CHAPTER 3

Understanding and using Windows 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 New-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 Microsoft .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 by using a common set of cmdlets—Get-ChildItem, for example—to work with different types of data. This chapter demonstrates how to take advantage of the Windows PowerShell providers.

Understanding Windows PowerShell providers

By identifying the providers installed with Windows PowerShell, you 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 methodology you would use to obtain a listing of files on the hard drive. The only difference is the specific name associated with each drive. Developers familiar with Windows or .NET programming can create new providers, but writing a provider can be complex. (See msdn.microsoft.com/en-us/library/windows/desktop/ee126192(v=vs.85).aspx for more information.) When a new provider is created, it might ship in a module or in a snap-in. A snap-in is
a dynamic-link library (DLL) file that must be installed in Windows PowerShell. As such, snap-ins are older technology that require elevated rights to install. After a snap-in has been installed, it cannot be uninstalled unless the developer provides removal logic—however, the snap-in can be removed from the current Windows PowerShell console. The preferred way to ship a provider is via a Windows PowerShell module. Modules are installable via an `xcopy` deployment and therefore do not necessarily require admin rights.

To obtain a listing of the providers, use the `Get-PSProvider` cmdlet. This command produces the following list on a default installation of Windows PowerShell.

```
Note Windows 10 does not load either the WSMAN or the Certificate provider until their respective drives are accessed.
```

```powershell
PS C:\> Get-PSProvider
```

### Understanding the alias provider

In Chapter 1, “Overview of Windows PowerShell 5.0,” I presented the various help tools that are 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 computer, 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 the Windows PowerShell console.
2. Obtain a listing of the providers by using the `Get-PSProvider` cmdlet.
3. The Windows PowerShell drive (PS drive) associated with the alias provider is called Alias. This is shown 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.

```powershell
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 gci alias in place of Get-ChildItem. This is shown here.

```powershell
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.

```powershell
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. (Keep in mind that different providers expose different objects that will have different properties.)

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

7. Press the Up Arrow key twice, and edit the previous filter to include only definitions that contain the word set. The modified filter is shown here.

```powershell
gci | Where-Object definition -like "set*"
```

8. The results of this command are shown here.

```
PS Alias:
```

<table>
<thead>
<tr>
<th>CommandType</th>
<th>Name</th>
<th>Version</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias</td>
<td>cd -&gt; Set-Location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>chdir -&gt; Set-Location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>sal -&gt; Set-Alias</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>sbp -&gt; Set-PSBreakpoint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>sc -&gt; Set-Content</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>scb -&gt; Set-Clipboard</td>
<td>3.1.0.0</td>
<td>Mic...</td>
</tr>
<tr>
<td>Alias</td>
<td>set -&gt; Set-Variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>si -&gt; Set-Item</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>sl -&gt; Set-Location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>sp -&gt; Set-ItemProperty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>sv -&gt; Set-Variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>swmi -&gt; Set-WmiInstance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9. Press the Up Arrow key three times, and edit the previous filter to include only names of aliases that contain the letter w. This revised command is shown here.

   gci | Where-Object name -like "*w*"

   The results from this command will be similar to those shown here.

<table>
<thead>
<tr>
<th>CommandType</th>
<th>Name</th>
<th>Version</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias</td>
<td>fw -&gt; Format-Wide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>gwmi -&gt; Get-WmiObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>iwm -&gt; Invoke-WmiMethod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>iwr -&gt; Invoke-WebRequest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>pwd -&gt; Get-Location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>rwmi -&gt; Remove-WmiObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>swmi -&gt; Set-WmiInstance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>wget -&gt; Invoke-WebRequest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>where -&gt; Where-Object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>wjb -&gt; Wait-Job</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias</td>
<td>write -&gt; Write-Output</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10. In the preceding list, note that where is an alias for the Where-Object cmdlet. Press the Up Arrow key 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 shown here.

   gci | where name -like "*w*"

**Caution** When using the Set-Location cmdlet to switch to a different PS drive, you must follow the name of the PS drive with a colon. A trailing forward slash or backward slash is optional. An error will be generated if the colon is left out or if the complete drive name is not supplied, as shown in Figure 3-1. I prefer to use the backward slash (\) because it is consistent with normal Windows file system operations.

![PowerShell](image)

**FIGURE 3-1** Using Set-Location without a colon or complete name results in an error.
Creating new aliases

One of the useful things about providers is that they enable you to use the same methodology to perform standard activities. For example, to create a new alias, I can use the New-Item cmdlet while on the Alias drive. I need to specify only the Name and the Value parameters. Though this does not directly expose all of the configurable properties of an Alias object, it does expose the main properties: the name of the new alias and the command that the alias resolves to. The following is an example of creating a new alias by using the New-Item cmdlet. This example creates a new alias named listing that, when called, will run the Get-ChildItem cmdlet.

```
PS Alias:\> New-Item -Name listing -Value Get-ChildItem
```

<table>
<thead>
<tr>
<th>CommandType</th>
<th>Name</th>
<th>Version</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias</td>
<td>listing</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>listing</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>-&gt; Get-ChildItem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Understanding the certificate provider

The preceding section explored working with the alias provider. Because the file system model applies to the certificate provider in much the same way as it does to the alias provider, many of the same cmdlets can be used. To find information about the certificate provider, use the Get-Help cmdlet and search for about_Providers. If you are unsure what articles in Help might be related to certificates, you can use the wildcard asterisk (*) parameter. This command is shown here.

```
Get-Help *cer*
```

In addition to allowing you to use the certificate provider, Windows PowerShell gives you the ability to sign scripts; Windows PowerShell can work with both signed and unsigned scripts. The certificate provider gives you the ability search for, copy, move, and delete certificates. By using the certificate provider, you can open the Certificates Microsoft Management Console (MMC). The commands used in the following procedure use the certificate provider to obtain a listing of the certificates installed on the local computer.

**Obtaining a listing of certificates**

1. Open the Windows PowerShell console.
2. Set your location to the cert PS drive. To do this, use the Set-Location cmdlet, as shown here.

```
Set-Location cert:\
```
3. Use the `Get-ChildItem` cmdlet, shown here, to produce a list of the certificates.

   Get-ChildItem

   The list produced is shown here.

   Location : CurrentUser
   StoreNames : {SmartCardRoot, Root, Trust, AuthRoot...}

   Location : LocalMachine
   StoreNames : {TrustedPublisher, ClientAuthIssuer, Remote Desktop, Root...}

4. Use the `-Recurse` parameter to cause the `Get-ChildItem` cmdlet to produce a list of all the certificate stores and the certificates in those 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

5. Use the `-Path` parameter for `Get-ChildItem` to produce a listing of certificates in another store, without using the `Set-Location` cmdlet to change your current location. Use the `gci` alias, as shown here.

   GCI -Path currentUser

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

   Name : SmartCardRoot
   Name : Root
   Name : Trust
   Name : AuthRoot
   Name : CA
   Name : UserDS
   Name : Disallowed
   Name : My
   Name : TrustedPeople
   Name : TrustedPublisher
   Name : ClientAuthIssuer

6. 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 (`sl` is an alias for the `Set-Location` cmdlet). This command is shown here.

   sl currentUser\authroot
7. Use the `Get-ChildItem` cmdlet to produce a listing of certificates in the currentuser\authroot certificate store that contain the name *Entrust* in the subject field. Use the `gci` alias to reduce the amount of typing. Use the *where* method instead of pipelining the output to *Where-Object*. The code to do this is shown here.

