CHAPTER 3

PowerShell: A More In-Depth Look

IN THIS CHAPTER

- Introduction
- Object Based
- Understanding Providers
- Understanding Errors
- Error Handling
- PowerShell Profiles
- Understanding Security
- The PowerShell Language

Introduction

This chapter delves into some specifics of how PowerShell works that you need to understand for the later scripting chapters. Try not to get too bogged down in details; instead, focus on understanding the concepts. Because PowerShell is a change from Windows scripting of the past, you might also need to change your scripting methods. With practice, it will start to feel as familiar as Windows scripting via VBScript or JScript, which was the standard method for Windows automation tasks.

Object Based

Most shells operate in a text-based environment, which means you typically have to manipulate the output for automation purposes. For example, if you need to pipe data from one command to the next, the output from the first command usually must be reformatted to meet the second command's requirements. Although this method has worked for years, dealing with text-based data can be difficult and frustrating.

Often, a lot of work is necessary to transform text data into a usable format. Microsoft has set out to change the standard with PowerShell, however. Instead of transporting data as plain text, PowerShell retrieves data in the form of .NET Framework objects, which makes it possible for commands (cmdlets) to access object properties and methods directly. This change has simplified shell use. Instead of modifying text data, you can just refer to the

required data by name. Similarly, instead of writing code to transform data into a usable format, you can simply refer to objects and manipulate them as needed.

Understanding the Pipeline

The use of objects gives you a more robust method for dealing with data. In the past, data was transferred from one command to the next by using the pipeline, which makes it possible to string a series of commands together to gather information from a system. However, as mentioned previously, most shells have a major disadvantage: The information gathered from commands is text based. Raw text needs to be parsed (transformed) into a format the next command can understand before being piped. To see how parsing works, take a look at the following Bash example:

```
$ ps -ef | grep "bash" | cut -f2
```

The goal is to get the process ID (PID) for the `bash` process. A list of currently running processes is gathered with the `ps` command and then piped to the `grep` command and filtered on the string `"bash"`. Next, the remaining information is piped to the `cut` command, which returns the second field containing the PID based on a tab delimiter.

NOTE

A **delimiter** is a character used to separate data fields. The default delimiter for the `cut` command is a tab. If you want to use a different delimiter, use the `-d` parameter.

Based on the `man` information for the `grep` and `cut` commands, it seems as though the `ps` command should work. However, the PID isn't returned or displayed in the correct format.

The command doesn't work because the Bash shell requires you to manipulate text data to display the PID. The output of the `ps` command is text based, so transforming the text into a more usable format requires a series of other commands, such as `grep` and `cut`. Manipulating text data makes this task more complicated. For example, to retrieve the PID from the data piped from the `grep` command, you need to provide the field location and the delimiter for separating text information to the `cut` command. To find this information, run the first part of the `ps` command:

```
$ ps -ef | grep "bash"
   bob    3628       1 con  16:52:46 /usr/bin/bash
```

The field you need is the second one (3628). Notice that the `ps` command doesn't use a tab delimiter to separate columns in the output; instead, it uses a variable number of spaces or a whitespace delimiter, between fields.

NOTE

A **whitespace delimiter** consists of characters, such as spaces or tabs, that equate to blank space.

The `cut` command has no way to tell that spaces should be used as a field separator, which is why the command doesn't work. To get the PID, you need to use the awk scripting language. The command and output in that language would look like this:

```
$ ps -ef | grep "bash" | awk '{print $2}'
3628
```


The point is that although most UNIX and Linux shell commands are powerful, using them can be complicated and frustrating. Because these shells are text-based, often commands lack functionality or require using additional commands or tools to perform tasks. To address the differences in text output from shell commands, many utilities and scripting languages have been developed to parse text.

The result of all this parsing is a tree of commands and tools that make working with shells unwieldy and time consuming, which is one reason for the proliferation of management interfaces that rely on GUIs. This trend can be seen among tools Windows administrators use, too; as Microsoft has focused on enhancing the management GUI at the expense of the CLI.

Windows administrators now have access to the same automation capabilities as their UNIX and Linux counterparts. However, PowerShell and its use of objects fill the automation need Windows administrators have had since the days of batch scripting and WSH in a more usable and less parsing intense manner. To see how the PowerShell pipeline works, take a look at the following PowerShell example:

```
PS C:\> get-process bash | format-table id -autosize

  Id
  --
3628


PS C:\>
```

Like the Bash example, the goal of this PowerShell example is to display the PID for the `bash` process. First, information about the `bash` process is gathered by using the `Get-Process` cmdlet. Second, the information is piped to the `Format-Table` cmdlet, which returns a table containing only the PID for the `bash` process.

The Bash example requires complex shell scripting, but the PowerShell example simply requires formatting a table. As you can see, the structure of PowerShell cmdlets is much easier to understand and use.

Now that you have the PID for the `bash` process, take a look at the following example, which shows how to kill (stop) that process:

```
PS C:\> get-process bash | stop-process
PS C:\>
```

.NET Framework Tips

Before continuing, you need to know a few points about how PowerShell interacts with the .NET Framework. This information is critical to understanding the scripts you review in later chapters.

`New-Object` cmdlet

You use the `New-Object` cmdlet to create an instance of a .NET object. To do this, you simply provide the fully qualified name of the .NET class you want to use, as shown:

```
PS C:\> $Ping = new-object Net.NetworkInformation.Ping
PS C:\>
```

By using the `New-Object` cmdlet, you now have an instance of the `Ping` class that enables you to detect whether a remote computer can be reached via Internet Control Message Protocol (ICMP). Therefore, you have an object-based version of the `Ping.exe` command-line tool.

If you're wondering what the replacement is for the VBScript `CreateObject` method, it's the `New-Object` cmdlet. You can also use the `comObject` switch with this cmdlet to create a COM object, simply by specifying the object's programmatic identifier (ProgID), as shown here:

```
PS C:\> $IE = new-object -comObject InternetExplorer.Application
PS C:\> $IE.Visible=$True
PS C:\> $IE.Navigate("www.cnn.com")
PS C:\>
```

Square Brackets

Throughout this book, you'll notice the use of square brackets ([and]), which indicate that the enclosed term is a .NET Framework reference. These references can be one of the following:

- *A fully qualified class name*—`[System.DirectoryServices.ActiveDirectory.Forest]`, for example

- *A class in the* `System` *namespace*—`[string]`, `[int]`, `[boolean]`, and so forth
- *A type accelerator*—`[ADSI]`, `[WMI]`, `[Regex]`, and so on

NOTE

Chapter 8, “PowerShell and WMI,” explains type accelerators in more detail.

Defining a variable is a good example of when to use a .NET Framework reference. In this case, the variable is assigned an enumeration value by using an explicit cast of a .NET class, as shown in this example:

```
PS C:\> $SomeNumber = [int]1
PS C:\> $Identity = [System.Security.Principal.NTAccount]"Administrator"
PS C:\>
```

If an enumeration can consist of only a fixed set of constants, and you don’t know these constants, you can use the `System.Enum` class’s `GetNames` method to find this information:

```
PS C:\>
[enum]::GetNames([System.Security.AccessControl.FileSystemRights])
ListDirectory
ReadData
WriteData
CreateFiles
CreateDirectories
AppendData
ReadExtendedAttributes
WriteExtendedAttributes
Traverse
ExecuteFile
DeleteSubdirectoriesAndFiles
ReadAttributes
WriteAttributes
Write
Delete
ReadPermissions
Read
ReadAndExecute
Modify
ChangePermissions
TakeOwnership
Synchronize
FullControl
PS C:\>
```

