CHAPTER 3

Advanced Programming

So far, we have just scratched the surface of what PowerShell can do—many books stop there. This can leave the reader with the impression that PowerShell is fairly limited as a development platform. This chapter will dispel that notion by discussing a wide variety of advanced programming techniques. We'll start by discussing the use of parameters to support code reusability. This leads into a discussion of the `CmdletBinding` attribute, which causes PowerShell to add a number of powerful features to the code, including parameter validation, parameter sets, and support for PowerShell's common parameters. We will then get into a discussion on creating functions so as to maximize reusability. From there we will move on to a discussion of creating custom objects in PowerShell complete with properties and methods. The value of custom objects will be demonstrated by creating a custom object to support ETL. To fully leverage PowerShell we need to understand how to use the pipeline. We will discuss how to write functions that do this with the special process blocks `begin`, `process`, and `end`. PowerShell's built-in support for customizing output is limited, so we will discuss how we can create highly customized output by using a function that generates HTML. Windows has built-in support for application automation. We will review an example of leveraging this to load an Excel spreadsheet with data from SQL Server. Although we can use the SQLPS module to access SQL Server, this adds unnecessary overhead, so we will show how we can write code to directly query SQL Server tables using the .Net library. There are times when we need to know when something happens to files on the network. For example, when a file is created, we may need to run an ETL job to load it. We will discuss trapping such events in order to execute custom code using the .Net `FileSystemWatcher` object. Hopefully, by the time you finish this chapter you will be convinced of the wide-ranging capabilities of PowerShell.

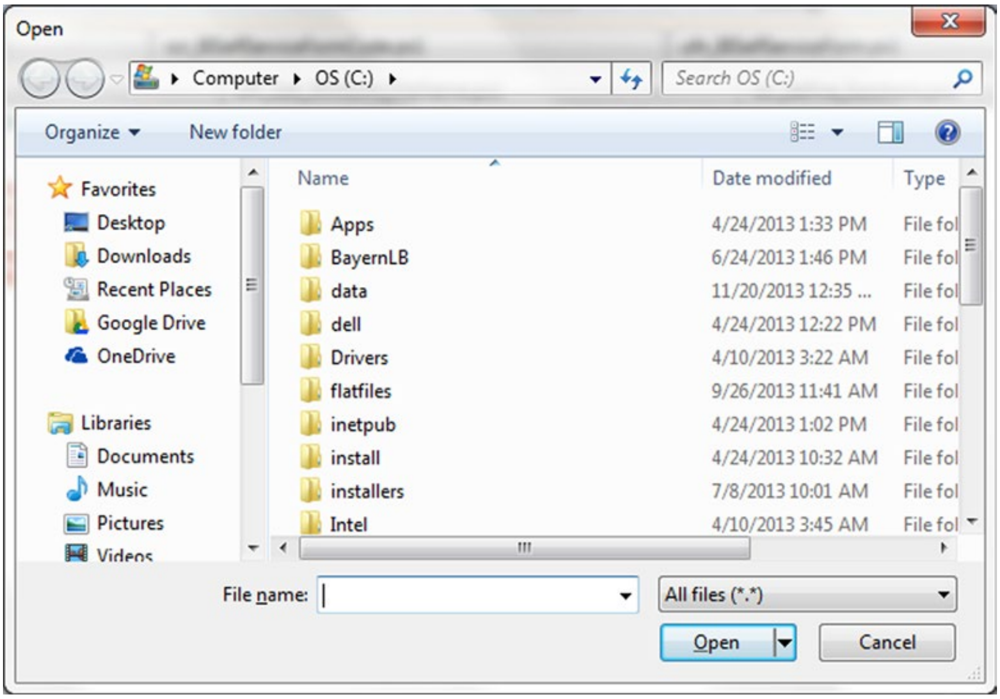
Passing Parameters

The key to code reusability is the ability to pass in values that modify the code's behavior. If a piece of code displays a window but all the properties of the window are assigned as constants, this is known as hard coding, and the related code can only be used to do a very specific task. When a block of code can be called and values passed to it that customize its process, the code can be reused to fill different needs at different times.