```
(gci).where({$psitem.subject -match 'entrust'})
```

On my machine, there are three certificates listed; they are shown here.

<table>
<thead>
<tr>
<th>Thumbprint</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>B31EB1B740E36C8402DADC37D44DF5D4674952F9</td>
<td>CN=Entrust Root Certification Authority...</td>
</tr>
<tr>
<td>99A69BEC61AFE886B4D28B2007CBE54FC317E1539</td>
<td>CN=Entrust.net Secure Server Certificate...</td>
</tr>
<tr>
<td>503006091D97D4F5AE39F7CBE7927D7D652D3431</td>
<td>CN=Entrust.net Certification Authority...</td>
</tr>
</tbody>
</table>

8. Use the Up Arrow key, and edit the previous command so that it will return only certificates that contain the phrase *2006* in the subject property. The revised command is shown here.

```
(gci).where({$psitem.subject -match '2006'})
```

9. The resulting output on my machine contains three certificates. The results display has been truncated. This is shown here.

<table>
<thead>
<tr>
<th>Thumbprint</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>B31EB1B740E36C8402DADC37D44DF5D4674952F9</td>
<td>CN=Entrust Root Certification Authority...</td>
</tr>
</tbody>
</table>

10. Use the Up Arrow key, and edit the previous command. This time, change the *where* method so that it filters on the thumbprint attribute that is equal to B31EB1B740E36C-8402DADC37D44DF5D4674952F9. 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({$psitem.thumbprint -eq 'B31EB1B740E36C8402DADC37D44DF5D4674952F9'})
```

**FIGURE 3-2** Highlight items to copy by using a mouse.
Troubleshooting  If copying from inside the Windows PowerShell console window does not work, you might need to enable QuickEdit mode. To do this, right-click the Windows PowerShell icon in the upper-left corner of the Windows PowerShell window. Click Properties, click the Options tab, and then select the QuickEdit Mode check box. This is shown in Figure 3-3.

![Figure 3-3](image)

**Figure 3-3** Select the QuickEdit Mode check box to turn on Clipboard support.

11. To view 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.

```powershell
(gci).where({$psitem.thumbprint -eq '4EB6D578499B1CCF5F581EAD56BE3D9B6744A5E5'}) | Format-List *
```

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

- **PSPath**: Microsoft.PowerShell.Security\Certificate::CurrentUser\AuthRoot\4EB6D578499B1CCF5F581EAD56BE3D9B6744A5E5
- **PSParentPath**: Microsoft.PowerShell.Security\Certificate::CurrentUser\AuthRoot
- **PSChildName**: 4EB6D578499B1CCF5F581EAD56BE3D9B6744A5E5
- **PSDrive**: Cert
- **PSPort**: Microsoft.PowerShell.Security\Certificate
- **PSIsContainer**: False
- **EnhancedKeyUsageList**: {Server Authentication (1.3.6.1.5.5.7.3.1), Client
12. Open the Certificates MMC file. This MMC file is called Certmgr.msc; you can launch it by entering the name inside Windows PowerShell, as shown here.

Certmgr.msc

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

Invoke-Item cert:\

14. Compare the information obtained from Windows PowerShell with the information displayed in the Certificates MMC. It should be the same. The certificate is shown in Figure 3-4.

```powershell
Get-ItemProperty -Path 'Cert:\VeriSign-Class-3-Primary-Cert-G5.msp' -Name Properties -PSDrive cert -ErrorAction SilentlyContinue | Select-Object -Property Authentication, SecureEmail, CodeSigning, DnsNameList, SendAsTrustedIssuer, EnrollmentPolicyEndPoint, EnrollmentServerEndPoint, PolicyId, Archived, Extensions, FriendlyName, IssuerName, NotAfter, NotBefore, HasPrivateKey, PrivateKey, PublicKey, RawData, SerialNumber, SubjectName, SignatureAlgorithm, Thumbprint, Version, Handle, Issuer, Subject
```
FIGURE 3-4 Certmgr.msc can be used to examine certificate properties.

This concludes this procedure.

Searching for specific certificates

To search for specific certificates, you might want to examine the *subject* property. For example, the following code examines the *subject* property of every certificate in the current user store beginning at the root level. It does a recursive search and returns only the certificates that contain the word *test* in some form in the *subject* property. This command and associated output are shown here.

```powershell
PS C:\Users\administrator.IAMMRED> dir Cert:\CurrentUser -Recurse | ? subject -match 'test'
```


<table>
<thead>
<tr>
<th>Thumbprint</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>8A334AA8052DD244A647306A768178FA215F344</td>
<td>CN=Microsoft Testing Root Certificate A...</td>
</tr>
<tr>
<td>2BD63D28D7BCD0E251195AEBS19243C13142EBC3</td>
<td>CN=Microsoft Test Root Authority, OU=Mi...</td>
</tr>
</tbody>
</table>

To delete these *test* certificates, you just pipeline the results of the previous command to the *Remove-Item* cmdlet.

**Note** When performing any operation that might alter system state, it is a good idea to use the *-WhatIf* parameter to prototype the command prior to actually executing it.
The following command uses the `-WhatIf` parameter from `Remove-Item` to prototype the command to remove all of the certificates from the current user store that contain the word `test` in the `subject` property. When this is completed, retrieve the command by using the Up Arrow key and remove the `-WhatIf` switch parameter from the command prior to actual execution. This technique appears here.

```powershell
PS C:\Users\administrator.IAMMRED> dir Cert:\CurrentUser -Recurse | ? subject -match 'test' | Remove-Item -WhatIf
What if: Performing operation "Remove certificate" on Target "Item: CurrentUser\Root\8A334AA8052DD244A647306A76B8178FA215F344 ".
What if: Performing operation "Remove certificate" on Target "Item: CurrentUser\Root\2BD63D28D78CD0E251195AE6519243C13142EB3 ".
PS C:\Users\administrator.IAMMRED> dir Cert:\CurrentUser -Recurse | ? subject -match 'test' | Remove-Item
```

### Finding expiring certificates

A common task for companies that use certificates is to identify certificates that either have expired or that will expire soon. By using the certificate provider, you can easily identify expired certificates. To do this, use the `notafter` property from the certificate objects returned from the certificate drives. One approach is to look for certificates that expire prior to a specific date. This technique is shown here.

```powershell
PS Cert:\CurrentUser> dir -Recurse | where notafter -lt "5/1/2015"
```

A more flexible approach is to use the current date—that way, each time the command runs, it retrieves expired certificates. This technique appears here.

```powershell
PS Cert:\CurrentUser> dir -Recurse | where notafter -lt (Get-Date)
```

One problem with just using the `Get-ChildItem` cmdlet on the current user store is that it returns certificate stores in addition to certificates. To obtain only certificates, you must filter out the `psiscontainer` property. Because you will also need to filter based upon date, you can no longer use the simple `Where-Object` syntax. The `$._` character represents the current certificate as it comes across the pipeline. Because you’re comparing two properties, you must repeat the `$_` character for each property. The following command retrieves the expiration dates, thumbprints, and subjects of all expired certificates. It also creates a table displaying the information. (The command is a single logical command, but it is broken at the pipe character to permit better display in the book.)

```powershell
PS Cert:\CurrentUser> dir -Recurse | where { !$_.psiscontainer -AND $_.notafter -lt (Get-Date) } | ft notafter, thumbprint, subject -AutoSize -Wrap
```

**Caution** All versions of Windows are released with expired certificates to permit verification of old executable files that were signed with those certificates. Do not arbitrarily delete an expired certificate—if you do, you could cause serious damage to your system.
If you want to identify certificates that will expire in the next 30 days, you use the same technique involving a compound *Where-Object* command. The command shown here identifies certificates expiring in the next 30 days.

```powershell
PS Cert:\CurrentUser> dir -Recurse |
where { $_.NotAfter -gt (Get-Date) -AND $_.NotAfter -le (Get-Date).Add(30) }
```

### 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 enter `set`, you will obtain a listing of all the environment variables defined on the system. (You can run the old-fashioned command prompt inside Windows PowerShell.)