Static Classes and Methods

Square brackets are used not only for defining variables, but also for using or calling static members of a .NET class. To do this, just use a double colon (`::`) between the class name and the static method or property, as shown in this example:

```
PS C:\> [System.DirectoryServices.ActiveDirectory.Forest]::
GetCurrentForest()


Name                  : taosage.internal
Sites                 : {HOME}
Domains               : {taosage.internal}
GlobalCatalogs        : {sol.taosage.internal}
ApplicationPartitions : {DC=DomainDnsZones,DC=taosage,DC=internal,
DC=ForestDns
                        Zones,DC=taosage,DC=internal}
ForestMode            : Windows2003Forest
RootDomain            : taosage.internal
Schema                :
CN=Schema,CN=Configuration,DC=taosage,DC=internal
SchemaRoleOwner       : sol.taosage.internal
NamingRoleOwner       : sol.taosage.internal



PS C:\>
```

Reflection

Reflection is a feature in the .NET Framework that enables developers to examine objects and retrieve their supported methods, properties, fields, and so on. Because PowerShell is built on the .NET Framework, it provides this feature, too, with the `Get-Member` cmdlet. This cmdlet analyzes an object or collection of objects you pass to it via the pipeline. For example, the following command analyzes the objects returned from the `Get-Process` cmdlet and displays their associated properties and methods:

```
PS C:\> get-process | get-member
```

Developers often refer to this process as “interrogating” an object. It’s a faster way to get information about objects than using the `Get-Help` cmdlet (which at the time of this writing provides limited information), reading the MSDN documentation, or searching the Internet.

```
PS C:\> get-process | get-member


   TypeName: System.Diagnostics.Process

Name                           MemberType     Definition
----                           ----------     ----------
Handles                        AliasProperty  Handles = Handlecount
Name                           AliasProperty  Name = ProcessName
NPM                            AliasProperty  NPM = NonpagedSystemMemorySize
PM                             AliasProperty  PM = PagedMemorySize
VM                             AliasProperty  VM = VirtualMemorySize
WS                             AliasProperty  WS = WorkingSet
add_Disposed                   Method         System.Void add_Disposed(Event...
add_ErrorDataReceived          Method         System.Void add_ErrorDataRecei...
add_Exited                     Method         System.Void add_Exited(EventHa...
add_OutputDataReceived         Method         System.Void add_OutputDataRece...
BeginErrorReadLine             Method         System.Void BeginErrorReadLine()
BeginOutputReadLine            Method         System.Void BeginOutputReadLine()
CancelErrorRead                Method         System.Void CancelErrorRead()
CancelOutputRead               Method         System.Void CancelOutputRead()
Close                          Method         System.Void Close()
CloseMainWindow                Method         System.Boolean CloseMainWindow()
CreateObjRef                   Method         System.Runtime.Remoting.ObjRef...
Dispose                        Method         System.Void Dispose()
Equals                         Method         System.Boolean Equals(Object obj)
get_BasePriority               Method         System.Int32 get_BasePriority()
get_Container                  Method         System.ComponentModel.IContain...
get_EnableRaisingEvents        Method         System.Boolean get_EnableRaisi...
...
__NounName                     NoteProperty   System.String __NounName=Process
BasePriority                   Property       System.Int32 BasePriority {get;}
Container                      Property       System.ComponentModel.IContain...
EnableRaisingEvents            Property       System.Boolean EnableRaisingEv...
ExitCode                       Property       System.Int32 ExitCode {get;}
ExitTime                       Property       System.DateTime ExitTime {get;}
Handle                         Property       System.IntPtr Handle {get;}
HandleCount                    Property       System.Int32 HandleCount {get;}
HasExited                      Property       System.Boolean HasExited {get;}
Id                             Property       System.Int32 Id {get;}
MachineName                    Property       System.String MachineName {get;}
MainModule                     Property       System.Diagnostics.ProcessModu...
MainWindowHandle               Property       System.IntPtr MainWindowHandle...
MainWindowTitle                Property       System.String MainWindowTitle ...
MaxWorkingSet                  Property       System.IntPtr MaxWorkingSet {g...
MinWorkingSet                  Property       System.IntPtr MinWorkingSet {g...
...
Company                        ScriptProperty System.Object Company {get=$th...
CPU                            ScriptProperty System.Object CPU {get=$this.T...
Description                    ScriptProperty System.Object Description {get...
FileVersion                    ScriptProperty System.Object FileVersion {get...
Path                           ScriptProperty System.Object Path {get=$this....
Product                        ScriptProperty System.Object Product {get=$th...
ProductVersion                 ScriptProperty System.Object ProductVersion {...


PS C:\>
```

3

This example shows that objects returned from the `Get-Process` cmdlet have additional property information that you didn't know. The following example uses this information to produce a report about Microsoft-owned processes and their folder locations. An example of such a report would be as follows:

```
PS C:\> get-process | where-object {$_.Company -match ".*Microsoft*"} |
format-table Name, ID, Path -Autosize

Name         Id Path
----         -- ----
ctfmon     4052 C:\WINDOWS\system32\ctfmon.exe
explorer   3024 C:\WINDOWS\Explorer.EXE
iexplore   2468 C:\Program Files\Internet Explorer\iexplore.exe
iexplore   3936 C:\Program Files\Internet Explorer\iexplore.exe
mobsync     280 C:\WINDOWS\system32\mobsync.exe
notepad    1600 C:\WINDOWS\system32\notepad.exe
notepad    2308 C:\WINDOWS\system32\notepad.exe
notepad    2476 C:\WINDOWS\system32\NOTEPAD.EXE
notepad    2584 C:\WINDOWS\system32\notepad.exe
OUTLOOK    3600 C:\Program Files\Microsoft Office\OFFICE11\OUTLOOK.EXE
powershell 3804 C:\Program Files\Windows PowerShell\v1.0\powershell.exe
WINWORD    2924 C:\Program Files\Microsoft Office\OFFICE11\WINWORD.EXE


PS C:\>
```

You wouldn't get nearly this much process information by using WSH with only a single line of code.

The `Get-Member` cmdlet isn't just for objects generated from PowerShell cmdlets. You can also use it on objects initialized from .NET classes, as shown in this example:

```
PS C:\> new-object System.DirectoryServices.DirectorySearcher
```

The goal of using the `DirectorySearcher` class is to retrieve user information from Active Directory, but you don't know what methods the returned objects support. To retrieve this information, run the `Get-Member` cmdlet against a variable containing the mystery objects, as shown in this example.

```
PS C:\> $Searcher = new-object System.DirectoryServices.DirectorySearcher
PS C:\> $Searcher | get-member


   TypeName: System.DirectoryServices.DirectorySearcher

Name                          MemberType Definition
----                          ---------- ----------
add_Disposed                  Method     System.Void add_Disposed(EventHandle...
```

```
CreateObjRef                 Method     System.Runtime.Remoting.ObjRef Creat...
Dispose                      Method     System.Void Dispose()
Equals                       Method     System.Boolean Equals(Object obj)
FindAll                      Method     System.DirectoryServices.SearchResul...
FindOne                      Method     System.DirectoryServices.SearchResul...
...
Asynchronous                 Property   System.Boolean Asynchronous {get;set;}
AttributeScopeQuery          Property   System.String AttributeScopeQuery {g...
CacheResults                 Property   System.Boolean CacheResults {get;set;}
ClientTimeout                Property   System.TimeSpan ClientTimeout {get;s...
Container                    Property   System.ComponentModel.IContainer Con...
DerefAlias                   Property   System.DirectoryServices.Dereference...
DirectorySynchronization     Property   System.DirectoryServices.DirectorySy...
ExtendedDN                   Property   System.DirectoryServices.ExtendedDN ...
Filter                       Property   System.String Filter {get;set;}
PageSize                     Property   System.Int32 PageSize {get;set;}
PropertiesToLoad             Property   System.Collections.Specialized.Strin...
PropertyNamesOnly            Property   System.Boolean PropertyNamesOnly {ge...
ReferralChasing              Property   System.DirectoryServices.ReferralCha...
SearchRoot                   Property   System.DirectoryServices.DirectoryEn...
SearchScope                  Property   System.DirectoryServices.SearchScope...
SecurityMasks                Property   System.DirectoryServices.SecurityMas...
ServerPageTimeLimit          Property   System.TimeSpan ServerPageTimeLimit ...
ServerTimeLimit              Property   System.TimeSpan ServerTimeLimit {get...
Site                         Property   System.ComponentModel.ISite Site {ge...
SizeLimit                    Property   System.Int32 SizeLimit {get;set;}
Sort                         Property   System.DirectoryServices.SortOption ...
Tombstone                    Property   System.Boolean Tombstone {get;set;}
VirtualListView              Property   System.DirectoryServices.
                                        DirectoryVi...


PS C:\>
```

