a number of cmdlets are used in conjunction with the PowerShell pipeline to carry out a significant task with only one line of code. Although this can be an effective form of PowerShell, sometimes it is emphasized too much. New PowerShell users can be put off, thinking that they have to be super smart to use PowerShell. (This complaint often comes from Windows administrators after comparing how it would have taken *x* number of lines to achieve the same task in VBScript.)

As with every programming language, there are many ways to achieve the same end result, so find the method that works for you best. You can always improve your technique later as you become more accomplished with PowerShell.

Bringing It All Together

Let's look at an example of a few cmdlets and how you can build them up along the pipeline to produce a useful end result.

1. Run `Get-Service` to retrieve all Windows services on this machine. The results listed next are cut short for brevity.

```
Get-Service
```

Status	Name	DisplayName
-----	----	-----
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Stopped	AppIDSvc	Application Identity
Running	Appinfo	Application Information
Stopped	AppMgmt	Application Management
Stopped	aspnet_state	ASP.NET State Service
Running	AudioEndpoin.	Windows Audio Endpoint Builder
Running	AudioSrv	Windows Audio
Stopped	AxInstSV	ActiveX Installer (AxInstSV)
Stopped	BDESVC	BitLocker Drive Encryption Service
Running	BFE	Base Filtering Engine
Running	BITS	Background Intelligent Transfer Ser.
Running	Browser	Computer Browser
Stopped	bthserv	Bluetooth Support Service

2. Pipe the objects returned from the `Get-Service` cmdlet into the `Where-Object` cmdlet. In the code that follows, we used `Where-Object` to filter the list of services down to only those beginning with the letter S. You can use this technique to filter the list to meet your needs.



TIP Notice the first two characters in the script block, `$_`. You might wonder what they are. `$_` refers to the current object being worked on in the pipeline. You will use it extensively when using the pipeline. In our example, it is used to cycle through each of the objects returned from `Get-Service`, access the `Name` property (by separating `$_` and the property name with a `.`), and then evaluate whether each one begins with the letter S.

```
Get-Service | Where-Object {$_.Name -like 's*'}
```

Status	Name	DisplayName
-----	----	-----
Running	SamSs	Security Accounts Manager
Running	SCardSvr	Smart Card
Running	Schedule	Task Scheduler
Stopped	SCPolicySvc	Smart Card Removal Policy
Stopped	SDRSVC	Windows Backup
Stopped	seclogon	Secondary Logon
Running	SENS	System Event Notification Service
Stopped	SensrSvc	Adaptive Brightness
Stopped	SessionEnv	Remote Desktop Configuration
Stopped	SharedAccess	Internet Connection Sharing (ICS)
Running	ShellHWDetec.	Shell Hardware Detection
Stopped	SNMPTRAP	SNMP Trap
Running	Spooler	Print Spooler
Stopped	sppsvc	Software Protection
Stopped	sppuinotify	SPP Notification Service
Running	SSDPSSRV	SSDP Discovery
Stopped	SstpSvc	Secure Socket Tunneling Protocol Se.
Stopped	stisvc	Windows Image Acquisition (WIA)
Stopped	StorSvc	Storage Service
Stopped	swprv	Microsoft Software Shadow Copy Prov.
Running	SysMain	Superfetch

3. Pipe the objects returned from only services beginning with the letter S to the Select-Object cmdlet. Use Select-Object to pick particular properties of the services that you are interested in.

```
Get-Service | Where-Object {$_.Name -like 's*'} |  
Select-Object Name,Status,ServiceType
```

Name	Status	ServiceType
----	-----	-----
SamSs	Running	Win32ShareProcess
SCardSvr	Running	Win32ShareProcess

Schedule	Running	Win32ShareProcess
SCPolicySvc	Stopped	Win32ShareProcess
SDRSVC	Stopped	Win32OwnProcess
seclogon	Stopped	Win32ShareProcess
SENS	Running	Win32ShareProcess
SensrSvc	Stopped	Win32ShareProcess
SessionEnv	Stopped	Win32ShareProcess
SharedAccess	Stopped	Win32ShareProcess
ShellHWDetection	Running	Win32ShareProcess
SNMPTRAP	Stopped	Win32OwnProcess
Spooler	Running	Win32OwnProcess, InteractiveProcess
sppsvc	Stopped	Win32OwnProcess
sppuinotify	Stopped	Win32ShareProcess
SSDPSRV	Running	Win32ShareProcess
SstpSvc	Stopped	Win32ShareProcess
stisvc	Stopped	Win32OwnProcess
StorSvc	Stopped	Win32ShareProcess
swprv	Stopped	Win32OwnProcess
SysMain	Running	Win32ShareProcess

4. Finally, take the objects returned from the first three cmdlets and export them into a CSV file using the Export-CSV cmdlet.

```
Get-Service | Where-Object {$_.Name -like 's*'} |  
Select-Object Name,Status,ServiceType |  
Export-CSV Services_S.csv -NoTypeInformation -UseCulture  
[vSphere PowerCLI] C:\Scripts>
```

Note that with this final cmdlet in our pipeline there was no output to the console, simply a return to the PowerShell prompt. This is an experience that you should get used to working in PowerShell. While it can be quite strange (having gotten accustomed to console output to the screen during the previous three steps), it typically means that the command completed successfully without errors. If an error occurred, you will soon know about it. Bright red text explaining what error has occurred will be displayed.