**Note** It is easy to forget that you are running the CMD prompt when you are inside of the Windows PowerShell console. Entering `exit` returns you to Windows PowerShell. The best way to determine whether you are running the command shell or Windows PowerShell is to examine the prompt. The default Windows PowerShell prompt is `PS C:>`, assuming that you are working on drive C.

If you use the `echo` command in the CMD interpreter to print out the value of `%windir%`, you will obtain the results shown in Figure 3-5.

![Figure 3-5](https://via.placeholder.com/150)

**FIGURE 3-5** Use `set` at a CMD prompt to view environment variables.

Various applications and tools use environment variables 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 the Windows PowerShell console.
2. Obtain a listing of the PS drives by using the `Get-PSDrive` cmdlet. This is shown here.
   ```powershell
   Get-PSDrive
   ```
3. Note that the Environment PS drive is called `Env`. Use the `Env` name with the `Set-Location` cmdlet to change to the Environment PS drive. This is shown here.
   ```powershell
   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.
   ```powershell
   Get-Item *
   ```
5. Use the `Sort-Object` cmdlet to produce an alphabetical listing of all the environment variables by name. Use the Up Arrow key to retrieve the previous command, and then pipeline the returned object into the `Sort-Object` cmdlet. Use the `-Property` parameter, and supply `name` as the value. This command is shown here.
   ```powershell
   Get-Item * | Sort-Object -Property name
   ```
6. Use the `Get-Item` cmdlet to retrieve the value associated with the `windir` environment variable. This is shown here.
   ```powershell
   Get-Item windir
   ```
7. Use the Up Arrow key to retrieve the previous command. Pipeline the object returned to the `Format-List` cmdlet and use the wildcard character to print out all the properties of the object. The modified command is shown here.
   ```powershell
   Get-Item windir | Format-List *
   ```

The properties and their associated values are shown here.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSPath</td>
<td>Microsoft.PowerShell.Core\Environment::windir</td>
</tr>
<tr>
<td>PSDrive</td>
<td>Env</td>
</tr>
<tr>
<td>PSPProvider</td>
<td>Microsoft.PowerShell.Core\Environment</td>
</tr>
<tr>
<td>PSIsContainer</td>
<td>False</td>
</tr>
<tr>
<td>Name</td>
<td>windir</td>
</tr>
<tr>
<td>Key</td>
<td>windir</td>
</tr>
<tr>
<td>Value</td>
<td>C:\WINDOWS</td>
</tr>
</tbody>
</table>

This concludes this procedure. Do not close Windows PowerShell. Leave it open for the next procedure.
Creating a temporary new environment variable

1. You should still be in the Environment PS drive from the previous procedure. If you are 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 by using the `name` property. To reduce typing, use the `gi` alias and the `sort` alias. This is shown here.

   ```powershell
   gi * | sort -Property name
   ```

3. Use the `New-Item` cmdlet to create a new environment variable. The `-Path` parameter will be dot (.) because you are already on the env:\ PS drive; therefore, it can be omitted in the command. The `-Name` parameter will be `admin`, and the `-Value` parameter will be your given name. The completed command is shown here.

   ```powershell
   New-Item -Name admin -Value mred
   ```

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

   ```powershell
   Get-Item admin
   ```

   The results of the previous command are shown here.

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

5. Use the Up Arrow key to retrieve the previous command. Pipeline the results to the `Format-List` cmdlet and choose All Properties. This command is shown here.

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

   The results of the previous command include the PS path, PS drive, and additional information about the newly created environment variable. These results are shown here.

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

   The new environment variable exists until you close the Windows PowerShell console.

   This concludes this procedure. Leave Windows 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.

   ```powershell
gci | sort -Property name
   ```

2. The `admin` environment variable should be near the top of the list of system variables. If it is not, create it by using the `New-Item` cmdlet. The `-Name` parameter has the value of `admin`, and the `-Value` parameter should be your given name. If this environment variable was created in the previous exercise, Windows PowerShell will report that it already exists. The command shown here allows you to re-create the `admin` environment variable.

   ```powershell
   New-Item -Name admin -Value mred
   ```

3. Use the `Rename-Item` cmdlet to rename the `admin` environment variable to `super`. The `-Path` parameter combines the PS drive name with the environment variable name, but it is not necessary if you are working on the Environment PS drive, as in this procedure. The `-NewName` parameter is the new name you want, without the PS drive specification. This command is shown here.

   ```powershell
   Rename-Item admin -NewName super
   ```

4. To verify that the old `admin` environment variable has been renamed to `super`, use the `Get-Item` cmdlet and specify the name `super`. You will not get the old name of the variable, but you will get the `super` variable with the same value as the previous `admin` variable. This command appears here.

   ```powershell
   Get-Item super
   ```

This concludes this procedure. Do not close 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.

   ```powershell
gci | sort -Property name
   ```

2. The `super` environment variable should be in the list of system variables. If it is not, create it by using the `New-Item` cmdlet. The `-Name` parameter has a value of `super`, and the `-Value` parameter should be your given name. If this environment variable was created in the
previous exercise, Windows PowerShell will report that it already exists. If you have deleted the *super* environment variable, the command shown here creates it.

```powershell
New-Item -Name super -Value mred
```

3. Use the `Remove-Item` cmdlet to remove the *super* environment variable. Enter the name of the item to be removed after the name of the cmdlet. If you are still in the Environment PS drive, you will not need to supply a `-Path` parameter value. The command is shown here.

```powershell
Remove-Item super
```

4. Use the `Get-ChildItem` cmdlet to verify that the *super* environment variable has been removed. To do this, press the Up Arrow key two or three times to retrieve the `gci | sort -property name` command. This command is shown here.

```powershell
gci | sort -property name
```

This concludes this procedure.

**Understanding the filesystem provider**

The filesystem provider is the easiest Windows PowerShell provider to understand—it provides access to the file system. When Windows PowerShell is started, it automatically opens on the user documents folder. By using the Windows PowerShell filesystem provider, you can create both directories and files. You can retrieve properties of files and directories, and you can also delete them. 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.

**Working with directory listings**

1. Open the Windows PowerShell console.

2. Use the `Get-ChildItem` cmdlet to obtain a directory listing of drive C. Use the `gci` alias to reduce typing. This is shown here.

   ```powershell
   GCI C:"
   ```

3. Use the Up Arrow key 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.

   ```powershell
   GCI C:\ | where psiscontainer
   ```

4. Use the Up Arrow key to retrieve the `gci C:\ | where psiscontainer` command, and use the exclamation point (!) (meaning *not*) to retrieve only items in the PS drive that are not directories.
The modified command is shown here. (The simplified Where-Object syntax does not support using the not operator directly on the input property.)

```
gci | where {!($psitem.psiscontainer)}
```

5. Now use the -Directory parameter to display only containers (directories). To do this, use the Get-ChildItem cmdlet but specify the -Directory parameter. This command is shown here.

```
gci -Directory
```

6. Now look for files. To do this, use the -File parameter. Use the Up Arrow key to retrieve the previous command, backspace to remove the -Directory parameter, and then add the -File parameter. The command appears here.

```
Gci -File
```

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

**Tip** Aliases in Windows PowerShell are not case sensitive. This makes them easy to type. Therefore, GCI, Gci, gci, and even GCi all work as aliases for Get-ChildItem.

### Identifying properties of directories

1. Use the Get-ChildItem cmdlet, and supply a value of C:\ for the -Path parameter. 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 piping it into a Where-Object cmdlet and specifying the membertype attribute as equal to property. To do this, use the Up Arrow key to retrieve the previous gci -Path C:\ | gm command. This time, remove the path parameter, because by default it operates against the current path. Pipeline the resulting object into the Where-Object cmdlet, and filter on the membertype attribute. The resulting command is shown here.