Notice the `FindAll` method and the `Filter` property. These are object attributes that can be used to search for information about users in an Active Directory domain. To use these attributes the first step is to filter the information returned from `DirectorySearcher` by using the `Filter` property, which takes a filter statement similar to what you'd find in a Lightweight Directory Access Protocol (LDAP) statement:

```
PS C:\> $Searcher.Filter = ("(objectCategory=user)")
```

Next, you retrieve all users from the Active Directory domain with the `FindAll` method:

```
PS C:\> $Users = $Searcher.FindAll()
```

At this point, the `$Users` variable contains a collection of objects holding the distinguished names for all users in the Active Directory domain:

```
PS C:\> $Users

Path                                        Properties
----                                        ----------
LDAP://CN=Administrator,CN=Users,DC=... {homemdb, samaccounttype, countrycod...
LDAP://CN=Guest,CN=Users,DC=taosage,... {samaccounttype, objectsid, whencrea...
LDAP://CN=krbtgt,CN=Users,DC=taosage... {samaccounttype, objectsid, whencrea...
LDAP://CN=admintyson,OU=Admin Accoun... {countrycode, cn, lastlogoff, usncre...
LDAP://CN=servmom,OU=Service Account... {samaccounttype, lastlogontimestamp,...
LDAP://CN=SUPPORT_388945a0,CN=Users,... {samaccounttype, objectsid, whencrea...
LDAP://CN=Tyson,OU=Acc... {msmqsigncertificates, distinguished...
LDAP://CN=Maiko,OU=Acc... {homemdb, msexchhomeservername, coun...
LDAP://CN=servftp,OU=Service Account... {samaccounttype, lastlogontimestamp,...
LDAP://CN=Erica,OU=Accounts,OU... {samaccounttype, lastlogontimestamp,...
LDAP://CN=Garett,OU=Accou... {samaccounttype, lastlogontimestamp,...
LDAP://CN=Fujio,OU=Accounts,O... {samaccounttype, givenname, sn, when...
LDAP://CN=Kiyomi,OU=Accounts,... {samaccounttype, givenname, sn, when...
LDAP://CN=servsql,OU=Service Account... {samaccounttype, lastlogon, lastlogo...
LDAP://CN=servdhcp,OU=Service Accoun... {samaccounttype, lastlogon, lastlogo...
LDAP://CN=servrms,OU=Service Account... {lastlogon, lastlogontimestamp, msmq...


PS C:\>
```

NOTE

The commands in these examples use the default connection parameters for the `DirectorySearcher` class. This means the connection to Active Directory uses the default naming context. If you want to connect to a domain other than the one specified in the default naming context, you must set the appropriate connection parameters.

Now that you have an object for each user, you can use the `Get-Member` cmdlet to learn what you can do with these objects:

```
PS C:\> $Users | get-member


   TypeName: System.DirectoryServices.SearchResult

Name                 MemberType Definition
----                 ---------- ----------
Equals               Method     System.Boolean Equals(Object obj)
get_Path             Method     System.String get_Path()
get_Properties       Method     System.DirectoryServices.ResultPropertyCollecti...
GetDirectoryEntry    Method     System.DirectoryServices.DirectoryEntry GetDire...
GetHashCode          Method     System.Int32 GetHashCode()
```

```
GetType            Method     System.Type GetType()
ToString           Method     System.String ToString()
Path               Property   System.String Path {get;}
Properties         Property   System.DirectoryServices.ResultPropertyCollecti...


PS C:\>
```

To collect information from these user objects, it seems as though you need to step through each object with the `GetDirectoryEntry` method. To determine what data you can retrieve from these objects, you use the `Get-Member` cmdlet again, as shown here:


```
PS C:\> $Users[0].GetDirectoryEntry() | get-member -MemberType Property


   TypeName: System.DirectoryServices.DirectoryEntry

Name                            MemberType Definition
----                            ---------- ----------
accountExpires                  Property   System.DirectoryServices.Property...
adminCount                      Property   System.DirectoryServices.Property...
badPasswordTime                 Property   System.DirectoryServices.Property...
badPwdCount                     Property   System.DirectoryServices.Property...
cn                              Property   System.DirectoryServices.Property...
codePage                        Property   System.DirectoryServices.Property...
countryCode                     Property   System.DirectoryServices.Property...
description                     Property   System.DirectoryServices.Property...
displayName                     Property   System.DirectoryServices.Property...
distinguishedName               Property   System.DirectoryServices.Property...
homeMDB                         Property   System.DirectoryServices.Property...
homeMTA                         Property   System.DirectoryServices.Property...
instanceType                    Property   System.DirectoryServices.Property...
isCriticalSystemObject          Property   System.DirectoryServices.Property...
lastLogon                       Property   System.DirectoryServices.Property...
lastLogonTimestamp              Property   System.DirectoryServices.Property...
legacyExchangeDN                Property   System.DirectoryServices.Property...
logonCount                      Property   System.DirectoryServices.Property...
mail                            Property   System.DirectoryServices.Property...
mailNickname                    Property   System.DirectoryServices.Property...
mDBUseDefaults                  Property   System.DirectoryServices.Property...
memberOf                        Property   System.DirectoryServices.Property...
msExchALObjectVersion           Property   System.DirectoryServices.Property...
msExchHomeServerName            Property   System.DirectoryServices.Property...
msExchMailboxGuid               Property   System.DirectoryServices.Property...
msExchMailboxSecurityDescriptor Property   System.DirectoryServices.Property...
msExchPoliciesIncluded          Property   System.DirectoryServices.Property...
msExchUserAccountControl        Property   System.DirectoryServices.Property...
mSMQDigests                     Property   System.DirectoryServices.Property...
mSMQSignCertificates            Property   System.DirectoryServices.Property...
name                            Property   System.DirectoryServices.Property...
```

```
nTSecurityDescriptor          Property   System.DirectoryServices.Property...
objectCategory                Property   System.DirectoryServices.Property...
objectClass                   Property   System.DirectoryServices.Property...
objectGUID                    Property   System.DirectoryServices.Property...
objectSid                     Property   System.DirectoryServices.Property...
primaryGroupID                Property   System.DirectoryServices.Property...
proxyAddresses                Property   System.DirectoryServices.Property...
pwdLastSet                    Property   System.DirectoryServices.Property...
sAMAccountName                Property   System.DirectoryServices.Property...
sAMAccountType                Property   System.DirectoryServices.Property...
showInAddressBook             Property   System.DirectoryServices.Property...
textEncodedORAddress          Property   System.DirectoryServices.Property...
userAccountControl            Property   System.DirectoryServices.Property...
uSNChanged                    Property   System.DirectoryServices.Property...
uSNCreated                    Property   System.DirectoryServices.Property...
whenChanged                   Property   System.DirectoryServices.Property...
whenCreated                   Property   System.DirectoryServices.Property...


PS C:\>
```

NOTE

The `MemberType` parameter tells the `Get-Member` cmdlet to retrieve a specific type of member. For example, to display the methods associated with an object, use the `get-member –MemberType Method` command.

To use PowerShell effectively, you should make sure you're familiar with the `Get-Member` cmdlet. If you don't understand how it works, figuring out what an object can and can't do may be at times difficult.