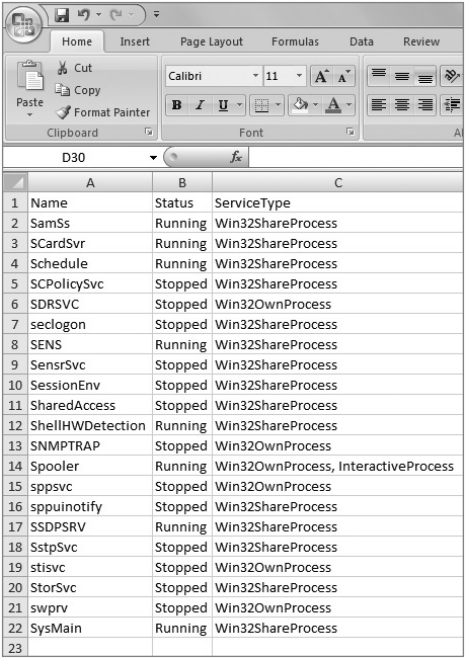
Let's check the CSV file to see if the expected data is there. You can even open the CSV file from PowerShell with the Invoke-Expression cmdlet. Invoke-Expression is the equivalent of double-clicking on a file in Windows Explorer and having Windows open it in the associated application.

```
Invoke-Expression .\Services_S.csv
```

Figure A.2 shows the resulting CSV.

So, the pipeline is complete. You have seen how straightforward it is to gradually build a set of cmdlets into one line of code and produce results. By working in this logical fashion, it is possible to examine the results at each stage and stay on the right track.

FIGURE A.2 Results of pipeline in CSV file



	A	B	C
1	Name	Status	ServiceType
2	SamSs	Running	Win32ShareProcess
3	SCardSvr	Running	Win32ShareProcess
4	Schedule	Running	Win32ShareProcess
5	SCPolicySvc	Stopped	Win32ShareProcess
6	SDRSVC	Stopped	Win32OwnProcess
7	seclogon	Stopped	Win32ShareProcess
8	SENS	Running	Win32ShareProcess
9	SensrSvc	Stopped	Win32ShareProcess
10	SessionEnv	Stopped	Win32ShareProcess
11	SharedAccess	Stopped	Win32ShareProcess
12	ShellHWDetection	Running	Win32ShareProcess
13	SNMPTRAP	Stopped	Win32OwnProcess
14	Spooler	Running	Win32OwnProcess, InteractiveProcess
15	sppsvc	Stopped	Win32OwnProcess
16	sppuotify	Stopped	Win32ShareProcess
17	SSDPDRV	Running	Win32ShareProcess
18	SstpSvc	Stopped	Win32ShareProcess
19	stisvc	Stopped	Win32OwnProcess
20	StorSvc	Stopped	Win32ShareProcess
21	swprv	Stopped	Win32OwnProcess
22	SysMain	Running	Win32ShareProcess
23			

Build Your Toolkit of Basic Cmdlets

When PowerShell 1.0 was released, over 100 cmdlets were available as standard. When PowerShell version 2 was released, this figure increased to over 200 cmdlets. On top of this, Window Server 2008 R2 contains roles and features that when installed include PowerShell modules, such as Active Directory, Failover Clustering, and IIS. A PowerShell module in this case is a set of cmdlets grouped together and made available when that role or feature is installed.

With this many PowerShell cmdlets available, it can be difficult to know where to start. Table A.1 lists 10 standard cmdlets to get you started.

TABLE A.1 Ten standard cmdlets

Cmdlet	Description
Get-Help	Find help information on any PowerShell cmdlet or topic via the <i>About_topics</i> .
Where-Object	Filter the list of returned objects based on specific criteria.
Select-Object	Select specific properties from the returned object.
Group-Object	Gather together objects that contain the same value for specified properties.
Measure-Object	Calculate basic statistics like Average and Sum from numeric properties of objects.
Foreach-Object	Loop through a collection of objects carrying out an action each time.
Get-Date	Return the current date and time, or carry out basic date calculations.
Get-Content	Read in data from files, such as a text or XML file, and create objects.
Get-WmiObject	Query WMI classes.
Invoke-Command	Execute commands on local or remote computers.

These cmdlets are enough for you to get started and give you a good feel for the automation tasks that you can achieve with PowerShell.

Reporting

One of the most popular use cases for PowerShell is generating reports. As administrators, we all know how management likes to receive reports, and there are a number of standard PowerShell cmdlets that are incredibly useful for reporting. You have already seen how `Export-CSV` can take objects from a pipeline of cmdlets and turn them into data in a CSV file; let's see what else is available (Table A.2).

TABLE A.2 Data export and conversion cmdlets

Cmdlet	Description
Export-CSV	Generate a CSV file report.
Export-Clixml	Generate an XML file report.
ConvertTo-XML	Generate an XML-based representation of an object.
ConvertTo-HTML	Generate an HTML file report.