```
gci | gm | Where {{$_.membertype -eq "property"}}
```

3. You need to use the -Force parameter to view hidden files, and therefore to view both directory-info objects and fileinfo objects. Also, you do not need to pipeline Get-Member to the Where-Object cmdlet to filter on membertype, because the Get-Member cmdlet has a -MemberType parameter. Here is the revised command.

```
gci -Force | gm -MemberType Property
```
4. The preceding command returns information about both the `System.IO.DirectoryInfo` and `System.IO.FileInfo` objects. To reduce the output to only the properties associated with the `System.IO.FileInfo` object, you need to use the `-File` parameter. Use the Up Arrow key to retrieve the previous command. Add the `-File` parameter. The modified command is shown here.

```
gci -Force -File | gm -MemberType Property
```

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

```
TypeName: System.IO.FileInfo

<table>
<thead>
<tr>
<th>Name</th>
<th>MemberType</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>Property</td>
<td><code>System.IO.FileAttributes Attributes {get;set;}</code></td>
</tr>
<tr>
<td>CreationTime</td>
<td>Property</td>
<td><code>datetime CreationTime {get;set;}</code></td>
</tr>
<tr>
<td>CreationTimeUtc</td>
<td>Property</td>
<td><code>datetime CreationTimeUtc {get;set;}</code></td>
</tr>
<tr>
<td>Directory</td>
<td>Property</td>
<td><code>System.IO.DirectoryInfo Directory {get;}</code></td>
</tr>
<tr>
<td>DirectoryName</td>
<td>Property</td>
<td><code>string DirectoryName {get;}</code></td>
</tr>
<tr>
<td>Exists</td>
<td>Property</td>
<td><code>bool Exists {get;}</code></td>
</tr>
<tr>
<td>Extension</td>
<td>Property</td>
<td><code>string Extension {get;}</code></td>
</tr>
<tr>
<td>FullName</td>
<td>Property</td>
<td><code>string FullName {get;}</code></td>
</tr>
<tr>
<td>IsReadOnly</td>
<td>Property</td>
<td><code>bool IsReadOnly {get;set;}</code></td>
</tr>
<tr>
<td>LastAccessTime</td>
<td>Property</td>
<td><code>datetime LastAccessTime {get;set;}</code></td>
</tr>
<tr>
<td>LastAccessTimeUtc</td>
<td>Property</td>
<td><code>datetime LastAccessTimeUtc {get;set;}</code></td>
</tr>
<tr>
<td>LastWriteTime</td>
<td>Property</td>
<td><code>datetime LastWriteTime {get;set;}</code></td>
</tr>
<tr>
<td>LastWriteTimeUtc</td>
<td>Property</td>
<td><code>datetime LastWriteTimeUtc {get;set;}</code></td>
</tr>
<tr>
<td>Length</td>
<td>Property</td>
<td><code>long Length {get;}</code></td>
</tr>
<tr>
<td>Name</td>
<td>Property</td>
<td><code>string Name {get;}</code></td>
</tr>
</tbody>
</table>
```

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

Tip  Spacing in Windows PowerShell commands does not matter. Therefore, `gci|gm` works the same as `gci | gm`, and both commands will retrieve members of objects from whatever provider drive you happen to be working on.

Creating folders and files

1. Set your location to your temp folder. To do this, use the `temp` variable from the Environment PS drive. Use the `Get-Item` cmdlet to obtain a listing of files and folders. Use the `gi` alias and the `sl` alias to reduce typing. The commands and my output appear here.

```
PS C:\> sl $env:TEMP
PS C:\Users\ed\AppData\Local\Temp> gi *
```
2. Use the `New-Item` cmdlet to create a folder named `mytempfolder`. Use the `-Name` parameter to specify the name of `mytempfolder`, and use the `-ItemType` parameter to tell Windows PowerShell that the new item will be a directory. This command is shown here.

```
New-Item -Name mytempfolder -ItemType Directory
```

The resulting output, shown here, confirms the operation.

```
Directory: C:\Users\ed\AppData\Local\Temp

Mode LastWriteTime Length Name
---- ------------- ------ -----
-d---- 5/2/2015 3:14 PM                mytempfolder
```

3. Use the `New-Item` cmdlet to create an empty text file. To do this, use the Up Arrow key and retrieve the previous `New-Item -Name mytempfolder -ItemType Directory` command. Edit the `-Name` parameter to specify a text file named `mytempfile`, and specify the `-ItemType` parameter as `file`. The resulting command is shown here.

```
New-Item -Name mytempfile -ItemType File
```

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

```
Directory: C:\Users\ed\AppData\Local\Temp

Mode LastWriteTime Length Name
---- ------------- ------ -----
 -a---- 5/2/2015 3:18 PM              0 mytempfile
```

4. Use the `mkdir` function (`md` is an alias) to create a new folder with a temporary name. To do this, use the `GetRandomFileName` method from the `[io.path]` class. Here is the command.

```
md -path ([io.path]::GetRandomFileName())
```

The output from the command will appear similar to the output here.

```
Directory: C:\Users\ed\AppData\Local\Temp

Mode LastWriteTime Length Name
---- ------------- ------ -----
   d---- 5/2/2015 3:23 PM              hdsa2rys.xcs
```

From the Library of Todd Schultz
5. Now create a temporary file with a temporary file name. To do this, use the `New-Temporary-File` cmdlet. It is important when using this cmdlet to use a variable to store the name of the returned `FileInfo` object so that you will have access to the name of the file. This command and the output associated with the command are shown here.

```
PS C:\Users\ed\AppData\Local\Temp> $tempfile = New-TemporaryFile
PS C:\Users\ed\AppData\Local\Temp> $tempfile

Directory: C:\Users\ed\AppData\Local\Temp

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----         5/2/2015   3:26 PM              0 tmpEE3A.tmp
```

Tip  Capitalization of variable names does not matter. Therefore, Windows PowerShell interprets `$mytempfile` the same as `$MyTempFile`.

6. Delete the files and folders in the temp folder. To do this, pipeline the results of the `Get-ChildItem` cmdlet (`dir` is an alias) to the `Remove-Item` cmdlet with the `-Recurse` parameter. To prototype the command and view the files and folders that will be deleted, use the `-WhatIf` parameter. This command is shown here.

```
dir | ri -Recurse -WhatIf
dir | ri -Recurse
```

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

**Reading and writing for files**

1. Use the Up Arrow key to retrieve the `New-Item -Name mytempfile -ItemType File` command. Add the `-Value` parameter to the end of the command line and supply a value of `My file`. This command is shown here.

```
New-Item -Name mytempfile -ItemType File -Value "My file"
```

2. Use the `Get-Content` cmdlet to read the contents of `mytempfile`. This command is shown here.

```
Get-Content mytempfile
```

3. Use the `Add-Content` cmdlet to add additional information to the `mytempfile` file. This command is shown here.