Now that you understand how to pull information from Active Directory, it's time to put together all the commands used so far:

```
PS C:\> $Searcher = new-object System.DirectoryServices.DirectorySearcher
PS C:\> $Searcher.Filter = ("(objectCategory=user)")
PS C:\> $Users = $Searcher.FindAll()
PS C:\> foreach ($User in $Users){$User.GetDirectoryEntry().sAMAccountName}
Administrator
Guest
krbtgt
admintyson
servmom
SUPPORT_388945a0
Tyson
Maiko
servftp
Erica
Garett
```

```
Fujio
Kiyomi
servsql
servdhcp
servrms
PS C:\>
```

Although the list of users in this domain isn't long, it shows that you can interrogate a set of objects to understand their capabilities.

The same is true for static classes, however, when attempting to use the `Get-Member` cmdlet in the same manner as before creates the following error:

```
PS C:\> new-object System.Net.Dns
New-Object : Constructor not found. Cannot find an appropriate constructor for
type System.Net.Dns.
At line:1 char:11
+ New-Object  <<<< System.Net.Dns
PS C:\>
```

As you can see, the `System.Net.Dns` class doesn't have a constructor, which poses a challenge when you're trying to find out what this class does. However, the `Get-Member` cmdlet can handle this challenge. With the `Static` parameter, you can gather information from static classes, as shown in this example:

```
PS C:\> [System.Net.Dns] | get-member -Static


   TypeName: System.Net.Dns

Name                    MemberType Definition
----                    ---------- ----------
BeginGetHostAddresses   Method     static System.IAsyncResult BeginGetHostAddr...
BeginGetHostByName      Method     static System.IAsyncResult BeginGetHostByNa...
BeginGetHostEntry       Method     static System.IAsyncResult BeginGetHostEntr...
BeginResolve            Method     static System.IAsyncResult BeginResolve(Str...
EndGetHostAddresses     Method     static System.Net.IPAddress[] EndGetHostAdd...
EndGetHostByName        Method     static System.Net.IPHostEntry EndGetHostByN...
EndGetHostEntry         Method     static System.Net.IPHostEntry EndGetHostEnt...
EndResolve              Method     static System.Net.IPHostEntry EndResolve(IA...
Equals                  Method     static System.Boolean Equals(Object objA, O...
GetHostAddresses        Method     static System.Net.IPAddress[] GetHostAddres...
GetHostByAddress        Method     static System.Net.IPHostEntry GetHostByAddr...
GetHostByName           Method     static System.Net.IPHostEntry GetHostByName...
GetHostEntry            Method     static System.Net.IPHostEntry GetHostEntry(...
GetHostName             Method     static System.String GetHostName()
```

```
ReferenceEquals       Method     static System.Boolean ReferenceEquals(Objec...
Resolve               Method     static System.Net.IPHostEntry Resolve(Strin...


PS C:\>
```

Now that you have information about the `System.Net.Dns` class, you can put it to work. As an example, use the `GetHostAddress` method to resolve the IP address for the Web site www.digg.com:

```
PS C:\> [System.Net.Dns]::GetHostAddresses("www.digg.com")


IPAddressToString : 64.191.203.30
Address           : 516669248
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False


PS C:\>
```

NOTE

As you have seen, the Get-Member cmdlet can be a powerful tool. It can also be time consuming because it's easy to spend hours exploring what you can do with different cmdlets and classes. To help prevent Get-Member User Stress Syndrome (GUSS), try to limit your discovery sessions to no more than a couple of hours a day.

Extended Type System (ETS)

You might think that scripting in PowerShell is typeless because you rarely need to specify the type for a variable. PowerShell is actually type driven, however, because it interfaces with different types of objects from the less than perfect .NET to Windows Management Instrumentation (WMI), Component Object Model (COM), ActiveX Data Objects (ADO), Active Directory Service Interfaces (ADSI), Extensible Markup Language (XML), and even custom objects. However, you typically don't need to be concerned about object types because PowerShell adapts to different object types and displays its interpretation of an object for you.

In a sense, PowerShell tries to provide a common abstraction layer that makes all object interaction consistent, despite the type. This abstraction layer is called the `PSObject`, a common object used for all object access in PowerShell. It can encapsulate any base object (.NET, custom, and so on), any instance members, and implicit or explicit access to adapted and type-based extended members, depending on the type of base object.

Furthermore, it can state its type and add members dynamically. To do this, PowerShell uses the **Extended Type System (ETS)**, which provides an interface that allows PowerShell cmdlet and script developers to manipulate and change objects as needed.

NOTE

When you use the `Get-Member` cmdlet, the information returned is from `PSObject`. Sometimes `PSObject` blocks members, methods, and properties from the original object. If you want to view the blocked information, use the `BaseObject` property with the `PSBase` standard name. For example, you could use the `$Procs.PSBase | get-member` command to view blocked information for the `$Procs` object collection.

Needless to say, this topic is fairly advanced, as `PSBase` is hidden from view. The only time you should need to use it is when the `PSObject` doesn't interpret an object correctly or you're digging around for hidden jewels in PowerShell.

3

Therefore, with ETS, you can change objects by adapting their structure to your requirements or create new ones. One way to manipulate objects is to adapt (extend) existing object types or create new object types. To do this, you define custom types in a custom types file, based on the structure of the default types file, `Types.ps1xml`.

In the `Types.ps1xml` file, all types are contained in a `<Type></Type>` node, and each type can contain standard members, data members, and object methods. Using this structure as a basis, you can create your own custom types file and load it into a PowerShell session by using the `Update-TypeData` cmdlet, as shown here:

```
PS C:\> Update-TypeData D:\PS\My.Types.Ps1xml
```

You can run this command manually during each PowerShell session or add it to your `profile.ps1` file.

CAUTION

The `Types.ps1xml` file defines default behaviors for all object types in PowerShell. Do *not* modify this file for any reason. Doing so might prevent PowerShell from working, resulting in a "Game over"!

The second way to manipulate an object's structure is to use the `Add-Member` cmdlet to add a user-defined member to an existing object instance, as shown in this example:

```
PS C:\> $Procs = get-process
PS C:\> $Procs | add-member -Type scriptProperty "TotalDays" {
>> $Date = get-date
>> $Date.Subtract($This.StartTime).TotalDays}
>>
PS C:\>
```

This code creates a `scriptProperty` member called `TotalDays` for the collection of objects in the `$Procs` variable. The `scriptProperty` member can then be called like any other member for those objects, as shown in the next example:

NOTE

The `$This` variable represents the current object when you're creating a script method.

```
PS C:\> $Procs | where {$_.name -Match "WINWORD"} | ft Name,
TotalDays -AutoSize

Name             TotalDays
----             ---------
WINWORD 5.1238899696898148


PS C:\>
```

Although the new `scriptProperty` member isn't particularly useful, it does demonstrate how to extend an object. Being able to extend objects from both a scripting and cmdlet development context is extremely useful.

Understanding Providers

Most computer systems are used to store data, often in a structure such as a file system. Because of the amount of data stored in these structures, processing and finding information can be unwieldy. Most shells have interfaces, or **providers**, for interacting with data stores in a predictable, set manner. PowerShell also has a set of providers for presenting the contents of data stores through a core set of cmdlets. You can then use these cmdlets to browse, navigate, and manipulate data from stores through a common interface. To get a list of the core cmdlets, use the following command:

```
PS C:\> help about_core_commands
…
    ChildItem CMDLETS
    Get-ChildItem

    CONTENT CMDLETS
    Add-Content
    Clear-Content
    Get-Content
    Set-Content

    DRIVE CMDLETS
    Get-PSDrive
    New-PSDrive
    Remove-PSDrive
```

```
    ITEM CMDLETS
    Clear-Item
    Copy-Item
    Get-Item
    Invoke-Item
    Move-Item
    New-Item
    Remove-Item
    Rename-Item
    Set-Item

    LOCATION CMDLETS
    Get-Location
    Pop-Location
    Push-Location
    Set-Location

    PATH CMDLETS
    Join-Path
    Convert-Path
    Split-Path
    Resolve-Path
    Test-Path

    PROPERTY CMDLETS
    Clear-ItemProperty
    Copy-ItemProperty
    Get-ItemProperty
    Move-ItemProperty
    New-ItemProperty
    Remove-ItemProperty
    Rename-ItemProperty
    Set-ItemProperty

    PROVIDER CMDLETS
    Get-PSProvider


PS C:\>
```

To view built-in PowerShell providers, use the following command:

```
PS C:\> get-psprovider

Name                 Capabilities                 Drives
----                 ------------                 ------
Alias                ShouldProcess                {Alias}
Environment          ShouldProcess                {Env}
```

```
FileSystem          Filter, ShouldProcess         {C, D, E, F...}
Function            ShouldProcess                 {Function}
Registry            ShouldProcess                 {HKLM, HKCU}
Variable            ShouldProcess                 {Variable}
Certificate         ShouldProcess                 {cert}


PS C:\>
```

The preceding list displays not only built-in providers, but also the drives each provider currently supports. A **drive** is an entity that a provider uses to represent a data store through which data is made available to the PowerShell session. For example, the Registry provider creates a PowerShell drive for the `HKEY_LOCAL_MACHINE` and `HKEY_CURRENT_USER` Registry hives.