Values passed to a block of code are called *parameters*, and most programming languages, including C#, VB.Net, and T-SQL, support parameters. PowerShell not only supports parameters, but it also has a number of advanced features that enhance their implementation and use.

When we think of using parameters, we might tend to think of code organized into functions. Functions are blocks of code given a name and possibly a set of parameters, but the function only executes when it is called. If you run the function definition without any code outside the function to call it, the function is simply created in memory. If you've written stored procedures before, know that it's similar to that. Running the code that defines the stored procedure just creates the stored procedure object and saves it to the database catalog. To execute it, you need to use the T-SQL `EXEC` statement. Sometimes we just want to run a series of commands without going to the trouble of creating a function. We can think of this code as fall-through code, i.e., execution starts at the first line and just falls through, executing each line. Consider the code below that will display the Windows file-selection dialog shown in Figure 3-1.

```
[System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | Out-Null

$OpenFileDialog = New-Object System.Windows.Forms.OpenFileDialog
$OpenFileDialog.initialDirectory = "C:\"
$OpenFileDialog.filter = "All files (*.*)| *.*"
$OpenFileDialog.ShowDialog() | Out-Null
$OpenFileDialog.filename
```



Name	Date modified	Type
Apps	4/24/2013 1:33 PM	File fol
BayernLB	6/24/2013 1:46 PM	File fol
data	11/20/2013 12:35 ...	File fol
dell	4/24/2013 12:22 PM	File fol
Drivers	4/10/2013 3:22 AM	File fol
flatfiles	9/26/2013 11:41 AM	File fol
inetpub	4/24/2013 1:02 PM	File fol
install	4/24/2013 10:32 AM	File fol
installers	7/8/2013 10:01 AM	File fol
Intel	4/10/2013 3:45 AM	File fol

Figure 3-1. *The Open File dialog*

Note: Windows has a number of pre-built, ready-to-use dialogs called Common Dialogs that we can easily use from PowerShell. These include the Open File, Save File, Font Selection, and Color Chooser dialogs.

Displaying a file-selection dialog is a useful thing, but the code to do it is difficult to remember—and do we really want to have to cut and paste a block of code like that every time we need it? Let's rewrite this as a script that takes parameters. Parameters are passed via the $args array, which is automatically created by PowerShell. Consider the following code:

```
[System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | Out-Null

$OpenFileDialog = New-Object System.Windows.Forms.OpenFileDialog
$OpenFileDialog.initialDirectory = $args[0]
$OpenFileDialog.filter = $args[1]
$OpenFileDialog.ShowDialog() | Out-Null
$OpenFileDialog.filename
```

This is the crudest yet simplest way to code for script parameters. `$args` is an automatically generated array that contains anything after the name of the script on the command line. Because the script now expects parameters, we need to save the script and call it. There is a script in the book's code files called `scr_openfiledlg.ps1`. Please copy it to your Documents folder and then call it with the lines below.

```
set-location ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\")

./scr_openfiledlg.ps1 "C:\" "All files (*.*)| *.*"
```

Note: This can be done from the ISE or from the CLI.

The first line above sets the current folder location to the current user's Documents folder. The second line calls the script we saved. Now we have a bit of reusable code. However, we can improve upon this by naming the parameters, as shown below.

```
Param(
  [string]$initdir,
  [string]$filter
 )

 [System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | Out-Null

 $OpenFileDialog = New-Object System.Windows.Forms.OpenFileDialog
 $OpenFileDialog.initialDirectory = $initdir
 $OpenFileDialog.filter = $filter
 $OpenFileDialog.ShowDialog() | Out-Null
 $OpenFileDialog.filename
```

The above version of the script is called `scr_openfiledlg_with_parms.ps1` in the book's code files. Please copy it to your Documents folder and call it with the lines below.

```
set-location ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\")
./scr_openfiledlg_with_parms.ps1 "C:\" "All files (*.*)| *.*"
```

In this code, notice that the values passed get mapped to the parameters based on the order of the values passed. We can also explicitly map the values to the parameters as shown below.

```
set-location ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\")
./scr_openfiledlg_with_parms.ps1  -filter "All files (*.*)| *.*"  -initdir "C:\"
```

Calling code using named parameters improves readability and can help you to avoid encountering bugs. It also helps to insulate you from changes to the code if the parameter order changes.