```
Add-Content mytempfile -Value "ADDITIONAL INFORMATION"
```

84  Windows PowerShell Step by Step, Third Edition
4. Press the Up Arrow key twice and retrieve the `Get-Content mytempfile` command, which is shown here.

   Get-Content mytempfile

   The output from the `Get-Content mytempfile` command is shown here.

   My file

5. Press the Up Arrow key twice, and retrieve the `Add-Content mytempfile -Value "ADDITIONAL INFORMATION"` command to add additional information to the file. This command is shown here.

   Add-Content mytempfile -Value "ADDITIONAL INFORMATION"

6. Use the Up Arrow key to retrieve the `Get-Content mytempfile` command, which is shown here.

   Get-Content mytempfile

   The output produced is shown here. Notice that the second time the command runs, the “ADDITIONAL INFORMATION” string is added to a new line in the original file.

   My file
   ADDITIONAL INFORMATION
   ADDITIONAL INFORMATION

7. Use the `Set-Content` cmdlet to overwrite the contents of the `mytempfile` file. Specify the `-Value` parameter as `Setting information`. This command is shown here.

   Set-Content mytempfile -Value "Setting information"

8. Use the Up Arrow key to retrieve the `Get-Content mytempfile` command, which is shown here.

   Get-Content mytempfile

   The output from the `Get-Content` command is shown here.

   Setting information

   This concludes this procedure.

Tip  File names are not case sensitive. Therefore, Windows PowerShell does not distinguish between `MyFile.Txt` and `myfile.txt`.

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 described earlier apply to working with functions.
### Listing all functions on the system

1. Open the Windows PowerShell console.

2. Use the `Set-Location` cmdlet to change the working location to the Function PS drive. This command is shown here.

   ```powershell
   Set-Location function:
   ```

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

   ```powershell
   gci
   ```

4. The resulting list contains many functions, such as functions that use `Set-Location` to change the current location to different drive letters (for example, `C:`). A partial view of this output is shown here.

<table>
<thead>
<tr>
<th>CommandType</th>
<th>Name</th>
<th>Version</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>A:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>B:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>C:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>cd..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>cd\</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Clear-Host</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>G:</td>
<td>3.1.0.0</td>
<td>Mic...</td>
</tr>
<tr>
<td>Function</td>
<td>Get-FileHash</td>
<td>3.1.0.0</td>
<td>Mic...</td>
</tr>
<tr>
<td>Function</td>
<td>Get-SerializedCommand</td>
<td>3.1.0.0</td>
<td>Mic...</td>
</tr>
<tr>
<td>Function</td>
<td>Get-Verb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>H:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>help</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>I:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>ImportSystemModules</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>mkdir</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>more</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>N:</td>
<td>3.1.0.0</td>
<td>Mic...</td>
</tr>
<tr>
<td>Function</td>
<td>New-Guid</td>
<td>3.1.0.0</td>
<td>Mic...</td>
</tr>
<tr>
<td>Function</td>
<td>New-TemporaryFile</td>
<td>3.1.0.0</td>
<td>Mic...</td>
</tr>
<tr>
<td>Function</td>
<td>O:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>oss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>P:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Pause</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>prompt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Q:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>TabExpansion2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the Library of Todd Schultz
5. To return only the functions that are used for drives, use the `Get-ChildItem` cmdlet and the `Where` method to filter definitions that contain the word `set`. Use the `-match` operator to perform the matching. The resulting command is shown here.

```
(Get-ChildItem).where({$psitem.definition -match 'set'})
```

6. If you are more interested in functions that are not related to drive mappings, you can use the `-notmatch` operator instead of `-match`. The easiest way to make this change is to use the Up Arrow key to retrieve the previous command and change the operator from `-match` to `-notmatch`. The resulting command is shown here.

```
(Get-ChildItem).where({$psitem.definition -notmatch 'set'})
```

The resulting listing of functions is shown here.

<table>
<thead>
<tr>
<th>CommandType</th>
<th>Name</th>
<th>Version</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Get-Verb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>ImportSystemModules</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>more</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>New-Guid</td>
<td>3.1.0.0</td>
<td>Mic...</td>
</tr>
<tr>
<td>Function</td>
<td>New-TemporaryFile</td>
<td>3.1.0.0</td>
<td>Mic...</td>
</tr>
<tr>
<td>Function</td>
<td>oss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Pause</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>prompt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7. Use the `Get-Content` cmdlet to retrieve the text of the `pause` function. This is shown here (`gc` is an alias for the `Get-Content` cmdlet).

```
gc pause
```

The content of the `pause` function is shown here.

```
Read-Host 'Press Enter to continue...' | Out-Null
```

This concludes this procedure.

### Using the registry provider to manage the Windows registry

In Windows PowerShell 1.0, the registry provider made it easy to work with the registry on the local system. Unfortunately, if you were not using remoting, you were limited to working with the local computer or using some other remoting mechanism (such as a logon script) to make changes on remote systems. Beginning with Windows PowerShell 2.0, the inclusion of remoting makes it possible to make remote registry changes as easily as changing the local registry. In Windows PowerShell 3.0, the registry provider was further improved with the introduction of transactions. In Windows PowerShell 4.0 and Windows PowerShell 5.0, there have been no major increases in functionality.
You can use the registry provider to access the registry in the same manner that the filesystem provider permits access to a local disk drive. The same cmdlets used to access the file system—New-Item, Get-ChildItem, Set-Item, Remove-Item, and the rest—also work with the registry.

The two registry drives

By default, the registry provider creates two registry drives. To find all of the drives exposed by the registry provider, use the Get-PSDrive cmdlet. These drives are shown here.

```powershell
Get-PSDrive -PSProvider registry | select name, root
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>HKCU</td>
<td>HKEY_CURRENT_USER</td>
</tr>
<tr>
<td>HKLM</td>
<td>HKEY_LOCAL_MACHINE</td>
</tr>
</tbody>
</table>

You can create additional registry drives by using the New-PSDrive cmdlet. For example, it is common to create a registry drive for the HKEY_CLASSES_ROOT registry hive. The code to do this is shown here.

```powershell
New-PSDrive -PSProvider Registry -Root HKEY_CLASSES_ROOT -Name HKCR
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Used (GB)</th>
<th>Free (GB)</th>
<th>Provider</th>
<th>Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>HKCR</td>
<td></td>
<td></td>
<td>Registry</td>
<td>HKEY_CLASSES_ROOT</td>
</tr>
</tbody>
</table>

After it is created, the new HKCR drive is accessible in the same way as any other drive. For example, to change the working location to the HKCR drive, use either the Set-Location cmdlet or one of its aliases (such as cd). This technique is shown here.

```powershell
Set-Location HKCR:
```

To determine the current location, use the Get-Location cmdlet. This technique is shown here.

```powershell
Get-Location
```

<table>
<thead>
<tr>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>HKCR:</td>
</tr>
</tbody>
</table>

After you've set the new working location, explore it by using the Get-ChildItem cmdlet (or one of the aliases for that cmdlet, such as dir). This technique is shown in Figure 3-6.
Retrieving registry values

To view the values stored in a registry key, use either the Get-Item or the Get-ItemProperty cmdlet. Using the Get-Item cmdlet reveals that there is one property (named default). This is shown here.

```powershell
PS HKCR:\> Get-Item .\.ps1 | fl *
```

```
Property      : {(default)}
PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_CLASSES_ROOT\.ps1
PSParentPath  : Microsoft.PowerShell.Core\Registry::HKEY_CLASSES_ROOT
PSChildName   : .ps1
PSDrive       : HKCR
PSProvider    : Microsoft.PowerShell.Core\Registry
PSIsContainer : True
SubKeyCount   : 1
View          : Default
Handle        : Microsoft.Win32.SafeHandles.SafeRegistryHandle
ValueCount    : 1
Name          : HKEY_CLASSES_ROOT\.ps1
```
To access the value of the *default* property, you must use the *Get-ItemProperty* cmdlet, as shown here.