To see a list of all current PowerShell drives, use the following command:

```
PS C:\> get-psdrive

Name       Provider      Root
----       --------      ----
Alias      Alias
C          FileSystem    C:\
cert       Certificate   \
D          FileSystem    D:\
E          FileSystem    E:\
Env        Environment
F          FileSystem    F:\
Function   Function
G          FileSystem    G:\
HKCU       Registry      HKEY_CURRENT_USER
HKLM       Registry      HKEY_LOCAL_MACHINE
U          FileSystem    U
Variable   Variable


PS C:\>
```

Accessing Drives and Data

One way to access PowerShell drives and their data is with the `Set-Location` cmdlet. This cmdlet, shown in the following example, changes the working location to another specified location that can be a directory, subdirectory, location stack, or Registry location:

```
PS C:\> set-location hklm:
PS HKLM:\> set-location software\microsoft\windows
PS HKLM:\software\microsoft\windows>
```

Next, use the `Get-ChildItem` cmdlet to list the subkeys under the `Windows` key:

```
PS HKLM:\software\microsoft\windows> get-childitem


   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software\micros
oft\windows

SKC  VC Name                           Property
---  -- ----                           --------
 55  13 CurrentVersion                 {DevicePath, MediaPathUnexpanded, SM_...
  0  16 Help                           {PINTLPAD.HLP, PINTLPAE.HLP, IMEPADEN...
  0  36 Html Help                      {PINTLGNE.CHM, PINTLGNT.CHM, PINTLPAD...
  1   0 ITStorage                      {}
  0   0 Shell                          {}


PS HKLM:\software\microsoft\windows>
```


Note that with a Registry drive, the `Get-ChildItem` cmdlet lists only the subkeys under a key, not the actual Registry values. This is because Registry values are treated as properties for a key rather than a valid item. To retrieve these values from the Registry, you use the `Get-ItemProperty` cmdlet, as shown in this example:

```
PS HKLM:\software\microsoft\windows> get-itemproperty currentversion


PSPath                      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHI
                              NE\software\microsoft\windows\currentversion
PSParentPath                : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHI
                              NE\software\microsoft\windows
PSChildName                 : currentversion
PSDrive                     : HKLM
PSProvider                  : Microsoft.PowerShell.Core\Registry
DevicePath                  : C:\WINDOWS\inf
MediaPathUnexpanded         : C:\WINDOWS\Media
SM_GamesName                : Games
SM_ConfigureProgramsName    : Set Program Access and Defaults
ProgramFilesDir             : C:\Program Files
CommonFilesDir              : C:\Program Files\Common Files
ProductId                   : 76487-OEM-0011903-00101
WallPaperDir                : C:\WINDOWS\Web\Wallpaper
MediaPath                   : C:\WINDOWS\Media
```

```
ProgramFilesPath          : C:\Program Files
SM_AccessoriesName        : Accessories
PF_AccessoriesName        : Accessories
(default)                 :


PS HKLM:\software\microsoft\windows>
```

As with the `Get-Process` command, the data returned is a collection of objects. You can modify these objects further to produce the output you want, as this example shows:

```
PS HKLM:\software\microsoft\windows> get-itemproperty currentversion |
select ProductId

ProductId
---------
76487-OEM-XXXXXXX-XXXXX


PS HKLM:\software\microsoft\windows>
```

Accessing data from a `FileSystem` drive is just as simple. The same type of command logic is used to change the location and display the structure:

```
PS HKLM:\software\microsoft\windows> set-location c:
PS C:\> set-location "C:\WINDOWS\system32\windowspowershell\v1.0"
PS C:\WINDOWS\system32\windowspowershell\v1.0> get-childitem about_a*


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\system32\window
    spowershell\v1.0


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-----          9/8/2006   2:10 AM       5662 about_alias.help.txt
-----          9/8/2006   2:10 AM       3504 about_arithmetic_operators.help.txt
-----          9/8/2006   2:10 AM       8071 about_array.help.txt
-----          9/8/2006   2:10 AM      15137 about_assignment_operators.help.txt
-----          9/8/2006   2:10 AM       5622 about_associative_array.help.txt
-----          9/8/2006   2:10 AM       3907 about_automatic_variables.help.txt
...


PS C:\WINDOWS\system32\windowspowershell\v1.0>
```

What's different is that data is stored in an item instead of being a property of that item. To retrieve data from an item, use the `Get-Content` cmdlet, as shown in this example:

```
PS C:\WINDOWS\system32\windowspowershell\v1.0> get-content
about_Alias.help.txt
TOPIC
    Aliases

SHORT DESCRIPTION
    Using pseudonyms to refer to cmdlet names in the Windows PowerShell

LONG DESCRIPTION
    An alias is a pseudonym, or "nickname," that you can assign to a
    cmdlet so that you can use the alias in place of the cmdlet name.
    The Windows PowerShell interprets the alias as though you had
    entered the actual cmdlet name. For example, suppose that you want
    to retrieve today's date for the year 1905. Without an alias, you
    would use the following command:

        Get-Date -year 1905
...


PS C:\WINDOWS\system32\windowspowershell\v1.0>
```

3

NOTE

Not all drives are based on a hierarchical data store. For example, the Environment, Function, and Variable PowerShell providers aren't hierarchical. Data accessed through these providers is in the root location on the associated drive.

Mounting a Drive

PowerShell drives can be created and removed, which is handy when you're working with a location or set of locations frequently. Instead of having to change the location or use an absolute path, you can create new drives (also referred to as "mounting a drive" in PowerShell) as shortcuts to those locations. To do this, use the `New-PSDrive` cmdlet, shown in the following example:

```
PS C:\> new-psdrive -name PSScripts -root D:\Dev\Scripts -psp FileSystem

Name       Provider      Root                                   CurrentLocation
----       --------      ----                                   ---------------
PSScripts  FileSystem    D:\Dev\Scripts


PS C:\> get-psdrive
```

```
Name       Provider      Root                                   CurrentLocation
----       --------      ----                                   ---------------
Alias      Alias
C          FileSystem    C:\
cert       Certificate   \
D          FileSystem    D:\
E          FileSystem    E:\
Env        Environment
F          FileSystem    F:\
Function   Function
G          FileSystem    G:\
HKCU       Registry      HKEY_CURRENT_USER                             software
HKLM       Registry      HKEY_LOCAL_MACHINE                  ...crosoft\windows
PSScripts  FileSystem    D:\Dev\Scripts
U          FileSystem    U:\
Variable   Variable


PS C:\>
```

To remove a drive, use the `Remove-PSDrive` cmdlet, as shown here:

```
PS C:\> remove-psdrive -name PSScripts
PS C:\> get-psdrive

Name       Provider      Root                       CurrentLocation
----       --------      ----                       ---------------
Alias      Alias
C          FileSystem    C:\
cert       Certificate   \
D          FileSystem    D:\
E          FileSystem    E:\
Env        Environment
F          FileSystem    F:\
Function   Function
G          FileSystem    G:\
HKCU       Registry      HKEY_CURRENT_USER          software
HKLM       Registry      HKEY_LOCAL_MACHINE         ...crosoft\windows
U          FileSystem    U:\
Variable   Variable


PS C:\>
```

Understanding Errors

PowerShell errors are divided into two types: terminating and nonterminating. **Terminating errors**, as the name implies, stop a command. **Nonterminating errors** are generally just reported without stopping a command. Both types of errors are reported in

the $Error variable, which is a collection of errors that have occurred during the current PowerShell session. This collection contains the most recent error, as indicated by $Error[0] up to $MaximumErrorCount, which defaults to 256.

