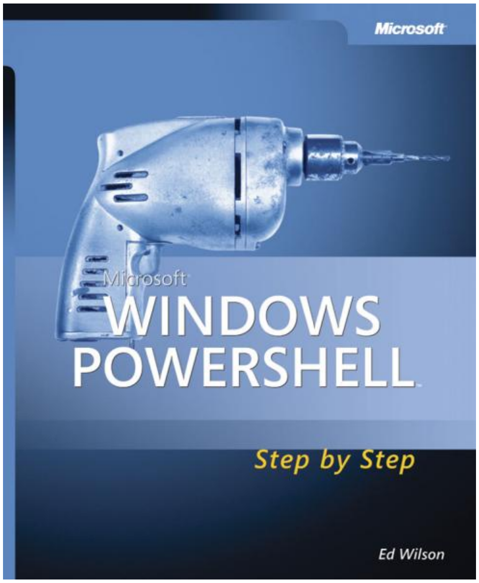


Microsoft® Windows PowerShell™ Step By Step

Ed Wilson

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/10329.aspx

9780735623958
Publication Date: May 2007

Chapter 3

Leveraging PowerShell Providers

After completing this chapter, you will be able to:

- Understand the role of providers in Windows PowerShell
- Use the *Get-PSProvider* cmdlet
- Use the *Get-PSDrive* cmdlet
- Use the *Get-Item* cmdlet
- Use the *Set-Location* cmdlet
- Use the file system model to access data from each of the built-in providers

Windows PowerShell provides a consistent way to access information external to the shell environment. To do this, it uses providers. These providers are actually .NET programs that hide all the ugly details to provide an easy way to access information. The beautiful thing about the way the provider model works is that all the different sources of information are accessed in exactly the same manner. This chapter demonstrates how to leverage the PowerShell providers. All the scripts mentioned in this chapter can be found in the corresponding scripts folder on the CD.

Identifying the Providers

By identifying the providers installed with Windows PowerShell, we can begin to understand the capabilities intrinsic to a default installation. Providers expose information contained in different data stores by using a drive and file system analogy. An example of this is obtaining a listing of registry keys—to do this, you would connect to the registry “drive” and use the *Get-ChildItem* cmdlet, which is exactly the same method you would use to obtain a listing of files on the hard drive. The only difference is the specific name associated with each drive. Providers can be created by anyone familiar with Windows .NET programming. When a new provider is created, it is called a *snap-in*. A snap-in is a dynamic link library (*dll*) file that must be installed into Windows PowerShell. After a snap-in has been installed, it cannot be un-installed—however, the snap-in can be removed from the current Windows PowerShell console.

Just the Steps To obtain a listing of all the providers, use the *Get-PSProvider* cmdlet. Example: get-psprovider. This command produces the following list on a default installation of the Windows PowerShell:

```
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}
```

Understanding the Alias Provider

In Chapter 1, Overview of Windows PowerShell, we presented the various *Help* utilities available that show how to use cmdlets. The alias provider provides easy-to-use access to all aliases defined in Windows PowerShell. To work with the aliases on your machine, use the *Set-Location* cmdlet and specify the Alias:\ drive. You can then use the same cmdlets you would use to work with the file system.

Tip With the alias provider, you can use a *Where-Object* cmdlet and filter to search for an alias by name or description.

Working with the alias provider

1. Open Windows PowerShell.
2. Obtain a listing of all the providers by using the *Get-PSProvider* cmdlet. This is shown here:

   ```
   Get-PSProvider
   ```

3. The PSDrive associated with the alias provider is called Alias. This is seen in the listing produced by the *Get-PSProvider* cmdlet. Use the *Set-Location* cmdlet to change to the Alias drive. Use the *sl* alias to reduce typing. This command is shown here:

   ```
   sl alias:\
   ```

4. Use the *Get-ChildItem* cmdlet to produce a listing of all the aliases that are defined on the system. To reduce typing, use the alias *gci* in place of *Get-ChildItem*. This is shown here:

   ```
   GCI
   ```

5. Use a *Where-Object* cmdlet filter to reduce the amount of information that is returned by using the *Get-ChildItem* cmdlet. Produce a listing of all the aliases that begin with the letter s. This is shown here:

   ```
   GCI | Where-Object {$_.name -like "s*"}
   ```

6. To identify other properties that could be used in the filter, pipeline the results of the *Get-ChildItem* cmdlet into the *Get-Member* cmdlet. This is shown here:

```
Get-ChildItem |Get-Member
```

7. Press the up arrow twice, and edit the previous filter to include only definitions that contain the word set. The modified filter is shown here:

```
GCI | Where-Object {$_.definition -like "set*"}
```

8. The results of this command are shown here:

```
CommandType     Name                            Definition
-----------     ----                            ----------
Alias           sal                             Set-Alias
Alias           sc                              Set-Content
Alias           si                              Set-Item
Alias           sl                              Set-Location
Alias           sp                              Set-ItemProperty
Alias           sv                              Set-Variable
Alias           cd                              Set-Location
Alias           chdir                           Set-Location
Alias           set                             Set-Variable
```

9. Press the up arrow three times, and edit the previous filter to include only names of aliases that are like the letter w. This revised command is seen here:

```
GCI | Where-Object {$_.name -like "*w*"}
```

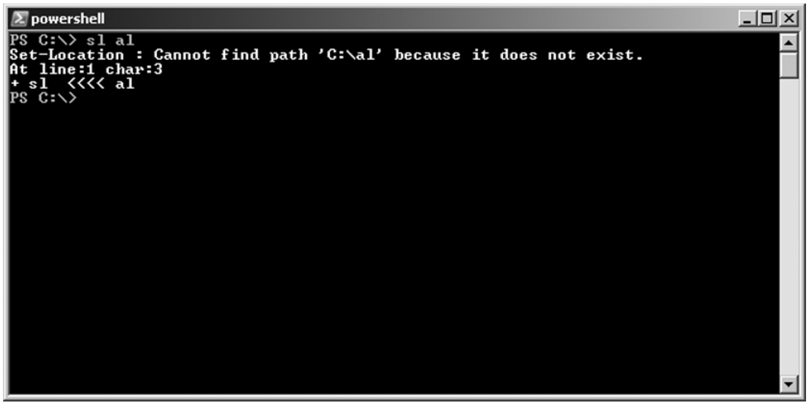
10. The results from this command are similar to those shown here:

```
CommandType     Name                            Definition
-----------     ----                            ----------
Alias           fw                              Format-Wide
Alias           gwmi                            Get-WmiObject
Alias           where                           Where-Object
Alias           write                           Write-Output
Alias           pwd                             Get-Location
```

11. From the list above, note that *where* is an alias for the *Where-Object* cmdlet. Press the up arrow one time to retrieve the previous command. Edit it to use the *where* alias instead of spelling out the entire *Where-Object* cmdlet name. This revised command is seen here:

```
GCI | where {$_.name -like "*w*"}
```