```powershell
PS HKCR:\> Get-ItemProperty .\..ps1 | fl *
```

```
(default)    : Microsoft.PowerShellScript.1
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CLASSES_ROOT\..ps1
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CLASSES_ROOT
PSChildName  : .ps1
PSDrive      : HKCR
PSPortier     : Microsoft.PowerShell.Core\Registry
```

The technique for accessing registry keys and the values associated with them is shown in Figure 3-7.

![Figure 3-7](image.png)

**FIGURE 3-7** Use the *Get-ItemProperty* cmdlet to access registry property values.

Returning only the value of the *default* property requires a bit of manipulation. The *default* property requires the use of literal quotation marks to force the evaluation of the parentheses in the name. This is shown here.

```powershell
PS HKCR:\> (Get-ItemProperty .\..ps1 -Name '(default)').'(default)'
```

```
Microsoft.PowerShellScript.1
```

The registry provider provides a consistent and easy way to work with the registry from within Windows PowerShell. By 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.
Two PS drives are created by default. To identify the PS drives 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 matches the word `registry`. This command is shown here.

```
PS C:\> Get-PSDrive | ? provider -match registry
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Used (GB)</th>
<th>Free (GB)</th>
<th>Provider</th>
<th>Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>HKCR</td>
<td></td>
<td></td>
<td>Registry</td>
<td>HKEY_CLASSES_ROOT</td>
</tr>
<tr>
<td>HKCU</td>
<td></td>
<td></td>
<td>Registry</td>
<td>HKEY_CURRENT_USER</td>
</tr>
<tr>
<td>HKLM</td>
<td></td>
<td></td>
<td>Registry</td>
<td>HKEY_LOCAL_MACHINE</td>
</tr>
</tbody>
</table>

### Obtaining a listing of registry keys

1. Open the Windows PowerShell console.
2. Use the `Get-ChildItem` cmdlet and supply `HKLM:\software` as the value for the `-Path` parameter. 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
```

The corresponding keys, as displayed in Regedit.exe, are shown in Figure 3-8. A partial listing of example output is shown after the figure.

![Regedit.exe view of HKEY_LOCAL_MACHINE\SOFTWARE](image)

**FIGURE 3-8** The Registry Editor tool shows a Regedit.exe view of HKEY_LOCAL_MACHINE\SOFTWARE.
Hive: HKEY_LOCAL_MACHINE\SOFTWARE

<table>
<thead>
<tr>
<th>Name</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td></td>
</tr>
<tr>
<td>Clients</td>
<td></td>
</tr>
<tr>
<td>Intel</td>
<td></td>
</tr>
<tr>
<td>Macromedia</td>
<td></td>
</tr>
<tr>
<td>Microsoft</td>
<td></td>
</tr>
<tr>
<td>ODBC</td>
<td></td>
</tr>
<tr>
<td>OEM</td>
<td></td>
</tr>
<tr>
<td>Partner</td>
<td></td>
</tr>
<tr>
<td>Policies</td>
<td></td>
</tr>
<tr>
<td>RegisteredApplications</td>
<td>Paint : SOFTWARE\Microsoft\Windows\CurrentVersion\Applets\Paint\Capabilities</td>
</tr>
<tr>
<td></td>
<td>Windows Search : Software\Microsoft\Windows Search\Capabilities</td>
</tr>
<tr>
<td></td>
<td>Windows Disc Image Burner : Software\Microsoft\IsoBurn\Capabilities</td>
</tr>
<tr>
<td></td>
<td>Windows File Explorer : SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Capabilities</td>
</tr>
<tr>
<td></td>
<td>Windows Photo Viewer : Software\Microsoft\Windows Photo Viewer\Capabilities</td>
</tr>
<tr>
<td></td>
<td>Wordpad : Software\Microsoft\Windows\CurrentVersion\Applets\Wordpad\Capabilities</td>
</tr>
<tr>
<td></td>
<td>Windows Media Player : Software\Clients\Media\Windows Media Player\Capabilities</td>
</tr>
<tr>
<td></td>
<td>Internet Explorer : SOFTWARE\Microsoft\Internet Explorer\Capabilities</td>
</tr>
<tr>
<td></td>
<td>Windows Address Book : Software\Clients\Contacts\Address Book\Capabilities</td>
</tr>
</tbody>
</table>

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

**Searching for software**

1. Use the `Get-ChildItem` cmdlet and supply a value for the `-Path` parameter. Use the HKLM:\PS drive and supply a path of `SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall`. To make the command easier to read, use single quotation marks (') to encase the string. You can use tab completion to assist with the typing. The completed command is shown here.

   `gci -Path 'HKLM:SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall'`
The resulting listing of software is shown in the output here, in abbreviated fashion.

Hive: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall

<table>
<thead>
<tr>
<th>Name</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressBook</td>
<td></td>
</tr>
<tr>
<td>CNXT_AUDIO_HDA</td>
<td>DisplayName     : Conexant 20672 SmartAudio HD</td>
</tr>
<tr>
<td></td>
<td>DisplayVersion : 8.32.23.2</td>
</tr>
<tr>
<td></td>
<td>VersionMajor  : 8</td>
</tr>
<tr>
<td></td>
<td>VersionMinor  : 0</td>
</tr>
<tr>
<td></td>
<td>Publisher     : Conexant</td>
</tr>
<tr>
<td></td>
<td>DisplayIcon   : C:\Program Files\CONEXANT\CNXT_AUDIO_HDA\UIU64a.exe</td>
</tr>
<tr>
<td></td>
<td>UninstallString : C:\Program Files\CONEXANT\CNXT_AUDIO_HDA\UIU64a.exe -U -G -Ichdrt.inf</td>
</tr>
<tr>
<td>Connection Manager</td>
<td>SystemComponent : 1</td>
</tr>
<tr>
<td>DirectDrawEx</td>
<td></td>
</tr>
<tr>
<td>DXM_Runtime</td>
<td></td>
</tr>
<tr>
<td>Fontcore</td>
<td></td>
</tr>
<tr>
<td>IE40</td>
<td></td>
</tr>
<tr>
<td>IE4Data</td>
<td></td>
</tr>
<tr>
<td>IEBAKEX</td>
<td></td>
</tr>
<tr>
<td>IEData</td>
<td></td>
</tr>
<tr>
<td>MobileOptionPack</td>
<td></td>
</tr>
<tr>
<td>MPlayer2</td>
<td></td>
</tr>
<tr>
<td>Office15.PROPLUS</td>
<td>Publisher                : Microsoft Corporation</td>
</tr>
<tr>
<td></td>
<td>CacheLocation            : C:\MSOCache\All Users</td>
</tr>
<tr>
<td></td>
<td>DisplayIcon              : C:\Program Files\Common</td>
</tr>
</tbody>
</table>

2. To retrieve information on a single software package, you will need to add a *Where-Object* cmdlet. You can do this by using the Up Arrow key to retrieve the previous `gci -Path 'HKLM:SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall'` command and pipelining the resulting object into the *Where-Object* cmdlet. Supply a value for the *name* property, as shown in the code listed here. Alternatively, supply a name from the previous output.

```
PS C:\> gci -path 'HKLM:SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall' | where name -match 'office'
```

This concludes this procedure.

Creating new registry keys

Creating a new registry key by using Windows PowerShell is the same as creating a new file or a new folder—all three processes use the *New-Item* cmdlet. In addition to using the *New-Item* cmdlet, you might use the *Test-Path* cmdlet to determine whether the registry key already exists. You might also want to change your working location to one of the registry drives. If you do this, you might use the *Push-Location*, *Set-Location*, and *Pop-Location* cmdlets. This is, of course, the long way of doing things. These steps appear next.
**Caution** The registry contains information vital to the operation and configuration of your computer. Serious problems could arise if you edit the registry incorrectly. Therefore, it is important to back up your system prior to attempting to make any changes. For information about backing up your registry, see Microsoft TechNet article KB322756. For general information about working with the registry, see Microsoft TechNet article KB310516.

1. Store the current working location by using the `Push-Location` cmdlet.
2. Change the current working location to the appropriate registry drive by using the `Set-Location` cmdlet.
3. Use the `Test-Path` cmdlet to determine whether the registry key already exists.
4. Use the `New-Item` cmdlet to create the new registry key.
5. Use the `Pop-Location` cmdlet to return to the starting working location.

The following example creates a new registry key named `test` off the HKEY_CURRENT_USERS software registry hive. It illustrates each of the five steps detailed previously.