Errors in the $Error variable can be represented by the ErrorRecord object. It contains error exception information as well as a number of other properties that are useful for understanding why an error occurred

The next example shows the information that is contained in InvocationInfo property of an ErrorRecord object:

```
PS C:\> $Error[0].InvocationInfo


MyCommand         : Get-ChildItem
ScriptLineNumber : 1
OffsetInLine      : -2147483648
ScriptName        :
Line              : dir z:
PositionMessage   :
                    At line:1 char:4
                    + dir  <<<< z:
InvocationName    : dir
PipelineLength    : 1
PipelinePosition : 1



PS C:\>
```

Based on this information, you can determine a number of details about $Error[0], including the command that caused the error to be thrown. This information is crucial to understanding errors and handling them effectively.

Use the following command to see a full list of ErrorRecord properties:

```
PS C:\> $Error[0] | get-member -MemberType Property


   TypeName: System.Management.Automation.ErrorRecord

Name                  MemberType Definition
----                  ---------- ----------
CategoryInfo          Property   System.Management.Automation.ErrorCategoryI...
ErrorDetails          Property   System.Management.Automation.ErrorDetails E...
Exception             Property   System.Exception Exception {get;}
FullyQualifiedErrorId Property   System.String FullyQualifiedErrorId {get;}
```

```
InvocationInfo        Property   System.Management.Automation.InvocationInfo...
TargetObject          Property   System.Object TargetObject {get;}


PS C:\>
```

Table 3.1 shows the definitions for each of the ErrorRecord properties that are listed in the preceding example:

TABLE 3.1 ErrorRecord Property Definitions

Property	Definition
`CategoryInfo`	Indicates under which category an error is classified
`ErrorDetails`	Can be null, but when used provides additional information about the error
`Exception`	The error that occurred
`FullyQualifiedErrorId`	Identifies an error condition more specifically
`InvocationInfo`	Can be null, but when used explains the context in which the error occurred
`TargetObject`	Can be null, but when used indicates the object being operated on

Error Handling

Methods for handling errors in PowerShell can range from simple to complex. The simple method is to allow PowerShell to handle the error. Depending on the type of error, the command or script might terminate or continue. However, if the default error handler doesn't fit your needs, you can devise a more complex error-handling scheme by using the methods discussed in the following sections.

Method One: cmdlet Preferences

In PowerShell, **ubiquitous parameters** are available to all cmdlets. Among them are the `ErrorAction` and `ErrorVariable` parameters, used to determine how cmdlets handle *nonterminating* errors, as shown in this example:

```
PS C:\> get-childitem z: -ErrorVariable Err -ErrorAction SilentlyContinue
PS C:\> if ($Err){write-host $Err -Foregroundcolor Red}
Cannot find drive. A drive with name 'z' does not exist.
PS C:\>
```

The `ErrorAction` parameter defines how a cmdlet behaves when it encounters a *nonterminating* error. In the preceding example, `ErrorAction` is defined as `SilentlyContinue`, meaning the cmdlet continues running with no output if it encounters a *nonterminating* error. Other options for `ErrorAction` are as follows:

- `Continue`—Print error and continue (default action)
- `Inquire`—Ask users whether they want to continue, halt, or suspend
- `Stop`—Halt execution of the command or script

NOTE

The term *nonterminating* has been emphasized in this section because a terminating error bypasses the defined `ErrorAction` and is delivered to the default or custom error handler.

The `ErrorVariable` parameter defines the variable name for the error object generated by a *nonterminating* error. As shown in the previous example, `ErrorVariable` is defined as `Err`. Notice the variable name doesn't have the `$` prefix. However, to access `ErrorVariable` outside a cmdlet, you use the variable's name with the `$` prefix (`$Err`). Furthermore, after defining `ErrorVariable`, the resulting variable is valid for the current PowerShell session or associated script block. This means other cmdlets can append error objects to an existing `ErrorVariable` by using a `+` prefix, as shown in this example:

```
PS C:\> get-childitem z: -ErrorVariable Err -ErrorAction SilentlyContinue
PS C:\> get-childitem y: -ErrorVariable +Err -ErrorAction SilentlyContinue
PS C:\> write-host $Err[0] -Foregroundcolor Red
Cannot find drive. A drive with name 'z' does not exist.
PS C:\> write-host $Err[1] -Foregroundcolor Red
Cannot find drive. A drive with name 'y' does not exist.
PS C:\>
```

Method Two: Trapping Errors

When encountering a terminating error, PowerShell's default behavior is to display the error and halt the command or script execution. If you want to use custom error handling for a terminating error, you must define an exception trap handler to prevent the terminating error (`ErrorRecord`) from being sent to the default error-handling mechanism. The same holds true for *nonterminating* errors as PowerShell's default behavior is to just display the error and continue the command or script execution.

To define a trap, you use the following syntax:

```
trap ExceptionType {code; keyword}
```

The first part is *`ExceptionType`*, which specifies the type of error a trap accepts. If no *`ExceptionType`* is defined, a trap accepts all errors. The *`code`* part can consist of a command or set of commands that run after an error is delivered to the trap. Defining

commands to run by a trap is optional. The last part, *keyword*, is what determines whether the trap allows the statement block where the error occurred to execute or terminate.

Supported keywords are as follows:

- `Break`—Causes the exception to be rethrown and stops the current scope from executing
- `Continue`—Allows the current scope execution to continue at the next line where the exception occurred
- `Return [argument]`—Stops the current scope from executing and returns the argument, if specified

If a keyword isn't specified, the trap uses the keyword `Return [argument]`; *argument* is the `ErrorRecord` that was originally delivered to the trap.

Trap Examples

The following two examples show how traps can be defined to handle errors. The first trap example shows a trap being used in conjunction with a *nonterminating* error that is produced from an invalid DNS name being given to the `System.Net.Dns` class. The second example shows a trap being again used in conjunction with a *nonterminating* error that is produced from the `Get-Item` cmdlet. However, in this case, because the `ErrorAction` parameter has been defined as `Stop`, the error is in fact a terminating error that is then handled by the trap.

Example one: `errortraps1.ps1`

```
$DNSName = "www.-baddnsname-.com"

trap [System.Management.Automation.MethodInvocationException]{
    write-host ("ERROR: " + $_) -Foregroundcolor Red; Continue}

write-host "Getting IP address for" $DNSName
write-host ([System.Net.Dns]::GetHostAddresses("www.$baddnsname$.com"))
write-host "Done Getting IP Address"
```

The `$_ parameter` in this example represents the ErrorRecord that was delivered to the trap.

Output:

```
PS C:\> .\errortraps1.ps1
Getting IP address for www.-baddnsname-.com
ERROR: Exception calling "GetHostAddresses" with "1" argument(s): "No such host
is known"
Done Getting IP Address
PS C:\>
```

Example two: `errortraps2.ps1`

```
write-host "Changing drive to z:"

trap {write-host("[ERROR] " + $_) -Foregroundcolor Red; Continue}

get-item z: -ErrorAction Stop
$TXTFiles = get-childitem *.txt -ErrorAction Stop

write-host "Done getting items"
```


NOTE

A cmdlet doesn't generate a terminating error unless there's a syntax error. This means a trap doesn't catch nonterminating errors from a cmdlet unless the error is transformed into a terminating error by setting the cmdlet's `ErrorAction` to `Stop`.