These cmdlets also accept pipeline input. So, choose the format you prefer and then replace `Export-CSV` with one of these cmdlets at the end of your pipeline. You will be able to produce a report in a different format.

Notice that there are two similar XML cmdlets. `ConvertTo-XML` was introduced in PowerShell version 2 and is better used for creating XML reports. `Export-Clixml` sounds like the more natural choice if you have been using `Export-CSV`, but it was intended to be used in conjunction with the `Import-Clixml` cmdlet to re-create an original object from a file.

To use the `ConvertTo-XML` cmdlet to generate an XML report, save the results to a variable. Then, use the `Save` method to export it to a file, as in the example code that follows:

```
$xml = Get-Service | Where-Object{$_ .Name -like 'b*'} |  
    ConvertTo-Xml  
$xml.Save("C:\temp\service.xml")
```

Enhancing Script Writing

At this point, you've started to work at the PowerShell command line; you're thinking about objects, properties, and methods; and you've put together a few cmdlets into a pipeline. The next step is to take the commands and save them into a script to make reusable code. When you save code that you've used already, it becomes instantly available the next time you need it. As a bonus, you can share your saved script with colleagues or the wider PowerShell community.

PowerShell scripts are stored in a *.ps1 file, and making your script reusable is as simple as copying your code into one of these files. By default, *.ps1 files open in Notepad rather than executing in PowerShell. This is a security feature implemented by the PowerShell team in reaction to Microsoft's experience with viruses being distributed in VBScript files that executed when opened. By opening the file in Notepad, there is no chance of the code being accidentally executed with a simple double-click.

Using Commercial Editors

Notepad, however, is not the ideal editor for writing PowerShell scripts. It has very few features that experienced script writers expect. Consequently, a number of commercial editors are available that enable you to write better scripts more easily.

PowerShell ISE With the release of PowerShell version 2, Microsoft included the PowerShell ISE. This tool contains both an interactive console and a basic script editor that has features such as syntax highlighting, multiple tabs, and script debugging.

PowerGUI This tool is available in both free and commercial versions. The free version has syntax highlighting, code completion, hints, code snippets, debugging, and real-time syntax checking. The commercial version features additional enterprise-style features such as MobileShell, which executes scripts from any device, and version control. Both versions also feature the ability to extend the editor with your own add-ons if a feature you require is not yet available. (You might also be aware of the vEcoShell script editor. It is essentially the same editor as PowerGUI. You can learn more in Chapter 21, “PowerGUI and vEcoShell.”)

PowerShell Plus This commercial editor includes all the standard editor features mentioned so far and has selling point features, such as an interactive console and an Interactive Learning Center. It also ships with hundreds of free scripts.

PrimalScript This commercial editor includes support for VBScript and JScript in addition to PowerShell. Other selling points are a datastore browser and script packaging as executables.

All of these editors, if not free, have evaluation versions so it is well worth trying out each one.

PowerShell version 2 introduced a number of features that make it easier for script writers to document their functions and scripts for others and to author functions that perform similarly to fully compiled cmdlets.

Adding Help to Your Scripts and Functions

We’ve already looked at the built-in help for cmdlets with PowerShell. Beginning with version 2, it’s possible to add help to your own functions and scripts. It is a good way to self-document, and it makes your code easier for others to use. The script examples throughout the book use this feature. Adding help is as simple as adding text in the format shown next to the top of your function or script:

```
<#
.SYNOPSIS
    A Synopsis of the script.
.DESCRIPTION
    A Description of the script
.NOTES
    Authors:  Script Author Name
.PARAMETER Parameter1
    A note about the first parameter
.PARAMETER Parameter2
    A note about the second parameter
```



```
.EXAMPLE
    One or more examples for how it works
#>
```

By utilizing this format, anyone who uses the function or the script will be able to type `Get-Help ScriptName.ps1` and access the help information with all the built-in help functionality. For example, you could type

```
Get-Help ScriptName.ps1 -Examples
```

and then look at just the examples. This is a fantastic way to document your scripts and make them easy to share.

Creating Script Cmdlets

You will also notice that a lot of the scripts in this book use another new PowerShell version 2 feature—Advanced Functions. This feature enables you to create functions that behave much like cmdlets. In fact, during the beta phases of PowerShell version 2, they were called script cmdlets. You can include a lot of features related to parameters, such as making them mandatory or forcing them to be of a particular type, like a string or an integer. You can also bind values from the pipeline into the parameters of the function. These features are beyond the scope of an introduction to PowerShell, but you should definitely consider using them as you become more advanced with PowerShell. Find out more by typing **`Get-Help about_functions_advanced`** in PowerShell.

Creating Modules

Finally, modules are another PowerShell version 2 feature to consider for enhancing your script writing. Previously in PowerShell version 1, a third party would supply their cmdlets (such as PowerCLI) in a compiled form known as a snap-in. With PowerShell version 2, you can package functions and scripts into a module for easy distribution to colleagues or the community. Again, this is beyond the scope of an introduction to PowerShell, but you can find more information by typing **`Get-Help about_modules`** in PowerShell.