```
Push-Location
Set-Location HKCU:
Test-Path .\Software\test
New-Item -Path .\Software -Name test
Pop-Location
```

The commands and the associated output from the commands appear in Figure 3-9.

![Figure 3-9](image.png)

**FIGURE 3-9** Create a new registry key by using the `New-Item` cmdlet.
The short way to create a new registry key

It is not always necessary to change the working location to a registry drive when creating a new registry key. In fact, it is not even necessary to use the `Test-Path` cmdlet to determine whether the registry key exists. If the registry key already exists, an error is generated. If you want to overwrite the registry key, use the `-Force` parameter. This technique works for all the Windows PowerShell providers, not just for the registry provider.

Note How to deal with an already existing registry key is one of those design decisions that confront IT professionals who venture far into the world of scripting. Software developers are very familiar with these types of decisions and usually deal with them in the analyzing-requirements portion of the development life cycle. IT professionals who open the Windows PowerShell ISE first and think about the design requirements second can easily become stymied, and possibly write in problems. For more information about this, see my book, *Windows PowerShell Best Practices* (Microsoft Press, 2014).

The following example creates a new registry key named `test` in the HKCU:\SOFTWARE location. Because the command includes the full path, it does not need to execute from the HKCU drive. Because the command uses the `-Force` switched parameter, the command overwrites the HKCU:\SOFTWARE\TEST registry key if it already exists.

```
New-Item -Path HKCU:\Software -Name test -Force
```

Note To watch the `New-Item` cmdlet in action when using the `-Force` switched parameter, use the `-Verbose` switched parameter. The command appears here.

```
New-Item -Path HKCU:\Software -Name test -Force -Verbose
```

The steps for creating a registry key are as follows:

1. Include the full path to the registry key that you want to create.
2. Use the `-Force` parameter to overwrite any existing registry key of the same name.

In Figure 3-10, the first attempt to create a test registry key fails because the key already exists. The second command uses the `-Force` parameter, causing the command to overwrite the existing registry key, and therefore it succeeds without creating an error.
FIGURE 3-10 Use the -Force parameter when creating a new registry key, to overwrite the key if it already exists.

Setting the default value for the key

The previous examples do not set the default value for the newly created registry key. If the registry key already exists (as it does in this specific case), you can use the Set-Item cmdlet to assign a default value to the registry key. The steps to accomplish this are detailed here:

1. Use the Set-Item cmdlet, and supply the complete path to the existing registry key.
2. Supply the default value in the -Value parameter of the Set-Item cmdlet.

The following command assigns the value test key to the default property value of the test registry key contained in the HKCU\SOFTWARE location.

```powershell
Set-Item -Path HKCU:\Software\test -Value "test key"
```

Using New-Item to create and assign a value

It is not necessary to use the New-Item cmdlet to create a registry key and then use the Set-Item cmdlet to assign a default value. You can combine these steps into a single command. The following command creates a new registry key with the name of HSG1 and assigns a default value of default value to the registry key.

```powershell
New-Item -Path HKCU:\Software\hsg1 -Value "default value"
```
Modifying the value of a registry property value

Modifying the value of a registry property value requires the use of the `Set-ItemProperty` cmdlet.

1. Use the `Push-Location` cmdlet to save the current working location.
2. Use the `Set-Location` cmdlet to change to the appropriate registry drive.
3. Use the `Set-ItemProperty` cmdlet to assign a new value to the registry property.
4. Use the `Pop-Location` cmdlet to return to the original working location.

When you know that a registry property value exists, the solution is simple: you use the `Set-ItemProperty` cmdlet and assign a new value. The code that follows saves the current working location, changes the new working location to the registry key, uses the `Set-ItemProperty` cmdlet to assign a new value, and then uses the `Pop-Location` cmdlet to return to the original working location.

```
Note The code that follows relies upon positional parameters for the `Set-ItemProperty` cmdlet. The first parameter is `-Path`. Because the `Set-Location` cmdlet sets the working location to the registry key, a period identifies the path as the current directory. The second parameter is the name of the registry property to change—in this example, it is `newproperty`. The last parameter is `-Value`, and that defines the value to assign to the registry property. In this example, it is `mynewvalue`. The command with complete parameter names would thus be `Set-ItemProperty -Path . newproperty` `-Value mynewvalue`. The quotation marks in the following code are not required, but do not harm anything either.
```

```
PS C:\> Push-Location
PS C:\> Set-Location HKCU:\Software\test
PS HKCU:\Software\test> Set-ItemProperty . newproperty "mynewvalue"
PS HKCU:\Software\test> Pop-Location
PS C:\>

Of course, all the pushing, popping, and setting of locations is not really required. It is entirely possible to change the registry property value from any location within the Windows PowerShell provider subsystem.

The short way to change a registry property value

To change a registry property value easily, use the `Set-ItemProperty` cmdlet to assign a new value. Ensure that you specify the complete path to the registry key. Here is an example of using the `Set-ItemProperty` cmdlet to change a registry property value without first navigating to the registry drive.

```
PS C:\> Set-ItemProperty -Path HKCU:\Software\test -Name newproperty -Value anewvalue
```
Dealing with a missing registry property

If you need to set a registry property value, you can set the value of that property easily by using the `Set-ItemProperty` cmdlet. But what if the registry property does not exist? How do you set the property value then? You can still use the `Set-ItemProperty` cmdlet to set a registry property value, even if the registry property does not exist, as shown in the following code.

```
Set-ItemProperty -Path HKCU:\Software\test -Name missingproperty -Value avalue
```

To determine whether a registry key exists, you can use the `Test-Path` cmdlet. It returns `true` if the key exists and `false` if it does not exist. This technique is shown here.

```
PS C:\> Test-Path HKCU:\Software\test
True
PS C:\> Test-Path HKCU:\Software\test\newproperty
False
```

Unfortunately, this technique does not work for a registry key property. It always returns `false`—even if the registry property exists. This is shown here.

```
PS C:\> Test-Path HKCU:\Software\test\newproperty
False
PS C:\> Test-Path HKCU:\Software\test\bogus
False
```

Therefore, if you do not want to overwrite a registry key property if it already exists, you need a way to determine whether the registry key property exists—and using the `Test-Path` cmdlet does not work. The following procedure shows how to handle this.

### Testing for a registry key property prior to writing a new value

1. Use the `if` statement and the `Get-ItemProperty` cmdlet to retrieve the value of the registry key property. Specify the `-ErrorAction (ea is an alias) parameter of `SilentlyContinue` (0 is the enumeration value associated with `SilentlyContinue`).

2. In the script block for the `if` statement, display a message that the registry property exists, or just exit.

3. In the `else` statement, call `Set-ItemProperty` to create and set the value of the registry key property.

This technique is shown here.

```
if(((Get-ItemProperty -Path HKCU:\Software\test -Name bogus -ea 0).bogus)
{'Property already exists'}
ELSE { Set-ItemProperty -Path HKCU:\Software\test -Name bogus -Value 'initial value'}
```
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 `*variable` as a value for the `-Name` parameter. In the following example, the `-Name` parameter is positional, in the first position, and is omitted. To return only cmdlets, you 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 "cmdlet"
```

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 key and pipeline the returned object into the `Format-List` cmdlet. Add the three properties you 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
```

**Note** You will get this output from Windows PowerShell only if you have run the `Update-Help` cmdlet.

The resulting output is much easier to read and understand; it is shown here.