Output:

```
PS C:\> .\errortraps2.ps1
Changing drive to z:
[ERROR] Command execution stopped because the shell variable
"ErrorActionPreference" is set to Stop: Cannot find drive. A drive
with name 'z' does not exist.
Done getting items
PS C:\>
```

Trap Scopes

A PowerShell scope, as discussed in Chapter 2, "PowerShell Basics," determines how traps are executed. Generally, a trap is defined and executed within the same scope. For example, you define a trap in a certain scope; when a terminating error is encountered in that scope, the trap is executed. If the current scope doesn't contain a trap and an outer scope does, any terminating errors encountered break out of the current scope and are delivered to the trap in the outer scope.

Method Three: The `Throw` Keyword

In PowerShell, you can generate your own terminating errors. This doesn't mean causing errors by using incorrect syntax. Instead, you can generate a terminating error on purpose by using the `throw` keyword, as shown in the next example if a user doesn't define the argument for the MyParam parameter when trying to run the `MyParam.ps1` script. This type of behavior is very useful when data from functions, cmdlets, data sources, applications, etc. is not what is expected and hence may prevent the script or set of commands from executing correctly further into the execution process.

Script:

```
param([string]$MyParam = $(throw write-host "You did not define MyParam"
-Foregroundcolor Red))

write-host $MyParam
```

Output:

```
PS C:\ .\MyParam.ps1
You did not define MyParam
ScriptHalted
At C:\MyParam.ps1:1 char:33
+ param([string]$MyParam = $(throw  <<<< write-host "You did not define MyParam
" -Foregroundcolor Red))
PS C:\>
```

PowerShell Profiles

A PowerShell **profile** is a saved collection of settings for customizing the PowerShell environment. There are four types of profiles, loaded in a specific order each time PowerShell starts. The following sections explain these profile types, where they should be located, and the order in which they are loaded.

The All Users Profile

This profile is located in `%windir%\system32\windowspowershell\v1.0\profile.ps1`. Settings in the All Users profile are applied to all PowerShell users on the current machine. If you plan to configure PowerShell settings across the board for users on a machine, then this would be the profile to use.

The All Users Host-Specific Profile

This profile is located in `%windir%\system32\windowspowershell\v1.0\`*`ShellID`*`_profile.ps1`. Settings in the All Users host-specific profile are applied to all users of the current shell (by default, the PowerShell console). PowerShell supports the concept of multiple shells or hosts. For example, the PowerShell console is a host and the one most users use exclusively. However, other applications can call an instance of the PowerShell runtime to access and run PowerShell commands and scripts. An application that does this is called a **hosting application** and uses a host-specific profile to control the PowerShell configuration. The host-specific profile name is reflected by the host's `ShellID`. In the PowerShell console, the `ShellID` is the following:

```
PS C:\ $ShellId
Microsoft.PowerShell
PS C:\
```

Putting this together, the PowerShell console's All Users host-specific profile is named `Microsoft.PowerShell_profile.ps1`. For other hosts, the `ShellID` and All Users host-specific profile names are different. For example, the PowerShell Analyzer (www.powershellanalyzer.com) is a PowerShell host that acts as a rich graphical interface for the PowerShell environment. Its `ShellID` is `PowerShellAnalyzer.PSA`, and its All Users host-specific profile name is `PowerShellAnalyzer.PSA_profile.ps1`.

The Current User's Profile

This profile is located in `%userprofile%\My Documents\WindowsPowerShell\profile.ps1`. Users who want to control their own profile settings can use the current user's profile. Settings in this profile are applied only to the user's current PowerShell session and doesn't affect any other users.

The Current User's Host-Specific Profile

This profile is located in `%userprofile%\My Documents\WindowsPowerShell\`*`ShellID`*`_profile.ps1`. Like the All Users host-specific profile, this profile type loads settings for the current shell. However, the settings are user specific.

NOTE

When you start the shell for the first time, you might see a message indicating that scripts are disabled and no profiles are loaded. You can modify this behavior by changing the PowerShell execution policy, discussed in the following section.

Understanding Security

When WSH was released with Windows 98, it was a godsend for Windows administrators who wanted the same automation capabilities as their UNIX brethren. At the same time, virus writers quickly discovered that WSH also opened up a large attack vector against Windows systems.

Almost anything on a Windows system can be automated and controlled by using WSH, which is an advantage for administrators. However, WSH doesn't provide any security in script execution. If given a script, WSH runs it. Where the script comes from or its purpose doesn't matter. With this behavior, WSH became known more as a security vulnerability than an automation tool.

Execution Policies

Because of past criticisms of WSH's security, when the PowerShell team set out to build a Microsoft shell, the team decided to include an execution policy to mitigate the security threats posed by malicious code. An **execution policy** defines restrictions on how PowerShell allows scripts to run or what configuration files can be loaded. PowerShell has four execution policies, discussed in more detail in the following sections: `Restricted`, `AllSigned`, `RemoteSigned`, and `Unrestricted`.

`Restricted`

By default, PowerShell is configured to run under the `Restricted` execution policy. This execution policy is the most secure because it allows PowerShell to operate only in an interactive mode. This means no scripts can be run, and only configuration files digitally signed by a trusted publisher are allowed to run or load.

`AllSigned`

The `AllSigned` execution policy is a notch under `Restricted`. When this policy is enabled, only scripts or configuration files that are digitally signed by a publisher you trust can be run or loaded. Here's an example of what you might see if the `AllSigned` policy has been enabled:

```
PS C:\Scripts> .\evilscript.ps1
The file C:\Scripts\evilscript.ps1 cannot be loaded. The file
C:\Scripts\evilscript.ps1 is not digitally signed. The script will not
execute on the system. Please see "get-help about_signing" for more
details.
At line:1 char:16
+ .\evilscript.ps1 <<<<
PS C:\Scripts>
```

Signing a script or configuration file requires a code-signing certificate. This certificate can come from a trusted certificate authority (CA), or you can generate one with the Certificate Creation Tool (`Makecert.exe`). Usually, however, you want a valid code-signing certificate from a well-known trusted CA, such as Verisign, Thawte, or your corporation's internal public key infrastructure (PKI). Otherwise, sharing your scripts or configuration files with others might be difficult because your computer isn't a trusted CA by default.

NOTE

Chapter 4, "Code Signing," explains how to obtain a valid trusted code-signing certificate. Reading this chapter is strongly recommended because of the importance of digitally signing scripts and configuration files.

`RemoteSigned`

The `RemoteSigned` execution policy is designed to prevent remote PowerShell scripts and configuration files that aren't digitally signed by a trusted publisher from running or loading automatically. Scripts and configuration files that are locally created can be loaded and run without being digitally signed, however.

A remote script or configuration file can be obtained from a communication application, such as Microsoft Outlook, Internet Explorer, Outlook Express, or Windows Messenger. Running or loading a file downloaded from any of these applications results in the following error message:

```
PS C:\Scripts> .\interscript.ps1
The file C:\Scripts\interscript.ps1 cannot be loaded. The file
C:\Scripts\interscript.ps1 is not digitally signed. The script will
not execute on the system. Please see "get-help about_signing" for
more details..
At line:1 char:17
+ .\interscript.ps1 <<<<
PS C:\Scripts>
```

To run or load an unsigned remote script or configuration file, you must specify whether to trust the file. To do this, right-click the file in Windows Explorer and click Properties. In the General tab, click the Unblock button (see Figure 3.1).