Caution When using the *Set-Location* cmdlet to switch to a newly created PSDrive, you must follow the name of the PSDrive with a colon. A trailing forward slash or backward slash is optional. An error will be generated if the colon is left out, as shown in Figure 3-1. I prefer to use the backward slash (\) because it is consistent with normal Windows file system operations.



```
PS C:\> sl al
Set-Location : Cannot find path 'C:\al' because it does not exist.
At line:1 char:3
+ sl  <<<< al
PS C:\>
```

Figure 3-1 Using Set-Location without : results in an error

Understanding the Certificate Provider

In the preceding section, we explored working with the alias provider. Because the file system model applies to the certificate provider in much the same way as it did the alias provider, many of the same cmdlets can be used. To find information about the certificate provider, use the *Get-Help* cmdlet. If you are unsure what articles in *Help* may be related to certificates, you can use the wild card asterisk (*) parameter. This command is shown here:

```
get-help *cer*
```

The certificate provider gives you the ability to sign scripts and allows Windows PowerShell to work with signed and unsigned scripts as well. It also gives you the ability search for, copy, move, and delete certificates. Using the certificate provider, you can even open the Certificates Microsoft Management Console (MMC). The commands used in the procedure are in the ObtainingAListingOfCertificates.txt file.

Obtaining a listing of certificates

1. Open Windows PowerShell.
2. Set your location to the cert PSDrive. To do this, use the *Set-Location* cmdlet, as shown here:

   ```
   Set-Location cert:\
   ```

3. Use the *Get-ChildItem* cmdlet to produce a list of the certificates, as shown here:

   ```
   Get-ChildItem
   ```

4. The list produced is shown here:

   ```
   Location   : CurrentUser
   StoreNames : {?, UserDS, AuthRoot, CA...}

   Location   : LocalMachine
   StoreNames : {?, AuthRoot, CA, AddressBook...}
   ```

5. Use the -recurse argument to cause the *Get-ChildItem* cmdlet to produce a list of all the certificate stores. To do this, press the up arrow key one time, and add the -recurse argument to the previous command. This is shown here:

   ```
   Get-ChildItem -recurse
   ```

6. Use the -path argument for *Get-ChildItem* to produce a listing of certificates in another store, without having to use the *Set-Location* cmdlet to change your current location. Using the *gci* alias, the command is shown here:

   ```
   GCI -path currentUser
   ```

7. Your listing of certificate stores will look similar to the one shown here:

   ```
   Name : ?

   Name : UserDS

   Name : AuthRoot

   Name : CA

   Name : AddressBook

   Name : ?

   Name : Trust

   Name : Disallowed

   Name : _NMSTR

   Name : ?????k

   Name : My

   Name : Root

   Name : TrustedPeople

   Name : ACRS

   Name : TrustedPublisher

   Name : REQUEST
   ```

8. Change your working location to the currentuser\authroot certificate store. To do this, use the *sl* alias followed by the path to the certificate store. This command is shown here:

   ```
   sl currentuser\authroot
   ```

9. Use the *Get-ChildItem* cmdlet to produce a listing of certificates in the currentuser\authroot certificate store that contain the name C&W in the subject field. Use the *gci* alias to reduce the amount of typing. Pipeline the resulting object to a *Where-Object* cmdlet, but use the *where* alias instead of typing *Where-Object*. The code to do this is shown here:

```
GCI | where {$_.subject -like "*c&w*"}
```

10. On my machine, there are four certificates listed. These are shown here:

```
Thumbprint                                Subject
----------                                -------
F88015D3F98479E1DA553D24FD42BA3F43886AEF  O=C&W HKT SecureNet CA SGC Root, C=hk
9BACF3B664EAC5A17BED08437C72E4ACDA12F7E7  O=C&W HKT SecureNet CA Class A, C=hk
4BA7B9DDD68788E12FF852E1A024204BF286A8F6  O=C&W HKT SecureNet CA Root, C=hk
47AFB915CDA26D82467B97FA42914468726138DD  O=C&W HKT SecureNet CA Class B, C=hk
```

11. Use the up arrow, and edit the previous command so that it will return only certificates that contain the phrase *SGC Root* in the subject property. The revised command is shown here:

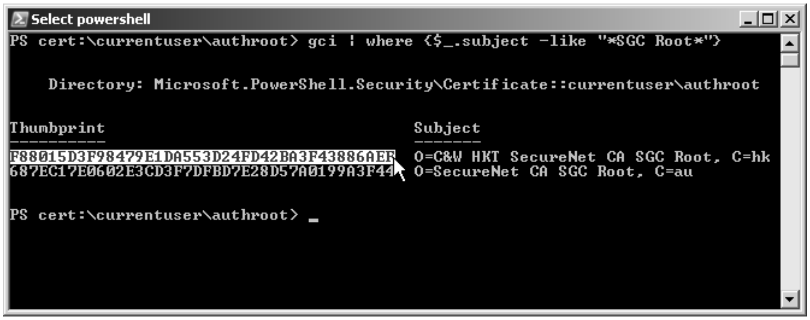
```
GCI | where {$_.subject -like "*SGC Root*"}
```

12. The resulting output on my machine contains an additional certificate. This is shown here:

```
Thumbprint                                Subject
----------                                -------
F88015D3F98479E1DA553D24FD42BA3F43886AEF  O=C&W HKT SecureNet CA SGC Root, C=hk
687EC17E0602E3CD3F7DFBD7E28D57A0199A3F44  O=SecureNet CA SGC Root, C=au
```

13. Use the up arrow, and edit the previous command. This time, change the *Where-Object* cmdlet so that it filters on the thumbprint attribute that is equal to F88015D3F98479E1DA553D24FD42BA3F43886AEF. You do not have to type that, however; to copy the thumbprint, you can highlight it and press Enter in Windows PowerShell, as shown in Figure 3-2. The revised command is shown here:

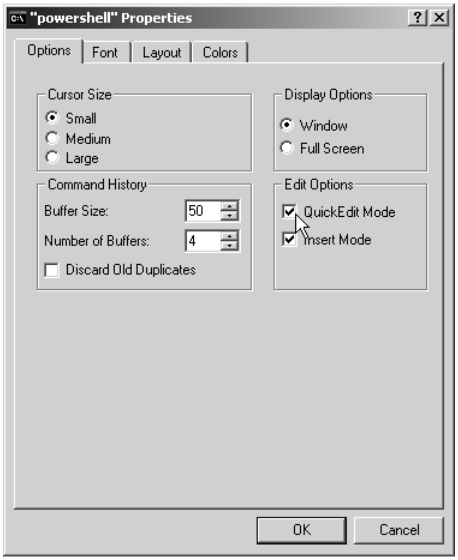
```
GCI | where {$_.thumbprint -eq "F88015D3F98479E1DA553D24FD42BA3F43886AEF"}
```



```
PS cert:\currentuser\authroot> gci | where {$_.subject -like "*SGC Root*"}

    Directory: Microsoft.PowerShell.Security\Certificate::currentuser\authroot

Thumbprint                                Subject
----------                                -------
F88015D3F98479E1DA553D24FD42BA3F43886AEF  O=C&W HKT SecureNet CA SGC Root, C=hk
687EC17E0602E3CD3F7DFBD7E28D57A0199A3F44  O=SecureNet CA SGC Root, C=au

PS cert:\currentuser\authroot> _
```

Figure 3-2 Highlight items to copy using the mouse

Troubleshooting If copying from inside a Windows PowerShell window does not work, then you probably need to enable Quick Edit Mode. To do this, right-click the PowerShell icon in the upper left-hand corner of the Windows PowerShell window. Choose Properties, and select Quick Edit Mode. This is shown in Figure 3-3.