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

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

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

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

Name     : Set-Variable
Category : Cmdlet
Synopsis : Sets the value of a variable. Creates the variable if one...
```
Working with variables

1. Open the Windows PowerShell console.

2. Use the `Set-Location` cmdlet to set the working location to the Variable PS drive. Use the `sl` alias to reduce the need for typing. 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. Press the Up Arrow key 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.

   \[ Get-ChildItem | Sort-Object Name \]

The output from the previous command is shown here:

```
Name                           Value
----                           ----- 
$                              name
?                              True
^ alias
alias:                        alias:
alias:\                       args {}
args:                         BufferSize 85,3000
BufferSize                    ConfirmPreference High
ConsoleFileName               DebugPreference SilentlyContinue
ErrorView                     NormalView
false                         False
FormatEnumerationLimit        4
HOST                          C:\Users\ed
HOME                          C:\Users\ed
Host                          System.Management.Automation.Internal.Host.Interna...
InformationPreference        Continue
input                         System.Collections.ArrayList<ArrayListEnumeratorSi...
LASTEXITCODE                  0
Matches                       {0}
MaximumAliasCount            4096
MaximumDriveCount            4096
MaximumErrorCode             256
MaximumFunctionCount         4096
MaximumHistoryCount          4096
```

From the Library of Todd Schultz
5. 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

6. Press the Up Arrow key 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 *

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

PSPath : Microsoft.PowerShell.Core\Variable::shellid
PSDrive : Variable
PSProvider : Microsoft.PowerShell.Core\Variable
PSIsContainer : False
Name : ShellId
Description : The ShellID identifies the current shell. This is used by #Requires.
Value : Microsoft.PowerShell
Visibility : Public
Module :
ModuleName : Options : Constant, AllScope
Attributes : {}
7. Create a new variable called *administrator*. To do this, use the *New-Variable* cmdlet. This command is shown here.

```
New-Variable administrator
```

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

```
Get-Variable administrator
```

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>administrator</td>
<td></td>
</tr>
</tbody>
</table>

9. 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
```

10. Press the Up Arrow key two times to retrieve the previous *Get-Variable administrator* command. This command is shown here.

```
Get-Variable administrator
```

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>administrator</td>
<td>mred</td>
</tr>
</tbody>
</table>

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

```
Remove-Variable administrator
```

You could also use the *Del* alias while on the Variable drive, as shown here.

```
del .\administrator
```

12. Press the Up Arrow key two times to retrieve the previous *Get-Variable administrator* command. This command is shown here.

```
Get-Variable administrator
```
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

This concludes this procedure.

Exploring Windows PowerShell providers: Step-by-step exercises

In this exercise, you’ll explore the use of the certificate provider in Windows PowerShell. You will navigate the certificate provider by using the same types of commands used with the file system. You will then explore the environment provider by using the same methodology.

Exploring the certificate provider

1. Open the Windows PowerShell console.

2. Obtain a listing of all the properties available for use with the Get-ChildItem cmdlet by pipelining 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*"

   The results of this command are shown here.

<table>
<thead>
<tr>
<th>Name</th>
<th>Used (GB)</th>
<th>Free (GB)</th>
<th>Provider</th>
<th>Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>110.38</td>
<td>38.33</td>
<td>FileSystem</td>
<td>C:\</td>
</tr>
<tr>
<td>Cert</td>
<td></td>
<td></td>
<td>Certificate</td>
<td>\</td>
</tr>
</tbody>
</table>

4. Use the Set-Location cmdlet to change to the Certificate drive.

   sl cert:

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

   GCI
The output from the previous command is shown here.

Location : CurrentUser
StoreNames : {SmartCardRoot, Root, Trust, AuthRoot...}

Location : LocalMachine
StoreNames : {TrustedPublisher, ClientAuthIssuer, Remote Desktop, Root...}

6. 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 key. The command is shown here.

   GCI -Recurse

7. 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, pipeline 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

This concludes this step-by-step exercise.

In the following exercise, you’ll work with the Windows PowerShell environment provider.

### Examining the environment provider

1. Open the Windows PowerShell console.

2. Use the `New-PSDrive` cmdlet to create a drive mapping to the alias provider. The name of the new PS drive will be `al`. The `-PSProvider` parameter 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 PS drive 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. Press the Up Arrow key 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 begins with a letter after `t` in the alphabet. 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"
```

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

```
SL C:\
```

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

```
Remove-PSDrive al
```

8. Use the Get-PSDrive cmdlet to confirm that the al drive has been 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 parameter and supply env:\ as the value. This is shown here.

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

10. Press the Up Arrow key 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
```

The results from the previous command are shown here.

```
TypeName: System.Collections.Generic.Dictionary'2+ValueCollection[[System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.Collections.DictionaryEntry, mscorlib, Version=2.0.0.0, Culture=neutral, 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+ValueCollection... 
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;}
```

```
From the Library of Todd Schultz
11. Press the Up Arrow key twice to return to the `Get-Item -Path env:` command. Use the Home key to move your cursor 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.

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

12. 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.

```powershell
$objEnv.Count
```

13. Examine the methods of the object returned by `Get-Item -Path env:`. Notice that there is a `Get_Count` method. Let's use that method. The code is shown here.

```powershell
$objEnv.Get_count
```

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

```
OverloadDefinitions
-------------------
  int get_Count()
  int ICollection[DictionaryEntry].get_Count()
  int ICollection.get_Count()
```

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

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

15. If you want to know exactly what type of object is contained in the `$objEnv` variable, you can use the `GetType` method, as shown here.

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

This command returns the results shown here.

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

This concludes this exercise.
# Chapter 3 quick reference

<table>
<thead>
<tr>
<th>To</th>
<th>Do this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produce a listing of all variables defined in a Windows PowerShell session</td>
<td>Use the <code>cmdlet to change location to the Variable PS drive, and then use the </code>Get-ChildItem cmdlet.</td>
</tr>
<tr>
<td>Obtain a listing of all the aliases</td>
<td>Use the <code>Set-Location cmdlet to change location to the Alias PS drive, and then use the </code>Get-ChildItem cmdlet to produce a listing of aliases. Pipeline the resulting object into the <code>Where-Object cmdlet and filter on the </code>name property for the appropriate value.</td>
</tr>
<tr>
<td>Delete a directory that is empty</td>
<td>Use the `Remove-Item cmdlet and supply the name of the directory.</td>
</tr>
<tr>
<td>Delete a directory that contains other items</td>
<td>Use the <code>Remove-Item cmdlet, supply the name of the directory, and specify the </code>-Recurse switch parameter.</td>
</tr>
<tr>
<td>Create a new text file</td>
<td>Use the <code>New-Item cmdlet and specify the </code>-Path parameter for the directory location. Supply the <code>-Name parameter, and specify the </code>-ItemType parameter as <code>file. Example: </code>New-Item -Path C:\Mytest -Name Myfile.txt -ItemType file</td>
</tr>
<tr>
<td>Obtain a listing of registry keys from a registry hive</td>
<td>Use the <code>Get-ChildItem cmdlet and specify the </code>-Path parameter for the appropriate PS drive name. Complete the path with the appropriate registry path. Example: `gci -Path HKLM:\software</td>
</tr>
<tr>
<td>Obtain a listing of all functions on the system</td>
<td>Use the <code>Get-ChildItem cmdlet and supply the PS drive name of </code>function: to the <code>-Path Parameter. Example: </code>gci -Path function:`</td>
</tr>
</tbody>
</table>