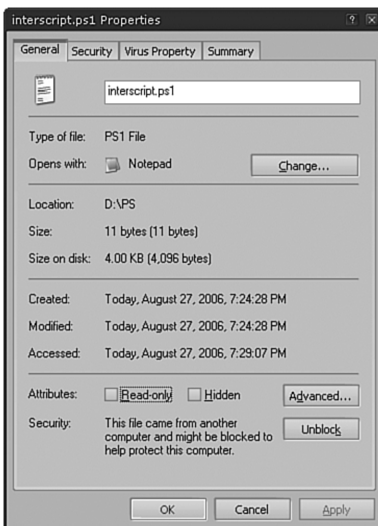

FIGURE 3.1 Trusting a remote script or configuration file

After you trust the file, the script or configuration file can be run or loaded. If it's digitally signed but the publisher isn't trusted, PowerShell displays the following prompt:

```
PS C:\Scripts> .\signed.ps1

Do you want to run software from this untrusted publisher?
File C:\Scripts\signed.ps1 is published by CN=companyabc.com, OU=IT,
O=companyabc.com, L=Oakland, S=California, C=US and is not trusted on
your system. Only run scripts from trusted publishers.
[V] Never run  [D] Do not run  [R] Run once  [A] Always run  [?] Help
(default is "D"):
```

In this case, you must choose whether to trust the file content.

NOTE

Chapter 4 explains the options in this prompt in more detail.

`Unrestricted`

As the name suggests, the `Unrestricted` execution policy removes almost all restrictions for running scripts or loading configuration files. All local or signed trusted files can run or load, but for remote files, PowerShell prompts you to choose an option for running or loading that file, as shown here:

```
PS C:\Scripts> .\remotescript.ps1

Security Warning
Run only scripts that you trust. While scripts from the Internet can
be useful, this script can potentially harm your computer. Do you want
to run
C:\Scripts\remotescript.ps1?
[D] Do not run  [R] Run once  [S] Suspend  [?] Help (default is "D"):
```

Setting the Execution Policy

To change the execution policy, you use the `Set-ExecutionPolicy` cmdlet, shown here:

```
PS C:\> set-executionpolicy AllSigned
PS C:\>
```

If you want to know the current execution policy, use the `Get-ExecutionPolicy` cmdlet:

```
PS C:\> get-executionpolicy
AllSigned
PS C:\>
```

By default, when PowerShell is first installed, the execution policy is set to `Restricted`. As you know, default settings never stay default for long. In addition, if PowerShell is installed on many machines, the likelihood of its execution policy being set to `Unrestricted` increases.

3

Fortunately, you can control the PowerShell execution policy through a Registry setting. This setting is a `REG_SZ` value named `ExecutionPolicy`, which is located in the `HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell` key. Controlling the execution policy through the Registry means you can enforce a policy setting across many machines managed by a Group Policy Object (GPO).

In the past, creating a GPO to control the execution policy was simple because the PowerShell installation includes a Group Policy Administrative Template (ADM). However, as of the PowerShell RC2 release, the ADM is no longer part of the installation and may or may not be available in a separate PowerShell download. If Microsoft doesn't provide an ADM to control the execution policy, you can always create your own, as shown in the following example:

```
CLASS MACHINE

CATEGORY !!PowerShell
    POLICY !!Security
        KEYNAME "SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell"

        EXPLAIN !!PowerShell_ExecutionPolicy

        PART !!ExecutionPolicy EDITTEXT REQUIRED
            VALUENAME "ExecutionPolicy"
        END PART
    END POLICY
END CATEGORY

[strings]
PowerShell=PowerShell
Security=Security Settings
PowerShell_ExecutionPolicy=If enabled, this policy will set the PowerShell
execution policy on a machine to the defined value.  Execution policy values can be
Restricted, AllSigned, RemoteSigned, and Unrestricted.
Executionpolicy=Execution Policy
```

You can find a working version of this ADM on the PowerShell Unleashed Reference Web site: www.samspublishing.com. Although the `PowerShellExecutionPolicy.adm` file has been tested and should work in your environment, note that the execution policy settings in this file are considered preference settings. Preference settings are GPOs that are Registry values found outside the approved Group Policy Registry trees. When a GPO containing preference settings goes out of scope, the preference settings aren't removed from the Registry.

NOTE

As with everything provided on the PowerShell Unleashed Reference Web site, test the ADM in a non-production environment before deploying a GPO that uses it.

To configure the `PowerShellExecutionPolicy.adm` file, follow these steps:

1. Log on to a GPO management machine as the GPO administrator.
2. Using the Group Policy MMC, create a GPO named **PowerShell**.
3. In the console tree, click to expand **Computer Configuration** and then **Administrative Templates**.
4. Right-click **Administrative Templates** and click **Add/Remove Templates** in the shortcut menu.
5. Navigate to the folder with the `PowerShellExecutionPolicy.adm` file. Select the file, click **Open**, and then click **Close**. The PowerShell node is then displayed under the Administrative Templates node.
6. Click the **Administrative Templates** node, and then click **View**, **Filtering** from the Group Policy MMC menu. Click to clear the **Only show policy settings that can be fully managed** checkbox. Clearing this option allows you to manage preference settings.
7. Next, click the **PowerShell** node under Administrative Templates.
8. In the details pane, right-click **Security Settings** and click **Properties** in the shortcut menu.
9. Click **Enabled**.
10. Set the **Execution Policy** to one of these values: Restricted, AllSigned, RemoteSigned, or Unrestricted.
11. Close the GPO, and then close the Group Policy MMC.

Controlling the execution policy through a GPO preference setting might seem like a less then perfect solution. After all, a preference setting doesn't offer the same level of security as an execution policy setting, so users with the necessary rights can modify it easily. This lack of security is probably why Microsoft removed the original ADM file from

PowerShell. A future release of PowerShell might allow controlling the execution policy with a valid GPO policy setting.

Additional Security Measures

Execution policies aren't the only security layer Microsoft implemented in PowerShell. PowerShell script files with the `.ps1` extension can't be run from Windows Explorer because they are associated with Notepad. In other words, you can't just double-click a `.ps1` file to run it. Instead, PowerShell scripts must run from a PowerShell session by using the relative or absolute path or through the cmd command prompt by using the PowerShell executable.

Another security measure, explained in Chapter 2, is that to run or open a file in the current directory from the PowerShell console, you must prefix the command with `.\` or `./`. This feature prevents PowerShell users from accidentally running a command or PowerShell script without specifying its execution explicitly.

Last, by default, there's no method for connecting to or calling PowerShell remotely. However, that doesn't mean you can't write an application that allows remote PowerShell connections. In fact, it has been done. If you're interested in learning how, download the PowerShell Remoting beta from www.gotdotnet.com/workspaces/workspace.aspx?id=ce09cdaf-7da2-4f1c-bed3-f8cb35de5aea.

The PowerShell Language

From this point on, this book varies from the usual format of many books on scripting languages, which try to explain scripting concepts instead of showing you actual working scripts. This book focuses on the practical applications of PowerShell.

It's assumed you have a basic understanding of scripting. In addition, because the PowerShell scripting language is similar to Perl, C#, and even VBScript, there's no need to spend time reviewing `for` loops, `if...then` statements, and other fundamentals of scripting.

Granted, there are some unique aspects to the PowerShell language, but you can consult the PowerShell documentation for that information. This is not a language reference book; it's about how PowerShell can be applied in the real world. For more detailed information about the PowerShell language, you can download the PowerShell User Guide from www.microsoft.com/downloads/details.aspx?FamilyId=B4720B00-9A66-430F-BD56-EC48BFCA154F&displaylang=en.

Summary

In this chapter, you have delved deeper into what PowerShell is and how it works. You reviewed such topics as Powershell's Providers, how it handles errors, its profiles, and its execution policies. However, of the items reviewed the most important concept to take from this chapter is that PowerShell is built from and around the .NET Framework. As

such, PowerShell is not like other shells because it is an object-based shell that attempts to abstract all objects into a common form that can be used without modification (parsing) in your commands and scripts. Going forward this and the knowledge that you have learned from Chapters 2 and 3 will be the keystone from which you shall explore PowerShell scripting. Moving through each chapter, the scripts will increase in complexity as we review different aspects of how PowerShell can be used for Windows automation.