Cursor Size
- ● Small
- ○ Medium
- ○ Large

Display Options
- ● Window
- ○ Full Screen

Command History
- Buffer Size: 50
- Number of Buffers: 4
- ☐ Discard Old Duplicates

Edit Options
- ☑ QuickEdit Mode
- ☑ Insert Mode

Figure 3-3 Enable Quick Edit Mode to enable Clipboard Support

14. To see all the properties of the certificate, pipeline the certificate object to a *Format-List* cmdlet and choose all the properties. The revised command is shown here:

```
GCI | where {$_.thumbprint -eq "F88015D3F98479E1DA553D24FD42BA3F43886AEF"} |
Format-List *
```

15. The output contains all the properties of the certificate object and is shown here:

```
PSPath             : Microsoft.PowerShell.Security\Certificate::currentuser\aut
                     hroot\F88015D3F98479E1DA553D24FD42BA3F43886AEF
PSParentPath       : Microsoft.PowerShell.Security\Certificate::currentuser\aut
                     hroot
PSChildName        : F88015D3F98479E1DA553D24FD42BA3F43886AEF
PSDrive            : cert
PSProvider         : Microsoft.PowerShell.Security\Certificate
PSIsContainer      : False
Archived           : False
Extensions         : {}
FriendlyName       : CW HKT SecureNet CA SGC Root
IssuerName         : System.Security.Cryptography.X509Certificates.X500Distingu
                     ishedName
NotAfter           : 10/16/2009 5:59:00 AM
NotBefore          : 6/30/1999 6:00:00 AM
HasPrivateKey      : False
PrivateKey         :
```

```
PublicKey          : System.Security.Cryptography.X509Certificates.PublicKey
RawData            : {48, 130, 2, 235...}
SerialNumber       : 00
SubjectName        : System.Security.Cryptography.X509Certificates.X500Distingu
                     ishedName
SignatureAlgorithm : System.Security.Cryptography.Oid
Thumbprint         : F88015D3F98479E1DA553D24FD42BA3F43886AEF
Version            : 1
Handle             : 75655840
Issuer             : O=C&W HKT SecureNet CA SGC Root, C=hk
Subject            : O=C&W HKT SecureNet CA SGC Root, C=hk
```

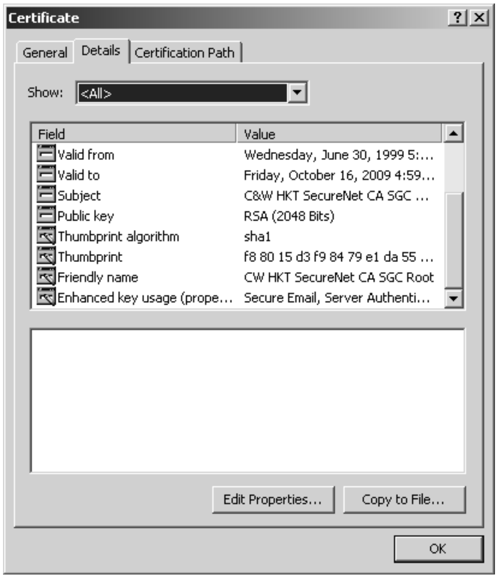
16. Open the Certificates MMC. This MMC is called Certmgr.msc and can be launched by simply typing the name inside Windows PowerShell, as shown here:

    ```
    Certmgr.msc
    ```

17. But it is more fun to use the *Invoke-Item* cmdlet to launch the Certificates MMC. To do this, supply the PSDrive name of cert:\ to the *Invoke-Item* cmdlet. This is shown here:

    ```
    Invoke-Item cert:\
    ```

18. Compare the information obtained from Windows PowerShell with the information displayed in the Certificates MMC. They are the same. The certificate is shown in Figure 3-4.



Show: <All>

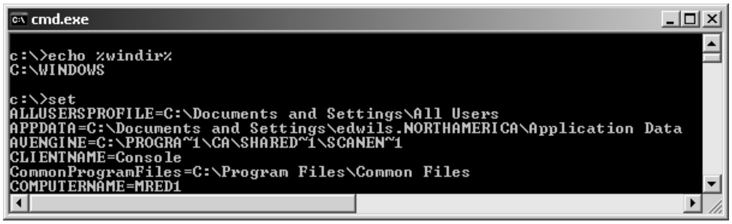
Field	Value
Valid from	Wednesday, June 30, 1999 5:...
Valid to	Friday, October 16, 2009 4:59...
Subject	C&W HKT SecureNet CA SGC ...
Public key	RSA (2048 Bits)
Thumbprint algorithm	sha1
Thumbprint	f8 80 15 d3 f9 84 79 e1 da 55 ...
Friendly name	CW HKT SecureNet CA SGC Root
Enhanced key usage (prope...	Secure Email, Server Authenti...

Figure 3-4 Certmgr.msc can be used to examine certificate properties

19. This concludes this procedure.

Understanding the Environment Provider

The environment provider in Windows PowerShell is used to provide access to the system environment variables. If you open a CMD (command) shell and type **set**, you will obtain a listing of all the environment variables defined on the system. If you use the echo command in the CMD shell to print out the value of %windir%, you will obtain the results seen in Figure 3-5.



```
c:\>echo %windir%
C:\WINDOWS

c:\>set
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\edwils.NORTHAMERICA\Application Data
AVENGINE=C:\PROGRA~1\CA\SHARED~1\SCANEN~1
CLIENTNAME=Console
CommonProgramFiles=C:\Program Files\Common Files
COMPUTERNAME=MRED1
```

Figure 3-5 Use *set* in a CMD prompt to see environment variables

Environment variables are used by various applications and other utilities as a shortcut to provide easy access to specific files, folders, and configuration data. By using the environment provider in Windows PowerShell, you can obtain a listing of the environment variables. You can also add, change, clear, and delete these variables.