Among some useful features related to parameters, PowerShell supports parameter validation. It is implemented using the `CmdletBinding` statement, as shown below.

```
[CmdletBinding()]
Param(
  [Parameter(Mandatory=$True,Position=1)]
  [string]$initdir,

  [Parameter(Mandatory=$True,Position=2)]
  [string]$filter
)
```

```
 [System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | Out-Null

 $OpenFileDialog = New-Object System.Windows.Forms.OpenFileDialog
 $OpenFileDialog.initialDirectory = $initdir
 $OpenFileDialog.filter = $filter
 $OpenFileDialog.ShowDialog() | Out-Null
 $OpenFileDialog.filename
```

By changing the parameter declaration to include `CmdletBinding`, we have made the parameters required via the `Mandatory` property. A nice aspect of this is that if we forget to pass a parameter, we will be prompted for it. We need to be careful about this if we are running the script in batch mode, however, such as from SQL Agent. `CmdletBinding` also enables the use of built-in common parameters such as `Debug` and `Verbose`, which allow the display or suppression of the output of `Write-Debug` and `Write-Verbose` statements. `Write-Debug` is intended to allow you to print informational messages during code testing and debugging. `Write-Verbose` is useful for display of extra information when desired. Let's take a look at how they are used in the following code:

```
[CmdletBinding()]
Param(
   [Parameter(Mandatory=$True,Position=1)]
   [string]$initdir,

   [Parameter(Mandatory=$True,Position=2)]
   [string]$filter
)
"VerbosePreference is $VerbosePreference"
 "DebugPreference is $DebugPreference"

 Write-Verbose "The initial directory is: $initdir"
 Write-Debug   "The filter is:  $filter"

 [System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | Out-Null

 $OpenFileDialog = New-Object System.Windows.Forms.OpenFileDialog
 $OpenFileDialog.initialDirectory = $initdir
 $OpenFileDialog.filter = $filter
 $OpenFileDialog.ShowDialog() | Out-Null
 $OpenFileDialog.filename
```

What PowerShell does when it sees `Write-Verbose` and `Write-Debug` cmdlets depends on the value assigned to `$VerbosePreference` and `$DebugPreference` variables respectively. These are predefined preference variables we can set to tell how PowerShell it should proceed after executing the `Write-Verbose` or `Write-Debug` statements. We can set them to `Stop`, which halts execution of the script, `SilentlyContinue`, which suppresses the message and continues with the line after the message, `Continue` which displays the message and continues with the next line, and `Inquire`, which prompts the user for the action they want to take. If we call the script with the `Verbose` switch, the `$VerbosePreference` variable is overridden with `Continue` and the `Write-Verbose` message is displayed. If we call the script with the `Debug` switch, the `$DebugPreference` variable is overridden with `Inquire` and we are prompted on what we want to do. The first two lines of the script above display the value of the preference variables. The default values for both preference variables is `SilentlyContinue`. Full documentation on PowerShell's preference variables is available at the link below.

http://technet.microsoft.com/en-us/library/hh847796.aspx

Please copy the script scr_openfiledlg_with_parms_andswitches.ps1 from the code accompanying the book to your documents folder. We can execute it and get the Write-Verbose message to display by including the Verbose switch parameter as shown below.

```
set-location ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\")
./scr_openfiledlg_with_parms_andswitches "C:\" "All files (*.*)| *.*" -Verbose
```

If we include the Debug switch as shown below, we are prompted on what action we want to take when the Write-Debug statement is encountered.

```
./scr_openfiledlg_with_parms_andswitches "C:\" "All files (*.*)| *.*" -Debug
```