Obtaining a listing of environment variables

1. Open Windows PowerShell.
2. Obtain a listing of the PSDrives by using the *Get-PSDrive* cmdlet. This is shown here:

   ```
   Get-PSDrive
   ```

3. Note that the Environment PSDrive is called *env*. Use the *env* name with the *Set-Locati*on cmdlet and change to the environment PSDrive. This is shown here:

   ```
   Set-Location env:\
   ```

4. Use the *Get-Item* cmdlet to obtain a listing of all the environment variables on the system. This is shown here:

   ```
   Get-Item *
   ```

5. Use the *Sort-Object* cmdlet to produce an alphabetical listing of all the environment variables by name. Use the up arrow to retrieve the previous command, and pipeline the returned object into the *Sort-Object* cmdlet. Use the property argument, and supply name as the value. This command is shown here:

   ```
   get-item * | Sort-Object  -property name
   ```

6. Use the *Get-Item* cmdlet to retrieve the value associated with the environment variable *windir*. This is shown here:

   ```
   get-item windir
   ```

7. Use the up arrow and retrieve the previous command. Pipeline the object returned to the *Format-List* cmdlet and use the wild card character to print out all the properties of the object. The modified command is shown here:

   ```
   get-item windir | Format-List *
   ```

8. The properties and their associated values are shown here:

   ```
   PSPath        : Microsoft.PowerShell.Core\Environment::windir
   PSDrive       : Env
   PSProvider    : Microsoft.PowerShell.Core\Environment
   PSIsContainer : False
   Name          : windir
   Key           : windir
   Value         : C:\WINDOWS
   ```

9. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Creating a new environment variable

1. You should still be in the Environment PSDrive from the previous procedure. If not, use the *Set-Location env:* command).
2. Use the *Get-Item* cmdlet to produce a listing of all the environment variables. Pipeline the returned object to the *Sort-Object* cmdlet using the property of name. To reduce typing, use the *gi* alias and the *sort* alias. This is shown here:

   ```
   GI * | Sort -Property Name
   ```

3. Use the *New-Item* cmdlet to create a new environment variable. The path argument will be dot (.) because you are already on the env:\ PSDrive. The -name argument will be admin, and the value argument will be your given name. The completed command is shown here:

   ```
   New-Item -Path . -Name admin -Value mred
   ```

4. Use the *Get-Item* cmdlet to ensure the *admin* environment variable was properly created. This command is shown here:

   ```
   Get-Item admin
   ```

5. The results of the previous command are shown here:

   ```
   Name                           Value
   ----                           -----
   admin                          mred
   ```

6. Use the up arrow to retrieve the previous command. Pipeline the results to the *Format-List* cmdlet, and choose All Properties. This command is shown here:

   ```
   Get-Item admin | Format-List *
   ```

7. The results of the previous command include the PSPath, PSDrive, and additional information about the newly created environment variable. These results are shown here:

   ```
   PSPath        : Microsoft.PowerShell.Core\Environment::admin
   PSDrive       : Env
   PSProvider    : Microsoft.PowerShell.Core\Environment
   PSIsContainer : False
   Name          : admin
   Key           : admin
   Value         : mred
   ```

8. This concludes this procedure. Leave PowerShell open for the next procedure.

Renaming an environment variable

1. Use the *Get-ChildItem* cmdlet to obtain a listing of all the environment variables. Pipeline the returned object to the *Sort-Object* cmdlet and sort the list on the name property. Use the *gci* and *sort* aliases to reduce typing. The code to do this is shown here:

   ```
   GCI | Sort -Property name
   ```

2. The *admin* environment variable should be near the top of the list of system variables. If it is not, then create it by using the *New-Item* cmdlet. The path argument has a value of dot (.); the name argument has the value of admin; and the value argument should be the user's given name. If this environment variable was created in the previous exercise, then PowerShell will report that it already exists. This is shown here:

   ```
   New-Item -Path . -Name admin -Value mred
   ```

3. Use the *Rename-Item* cmdlet to rename the *admin* environment variable to *super*. The path argument combines both the PSDrive name and the environment variable name. The NewName argument is the desired new name without the PSDrive specification. This command is shown here:

   ```
   Rename-Item -Path env:admin -NewName super
   ```

4. To verify that the old environment variable *admin* has been renamed *super*, press the up arrow two or three times to retrieve the *gci | sort -property name* command. This is command is shown here:

   ```
   GCI | Sort -Property name
   ```

5. This concludes this procedure. Do not close the Windows PowerShell. Leave it open for the next procedure.

Removing an environment variable

1. Use the *Get-ChildItem* cmdlet to obtain a listing of all the environment variables. Pipeline the returned object to the *Sort-Object* cmdlet and sort the list on the name property. Use the *gci* and *sort* aliases to reduce typing. The code to do this is shown here:

   ```
   GCI | Sort -Property name
   ```

2. The *super* environment variable should be in the list of system variables. If it is not, then create it by using the *New-Item* cmdlet. The path argument has a value of dot (.); the name argument has the value of super; and the value argument should be the user's given name. If this environment variable was created in the previous exercise, then PowerShell will report that it already exists. This is shown here:

   ```
   New-Item -Path . -Name super -Value mred
   ```

3. Use the *Remove-Item* cmdlet to remove the *super* environment variable. The name of the item to be removed is typed following the name of the cmdlet. If you are still in the env:\ PSDrive, you will not need to supply a -path argument. The command is shown here:

   ```
   Remove-Item super
   ```

4. Use the *Get-ChildItem* cmdlet to verify that the environment variable *super* has been removed. To do this, press the up arrow 2 or 3 times to retrieve the *gci | sort -property name* command. This command is shown here:

   ```
   GCI | Sort -Property name
   ```

5. This concludes this procedure.

Understanding File System Provider

The file system provider is the easiest Windows PowerShell provider to understand—it provides access to the file system. When Windows PowerShell is launched, it automatically opens on the C:\PSDrive. Using the Windows PowerShell filesystem provider, you can create both directories and files. You can retrieve properties of files and directories, and you can delete them as well. In addition, you can open files and append or overwrite data to the files. This can be done with inline code, or by using the pipelining feature of Windows PowerShell. The commands used in the procedure are in the IdentifyingPropertiesOfDirectories.txt, CreatingFoldersAndFiles.txt, and ReadingAndWritingForFiles.txt files.

Working with directory listings

1. Open Windows PowerShell.
2. Use the *Get-ChildItem* cmdlet to obtain a directory listing of the C:\ drive. Use the *gci* alias to reduce typing. This is shown here:

   ```
   GCI C:\
   ```

3. Use the up arrow to retrieve the *gci C:\* command. Pipeline the object created into a *Where-Object* cmdlet, and look for containers. This will reduce the output to only directories. The modified command is shown here:

   ```
   GCI C:\ | where {$_.psiscontainer}
   ```

4. Use the up arrow to retrieve the *gci C:\ | where {$_.psiscontainer}* command and use the exclamation point (!), meaning *not*, to retrieve only items in the PSDrive that are not directories. The modified command is shown here:

   ```
   GCI C:\ | where {!$_.psiscontainer}
   ```

5. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Identifying properties of directories

1. Use the *Get-ChildItem* cmdlet and supply a value of C:\ for the path argument. Pipeline the resulting object into the *Get-Member* cmdlet. Use the *gci* and *gm* aliases to reduce typing. This command is shown here:

   ```
   GCI  -Path C:\ | GM
   ```

2. The resulting output contains methods, properties, and more. Filter the output by pipelining the output into a *Where-Object* cmdlet and specifying the *membertype* attribute as equal to property. To do this, use the up arrow to retrieve the previous *gci -path C:\ | gm* command. Pipeline the resulting object into the *Where-Object* cmdlet and filter on the *membertype* attribute. The resulting command is shown here:

   ```
   GCI  -Path C:\ | GM | Where {$_.membertype -eq "property"}
   ```

3. The previous *gci -path C:\ | gm | where {$_.membertype -eq "property"}* command returns information on both the System.IO.DirectoryInfo and the System.IO.FileInfo objects. To reduce the output to only the properties associated with the System.IO.FileInfo object, we need to use a compound *Where-Object* cmdlet. Use the up arrow to retrieve the *gci -path C:\ | gm | where {$_.membertype -eq "property"}* command. Add the And conjunction and retrieve objects that have a typename that is like *file*. The modified command is shown here:

   ```
   GCI  -Path C:\ | GM | where {$_.membertype -eq "property" -AND $_.typename -like
   "*file*"}
   ```

4. The resulting output only contains the properties for a System.IO.FileInfo object. These properties are shown here:

   ```
      TypeName: System.IO.FileInfo

   Name              MemberType Definition
   ----              ---------- ----------
   Attributes        Property   System.IO.FileAttributes Attributes {get;set;}
   CreationTime      Property   System.DateTime CreationTime {get;set;}
   CreationTimeUtc   Property   System.DateTime CreationTimeUtc {get;set;}
   ```

```
Directory         Property   System.IO.DirectoryInfo Directory {get;}
DirectoryName     Property   System.String DirectoryName {get;}
Exists            Property   System.Boolean Exists {get;}
Extension         Property   System.String Extension {get;}
FullName          Property   System.String FullName {get;}
IsReadOnly        Property   System.Boolean IsReadOnly {get;set;}
LastAccessTime    Property   System.DateTime LastAccessTime {get;set;}
LastAccessTimeUtc Property   System.DateTime LastAccessTimeUtc {get;set;}
LastWriteTime     Property   System.DateTime LastWriteTime {get;set;}
LastWriteTimeUtc  Property   System.DateTime LastWriteTimeUtc {get;set;}
Length            Property   System.Int64 Length {get;}
Name              Property   System.String Name {get;}
```

5. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Creating folders and files

1. Use the *Get-Item* cmdlet to obtain a listing of files and folders. Pipeline the resulting object into the *Where-Object* cmdlet and use the PsIsContainer property to look for folders. Use the name property to find names that contain the word *my* in them. Use the *gi* alias and the *where* alias to reduce typing. The command is shown here:

```
GI * | Where {$_.PsisContainer -AND $_.name -Like "*my*"}
```

2. If you were following along in the previous chapters, you will have a folder called Mytest off the root of the C:\ drive. Use the *Remove-Item* cmdlet to remove the Mytest folder. Specify the recurse argument to also delete files contained in the C:\Mytest folder. If your location is still set to Env, then change it to C or search for C:\Mytest. The command is shown here:

```
RI mytest -recurse
```

3. Press the up arrow twice and retrieve the *gi * | where {$_.PsisContainer -AND $_.name -Like* "**my**"*}* command to confirm the folder was actually deleted. This command is shown here:

```
GI * | Where {$_.PsisContainer -AND $_.name -Like "*my*"}
```

4. Use the *New-Item* cmdlet to create a folder named Mytest. Use the path argument to specify the path of C:\. Use the name argument to specify the name of Mytest, and use the type argument to tell Windows PowerShell the new item will be a directory. This command is shown here:

```
New-Item -Path C:\ -Name mytest -Type directory
```

5. The resulting output, shown here, confirms the operation:

```
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----          1/4/2007   2:43 AM            mytest
```

6. Use the *New-Item* cmdlet to create an empty text file. To do this, use the up arrow and retrieve the previous *new-item -path C:\ -name Mytest -type directory* command. Edit the path argument so that it is pointing to the C:\Mytest directory. Edit the name argument to specify a text file named Myfile, and specify the type argument as file. The resulting command is shown here:

   ```
   New-Item -Path C:\mytest -Name myfile.txt -type file
   ```

7. The resulting message, shown here, confirms the creation of the file:

   ```
       Directory: Microsoft.PowerShell.Core\FileSystem::C:\mytest

   Mode                LastWriteTime     Length Name
   ----                -------------     ------ ----
   -a---          1/4/2007   3:12 AM          0 myfile.txt
   ```

8. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.

Reading and writing for files

1. Delete Myfile.txt (created in the previous procedure). To do this, use the *Remove-Item* cmdlet and specify the path argument as C:\Mytest\Myfile.txt. This command is shown here:

   ```
   RI -Path C:\mytest\myfile.txt
   ```

2. Use the up arrow twice to retrieve the *new-item -path C:\Mytest -name Myfile.txt -type* file. Add the -value argument to the end of the command line, and supply a value of *my file*. This command is shown here:

   ```
   New-Item -Path C:\mytest -Name myfile.txt -Type file -Value "My file"
   ```

3. Use the *Get-Content* cmdlet to read the contents of Myfile.txt. This command is shown here:

   ```
   Get-Content C:\mytest\myfile.txt
   ```

4. Use the *Add-Content* cmdlet to add additional information to the Myfile.txt file. This command is shown here:

   ```
   Add-Content C:\mytest\myfile.txt -Value "ADDITIONAL INFORMATION"
   ```

5. Press the up arrow twice and retrieve the *get-content C:\Mytest\Myfile.txt* command, which is shown here:

   ```
   Get-Content C:\mytest\myfile.txt
   ```

6. The output from the *get-content C:\Mytest\Myfile.txt* command is shown here:

   ```
   My fileADDITIONAL INFORMATION
   ```

7. Press the up arrow twice, and retrieve the *add-content C:\mytest\Myfile.txt -value* "ADDITIONAL INFORMATION" command to add additional information to the file. This command is shown here:

   ```
   Add-Content C:\mytest\myfile.txt -Value "ADDITIONAL INFORMATION"
   ```

8. Use the up arrow to retrieve the *get-content C:\Mytest\Myfile.txt* command, which is shown here:

   ```
   Get-Content C:\mytest\myfile.txt
   ```

9. The output produced is shown here. Notice that the second time, the "*ADDITIONAL INFORMATION*" command was added to a new line.

   ```
   My fileADDITIONAL INFORMATION
   ADDITIONAL INFORMATION
   ```

10. Use the *Set-Information* cmdlet to overwrite the contents of the Myfile.txt file. Specify the value argument as "*Setting information*". This command is shown here:

    ```
    Set-Content C:\mytest\myfile.txt -Value "Setting information"
    ```

11. Use the up arrow to retrieve the *get-content C:\Mytest\Myfile.txt* command, which is shown here:

    ```
    Get-Content C:\mytest\myfile.txt
    ```

12. The output from the *Get-Content* command is shown here:

    ```
    Setting information
    ```

13. This concludes this procedure.

Understanding the Function Provider

The Function provider provides access to the functions defined in Windows PowerShell. By using the function provider you can obtain a listing of all the functions on your system. You can also add, modify, and delete functions. The function provider uses a file system–based model, and the cmdlets learned earlier also apply to working with functions. The commands used in the procedure are in the ListingAllFunctionsOnTheSystem.txt file.

Listing all functions on the system

1. Open Windows PowerShell.
2. Use the *Set-Location* cmdlet to change the working location to the function PSDrive. This command is shown here:

   ```
   Set-Location function:\
   ```

3. Use the *Get-ChildItem* cmdlet to enumerate all the functions. Do this by using the *gci* alias, as shown here:

   ```
   GCI
   ```

4. The resulting list contains many functions that use *Set-Location* to the different drive letters. A partial view of this output is shown here:

```
CommandType     Name                            Definition
-----------     ----                            ----------
Function        prompt                          'PS ' + $(Get-Location) + $(...
Function        TabExpansion                    ...
Function        Clear-Host                      $spaceType = [System.Managem...
Function        more                            param([string[]]$paths);  if...
Function        help                            param([string]$Name,[string[...
Function        man                             param([string]$Name,[string[...
Function        mkdir                           param([string[]]$paths); New...
Function        md                              param([string[]]$paths); New...
Function        A:                              Set-Location A:
Function        B:                              Set-Location B:
Function        C:                              Set-Location C:
Function        D:                              Set-Location D:
```

5. To return only the functions that are used for drives, use the *Get-ChildItem* cmdlet and pipe the object returned into a *Where-Object* cmdlet. Use the default $_ variable to filter on the definition attribute. Use the like argument to search for definitions that contain the word *set*. The resulting command is shown here:

```
GCI | Where {$_.definition -like "set*"}
```

6. If you are more interested in functions that are not related to drive mappings, then you can use the notlike argument instead of like. The easiest way to make this change is to use the up arrow and retrieve the *gci | where {$_.definition -like "set*"}* and then change the filter from *like* to *notlike*. The resulting command is shown here:

```
GCI | Where {$_.definition -notlike "set*"}
```

7. The resulting listing of functions is shown here:

```
CommandType     Name                            Definition
-----------     ----                            ----------
Function        prompt                          'PS' + $(Get-Location) + $(...
Function        TabExpansion                    ...
Function        Clear-Host                      $spaceType = [System.Managem...
Function        more                            param([string[]]$paths);  if...
Function        help                            param([string]$Name,[string[...
Function        man                             param([string]$Name,[string[...
Function        mkdir                           param([string[]]$paths); New...
Function        md                              param([string[]]$paths); New...
Function        pro                             notepad $profile
```

8. Use the *Get-Content* cmdlet to retrieve the text of the *md* function. This is shown here:

```
Get-Content md
```

9. The content of the *md* function is shown here:

```
param([string[]]$paths); New-Item -type directory -path $paths
```

10. This concludes this procedure.

Understanding the Registry Provider

The registry provider provides a consistent and easy way to work with the registry from within Windows PowerShell. Using the registry provider, you can search the registry, create new registry keys, delete existing registry keys, and modify values and access control lists (ACLs) from within Windows PowerShell. The commands used in the procedure are in the UnderstandingTheRegistryProvider.txt file. Two PSDrives are created by default. To identify the PSDrives that are supplied by the registry provider, you can use the *Get-PSDrive* cmdlet, pipeline the resulting objects into the *Where-Object* cmdlet, and filter on the provider property while supplying a value that is like the word registry. This command is shown here:

```
get-psDrive | where {$_.Provider -like "*Registry*"}
```

The resulting list of PSDrives is shown here:

```
Name       Provider      Root                                   CurrentLocation
----       --------      ----                                   ---------------
HKCU       Registry      HKEY_CURRENT_USER
HKLM       Registry      HKEY_LOCAL_MACHINE
```

Obtaining a listing of registry keys

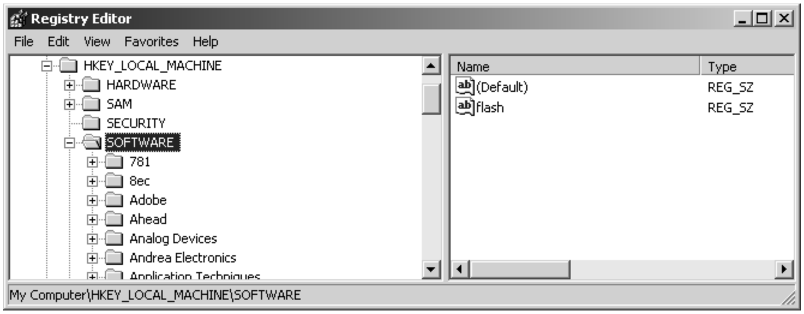
1. Open Windows PowerShell.
2. Use the *Get-ChildItem* cmdlet and supply the HKLM:\ PSDrive as the value for the path argument. Specify the software key to retrieve a listing of software applications on the local machine. The resulting command is shown here:

   ```
   GCI -path HKLM:\software
   ```

3. A partial listing of similar output is shown here. The corresponding keys, as seen in Regedit.exe, are shown in Figure 3-6.

   ```
      Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software

   SKC  VC Name                           Property
   ---  -- ----                           --------
     2   0 781                            {}
     1   0 8ec                            {}
     4   0 Adobe                          {}
    12   0 Ahead                          {}
     2   1 Analog Devices                 {ProductDir}
     2   0 Andrea Electronics             {}
     1   0 Application Techniques         {}
   ```

4. This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.



Name	Type
(Default)	REG_SZ
flash	REG_SZ

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE

Figure 3-6 A Regedit.exe similar view of HKEY_LOCAL_MACHINE\SOFTWARE

Searching for hotfixes

1. Use the *Get-ChildItem* cmdlet and supply a value for the path argument. Use the HKLM:\ PSDrive and supply a path of Software\Microsoft\Windows NT\CurrentVersion\Hotfix. Because there is a space in Windows NT, you will need to use a single quote (') to encase the command. You can use *Tab completion* to assist with the typing. The completed command is shown here:

```
GCI -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\HotFix'
```

2. The resulting similar list of hotfixes is seen in the output here, in abbreviated fashion:

```
   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Micros
oft\Windows NT\CurrentVersion\HotFix

SKC  VC Name                           Property
---  -- ----                           --------
  1   8 KB873333                       {Installed, Comments, Backup Dir, Fix...
  1   8 KB873339                       {Installed, Comments, Backup Dir, Fix...
  1   8 KB883939                       {Installed, Comments, Backup Dir, Fix...
  1   8 KB885250                       {Installed, Comments, Backup Dir, Fix...
```

3. To retrieve information on a single hotfix, you will need to add a *Where-Object* cmdlet. You can do this by using the up arrow to retrieve the previous *gci -path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\HotFix'* command and pipelining the resulting object into the *Where-Object* cmdlet. Supply a value for the name property, as seen in the code listed here. Alternatively, supply a "KB" number from the previous output.

```
GCI -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\HotFix' | where
{$_.Name -like "*KB928388"}
```

4. This concludes this procedure.

Understanding the Variable Provider

The variable provider provides access to the variables that are defined within Windows PowerShell. These variables include both user-defined variables, such as *$mred*, and system-defined variables, such as *$host*. You can obtain a listing of the cmdlets designed to work specifically with variables by using the *Get-Help* cmdlet and specifying the asterisk (*) variable. The commands used in the procedure are in the UnderstandingTheVariableProvider.txt and WorkingWithVariables.txt files. To return only cmdlets, we use the *Where-Object* cmdlet and filter on the category that is equal to cmdlet. This command is shown here:

```
Get-Help *variable | Where-Object {$_.category -eq ÒcmdletÓ}
```

The resulting list contains five cmdlets but is a little jumbled and difficult to read. So let's modify the preceding command and specify the properties to return. To do this, use the up arrow and pipeline the returned object into the *Format-List* cmdlet. Add the three properties we are interested in: name, category, and synopsis. The revised command is shown here:

```
Get-Help *variable | Where-Object {$_.category -eq "cmdlet"} |
Format-List name, category, synopsis
```

The resulting output is much easier to read and understand. It is shown here:

```
Name     : Get-Variable
Category : Cmdlet
Synopsis : Gets the variables in the current console.

Name     : New-Variable
Category : Cmdlet
Synopsis : Creates a new variable.

Name     : Set-Variable
Category : Cmdlet
Synopsis : Sets the value of a variable. Creates the variable if one with the requested
name does not exist.

Name     : Remove-Variable
Category : Cmdlet
Synopsis : Deletes a variable and its value.

Name     : Clear-Variable
Category : Cmdlet
Synopsis : Deletes the value of a variable.
```

Working with variables

1. Open Windows PowerShell.
2. Use the *Set-Location* cmdlet to set the working location to the variable PSDrive. Use the *sl* alias to reduce typing needs. This command is shown here:

   ```
   SL variable:\
   ```

3. Produce a complete listing of all the variables currently defined in Windows PowerShell. To do this, use the *Get-ChildItem* cmdlet. You can use the alias *gci* to produce this list. The command is shown here:

   ```
   Get-ChildItem
   ```

4. The resulting list is jumbled. Use the up arrow to retrieve the *Get-ChildItem* command, and pipeline the resulting object into the *Sort-Object* cmdlet. Sort on the name property. This command is shown here:

   ```
   Get-ChildItem | Sort {$_.Name}
   ```

5. The output from the previous command is shown here:

   ```
   Name                           Value
   ----                           -----
   $                              }
   ?                              True
   ^                              Get-ChildItem
   _
   args                           {}
   ConfirmPreference              High
   ConsoleFileName
   DebugPreference                SilentlyContinue
   Error                          {System.Management.Automation.ParseException:...
   ErrorActionPreference          Continue
   ErrorView                      NormalView
   ExecutionContext               System.Management.Automation.EngineIntrinsics
   false                          False
   FormatEnumerationLimit         4
   HOME                           C:\Documents and Settings\edwils.NORTHAMERICA
   Host                           System.Management.Automation.Internal.Host.In...
   input                          System.Array+SZArrayEnumerator
   LASTEXITCODE                   0
   lastWord                       get-c
   line                           get-c
   MaximumAliasCount              4096
   MaximumDriveCount              4096
   MaximumErrorCount              256
   MaximumFunctionCount           4096
   MaximumHistoryCount            64
   MaximumVariableCount           4096
   mred                           mred
   MyInvocation                   System.Management.Automation.InvocationInfo
   NestedPromptLevel              0
   null
   OutputEncoding                 System.Text.ASCIIEncoding
   PID                            292
   PROFILE                        C:\Documents and Settings\edwils.NORTHAMERICA...
   ProgressPreference             Continue
   PSHOME                         C:\WINDOWS\system32\WindowsPowerShell\v1.0
   PWD                            Variable:\
   ReportErrorShowExceptionClass  0
   ReportErrorShowInnerException  0
   ReportErrorShowSource          1
   ```

```
ReportErrorShowStackTrace      0
ShellId                        Microsoft.PowerShell
StackTrace                        at System.Number.StringToNumber(String str...
true                           True
VerbosePreference              SilentlyContinue
WarningPreference              Continue
WhatIfPreference               0
```

6. Use the *Get-Variable* cmdlet to retrieve a specific variable. Use the *ShellId* variable. You can use *Tab completion* to speed up typing. The command is shown here:

```
Get-Variable ShellId
```

7. Use the up arrow to retrieve the previous *Get-Variable ShellId* command. Pipeline the object returned into a *Format-List* cmdlet and return all properties. This is shown here:

```
Get-Variable ShellId | Format-List *
```

8. The resulting output includes the description of the variable, value, and other information shown here:

```
Name        : ShellId
Description : The ShellID identifies the current shell.  This is used by #Requires.
Value       : Microsoft.PowerShell
Options     : Constant, AllScope
Attributes  : {}
```

9. Create a new variable called *administrator*. To do this, use the *New-Variable* cmdlet. This command is shown here:

```
New-Variable administrator
```

10. Use the *Get-Variable* cmdlet to retrieve the new administrator variable. This command is shown here:

```
Get-Variable administrator
```

11. The resulting output is shown here. Notice that there is no value for the variable.

```
Name                           Value
----                           -----
administrator
```

12. Assign a value to the new administrator variable. To do this, use the *Set-Variable* cmdlet. Specify the *administrator* variable name, and supply your given name as the value for the variable. This command is shown here:

```
Set-Variable administrator -value mred
```

13. Use the up arrow one time to retrieve the previous *Get-Variable administrator* command. This command is shown here:

```
Get-Variable administrator
```

14. The output displays both the variable name and the value associated with the variable. This is shown here:

```
Name                           Value
----                           -----
administrator                  mred
```

15. Use the *Remove-Variable* cmdlet to remove the administrator variable you previously created. This command is shown here:

```
Remove-Variable administrator
```

16. Use the up arrow one time to retrieve the previous *Get-Variable administrator* command. This command is shown here:

```
Get-Variable administrator
```

17. The variable has been deleted. The resulting output is shown here:

```
Get-Variable : Cannot find a variable with name 'administrator'.
At line:1 char:13
+ Get-Variable <<<< administrator
```

18. This concludes this procedure.

Exploring the Certificate Provider: Step-by-Step Exercises

In this exercise, we explore the use of the Certificate provider in Windows PowerShell.

1. Start Windows PowerShell.
2. Obtain a listing of all the properties available for use with the *Get-ChildItem* cmdlet by piping the results into the *Get-Member* cmdlet. To filter out only the properties, pipeline the results into a *Where-Object* cmdlet and specify the *membertype* to be equal to property. This command is shown here:

```
Get-ChildItem |Get-Member | Where-Object {$_.membertype -eq "property"}
```

3. Set your location to the certificate drive. To identify the certificate drive, use the *Get-PSDrive* cmdlet. Use the *Where-Object* cmdlet and filter on names that begin with the letter c. This is shown here:

```
Get-PSDrive |where {$_.name -like "c*"}
```

4. The results of this command are shown here:

```
Name       Provider      Root                                          CurrentLocation
----       --------      ----                                          ---------------
C          FileSystem    C:\
cert       Certificate   \
```

5. Use the *Set-Location* cmdlet to change to the certificate drive.

```
Sl cert:\
```

6. Use the *Get-ChildItem* cmdlet to produce a listing of all the certificates on the machine.

```
GCI
```

7. The output from the previous command is shown here:

```
Location   : CurrentUser
StoreNames : {?, UserDS, AuthRoot, CA...}

Location   : LocalMachine
StoreNames : {?, AuthRoot, CA, AddressBook...}
```

8. The listing seems somewhat incomplete. To determine whether there are additional certificates installed on the machine, use the *Get-ChildItem* cmdlet again, but this time specify the recurse argument. Modify the previous command by using the up arrow. The command is shown here:

```
GCI -recurse
```

9. The output from the previous command seems to take a long time to run and produces hundreds of lines of output. To make the listing more readable, pipe the output to a text file, and then open the file in Notepad. The command to do this is shown here:

```
GCI -recurse >C:\a.txt;notepad.exe a.txt
```

10. This concludes this step-by-step exercise.

One Step Further: Examining the Environment Provider

In this exercise, we work with the Windows PowerShell Environment provider.

1. Start Windows PowerShell.
2. Use the *New-PSDrive* cmdlet to create a drive mapping to the alias provider. The name of the new PSDrive will be *al*. The PSProvider is alias, and the root will be dot (.). This command is shown here:

```
new-PSDrive -name al -PSProvider alias -Root .
```

3. Change your working location to the new PSDrive you called *al*. To do this, use the *sl* alias for the *Set-Location* cmdlet. This is shown here:

```
SL al:\
```

4. Use the *gci* alias for the *Get-ChildItem* cmdlet, and pipeline the resulting object into the *Sort-Object* cmdlet by using the *sort* alias. Supply name as the property to sort on. This command is shown here:

```
GCI | Sort -Property name
```

5. Use the up arrow to retrieve the previous *gci | sort -property name* command and modify it to use a *Where-Object* cmdlet to return aliases only when the name is greater than the letter t. Use the *where* alias to avoid typing the entire name of the cmdlet. The resulting command is shown here:

   ```
   GCI | sort -Property name | Where {$_.Name -gt "t"}c
   ```

6. Change your location back to the C:\ drive. To do this, use the *sl* alias and supply the C:\ argument. This is shown here:

   ```
   SL C:\
   ```

7. Remove the PSDrive mapping for al. To do this, use the *Remove-PSDrive* cmdlet and supply the name of the PSDrive to remove. Note, this command does not want a trailing colon (:) or colon with backslash (:\). The command is shown here:

   ```
   Remove-PSDrive al
   ```

8. Use the *Get-PSDrive* cmdlet to ensure the al drive was removed. This is shown here:

   ```
   Get-PSDrive
   ```

9. Use the *Get-Item* cmdlet to obtain a listing of all the environment variables. Use the path argument and supply env:\ as the value. This is shown here:

   ```
   Get-Item -Path env:\
   ```

10. Use the up arrow to retrieve the previous command, and pipeline the resulting object into the *Get-Member* cmdlet. This is shown here:

    ```
    Get-Item -Path env:\ | Get-Member
    ```

11. The results from the previous command are shown here:

    ```
       TypeName: System.Collections.Generic.Dictionary'2+ValueCollection[[System.St
    ring, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e0
    89],[System.Collections.DictionaryEntry, mscorlib, Version=2.0.0.0, Culture=neu
    tral, PublicKeyToken=b77a5c561934e089]]

    Name          MemberType   Definition
    ----          ----------   ----------
    CopyTo        Method       System.Void CopyTo(DictionaryEntry[] array, Int32...
    Equals        Method       System.Boolean Equals(Object obj)
    GetEnumerator Method       System.Collections.Generic.Dictionary`2+ValueColl...
    GetHashCode   Method       System.Int32 GetHashCode()
    GetType       Method       System.Type GetType()
    get_Count     Method       System.Int32 get_Count()
    ToString      Method       System.String ToString()
    PSDrive       NoteProperty System.Management.Automation.PSDriveInfo PSDrive=Env
    PSIsContainer NoteProperty System.Boolean PSIsContainer=True
    PSPath        NoteProperty System.String PSPath=Microsoft.PowerShell.Core\En...
    PSProvider    NoteProperty System.Management.Automation.ProviderInfo PSProvi...
    Count         Property     System.Int32 Count {get;}
    ```

12. Press the up arrow twice to return to the *get-item -path env:* command. Use the Home key to move your insertion point to the beginning of the line. Add a variable called *$objEnv* and use it to hold the object returned by the *get-item -path env:* command. The completed command is shown here:

    ```
    $objEnv=Get-Item -Path env:\
    ```

13. From the listing of members of the environment object, find the count property. Use this property to print out the total number of environment variables. As you type **$o**, try to use *Tab completion* to avoid typing. Also try to use *Tab completion* as you type the *c* in count. The completed command is shown here:

    ```
    $objEnv.Count
    ```

14. Examine the methods of the object returned by *get-item -path env:*. Notice there is a Get_Count method. Let's use that method. The code is shown here:

    ```
    $objEnv.Get_count
    ```

15. When this code is executed, however, the results define the method rather than execute the Get_Count method. These results are shown here:

    ```
    MemberType          : Method
    OverloadDefinitions : {System.Int32 get_Count()}
    TypeNameOfValue     : System.Management.Automation.PSMethod
    Value               : System.Int32 get_Count()
    Name                : get_Count
    IsInstance          : True
    ```

16. To retrieve the actual number of environment variables, we need to use empty parentheses is at the end of the method. This is shown here:

    ```
    $objEnv.Get_count()
    ```

17. If you want to know exactly what type of object you have contained in the *$objEnv* variable, you can use the GetType method, as shown here:

    ```
    $objEnv.GetType()
    ```

18. This command returns the results shown here:

    ```
    IsPublic IsSerial Name                                     BaseType
    -------- -------- ----                                     --------
    False    True     ValueCollection                          System.Object
    ```

19. This concludes this one step further exercise.

Chapter 3 Quick Reference

To	Do This
Produce a listing of all variables defined in a Windows PowerShell session	Use the *Set-Location* cmdlet to change location to the variable PSDrive, then use the *Get-ChildItem* cmdlet
Obtain a listing of all the aliases	Use the *Set-Location* cmdlet to change location to the alias PSDrive, then use the *Get-ChildItem* cmdlet to produce a listing of aliases. Pipeline the resulting object into the *Where-Object* cmdlet and filter on the name property for the appropriate value
Delete a directory that is empty	Use the *Remove-Item* cmdlet and supply the name of the directory
Delete a directory that contains other items	Use the *Remove-Item* cmdlet and supply the name of the directory and specify the recurse argument
Create a new text file	Use the *New-Item* cmdlet and specify the -path argument for the directory location. Supply the name argument, and specify the type argument as file. Example: *new-item -path C:\Mytest -name Myfile.txt -type file*
Obtain a listing of registry keys from a registry hive	Use the *Get-ChildItem* cmdlet and specify the appropriate PSDrive name for the -path argument. Complete the path with the appropriate registry path. Example: *gci -path HKLM:\software*
Obtain a listing of all functions on the system	Use the *Get-ChildItem* cmdlet and supply the PSDrive name of *function:* to the path argument. Example: *gci -path function:*