Not only do we get the built-in Verbose and Debug switches, but we can also create switches of our own to use to control how our code will execute. The Open File dialog supports multiple-file selection if the MultiSelect property is set to True. Let's add a switch called Multifile to the script that, if passed, will enable the selection of multiple files. If not passed, the dialog will only allow one to be selected as before:

```
[CmdletBinding()]
Param(
  [Parameter(Mandatory=$True,Position=1)]
   [string]$initdir,

   [Parameter(Mandatory=$True,Position=2)]
   [string]$filter,

   [switch]$multifile
)

[System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | Out-Null

$OpenFileDialog = New-Object System.Windows.Forms.OpenFileDialog
$OpenFileDialog.initialDirectory = $initdir
$OpenFileDialog.filter = $filter

If ($multifile) {
  $OpenFileDialog.Multiselect = $true
}

$OpenFileDialog.ShowDialog() | Out-Null
$OpenFileDialog.filename
```

A nice feature of switches is that they are optional. Suppose this script was used by hundreds of other scripts and we needed to add an option like this. The switch provides an elegant way to do so without breaking any existing calls to the script.

Functions

The reusability supported by fall-through scripts is impressive, but the real power comes when we structure our code as functions. Everything we've learned so far about scripts applies to functions, except that if we run the code of a function, it will only define the function in memory without executing. We need code outside the function to actually execute the function. Functions can call other functions—i.e., can be nested. From a coding standpoint there are only a couple of changes needed to turn the above script into a function. Listing 3-1 shows the code as a function.

Listing 3-1. Function to call Windows Open File dialog

```
function Invoke-UdfOpenFileDialog{
[CmdletBinding()]
Param(
   [Parameter(Mandatory=$True,Position=0)]
   [string]$initdir,

   [Parameter(Mandatory=$True,Position=1)]
   [string]$filter,

   [switch]$multifile
)

   [System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | Out-Null

   $OpenFileDialog = New-Object System.Windows.Forms.OpenFileDialog
   $OpenFileDialog.initialDirectory = $initdir
   $OpenFileDialog.filter = $filter

   If ($multifile)
   {
     $OpenFileDialog.Multiselect = $true
   }

   $OpenFileDialog.ShowDialog() | Out-Null
   $OpenFileDialog.filename
}

# Example calling the function...
Invoke-UdfOpenFileDialog "C:\" "All files (*.*)| *.*"  -multifile
```

All that we did to the code above was to add the word `function` followed by the name we want to give the function. This was followed by an open brace, `{`. We marked the end of the function with an end brace, `}`. If we want to call functions with parameters being passed based on position, we can add the order attribute to the `Parameter` properties. The last line—`Invoke-UdfOpenFileDialog"C:\" "All files (*.*)| *.*" -multifile` —is not part of the function. We need it to call the function. It's a good idea to include a sample call at the end of a function to help in testing it and to help others understand how to call it. Just remember to comment out the call when you are done testing.

Functions are all about reusability, but if others don't understand how to use your function, it will probably not get used. Remember how PowerShell provides Get-Help to display detailed information on any built-in cmdlet? Well, we can add the documentation to our functions and make it accessible to Get-Help. It is called *comment-based help*, which means that you can add comments to your code with special tags known to PowerShell, and Get-Help will pull them out and display them on demand. Let's look at the function we saw earlier with comment-based help added in Listing 3-2.

Listing 3-2. Function to call Windows Open File dialog with comment-based help

```
function Invoke-UdfOpenFileDialog{
<#
.SYNOPSIS
    Opens a Windows Open File Common Dialog.
.DESCRIPTION
    Use this function when you need to provide a selection of files to open.
.NOTES
    Author: Bryan Cafferky, BPC Global Solutions, LLC

.PARAMETER initdir
The directory to be displayed when the dialog opens.

.PARAMETER title
This is the title to be put in the window title bar.

.PARAMETER filter
The file filter you want applied, such as *.csv in the format 'All files (*.*)| *.*' .

.PARAMETER multifile
Switch that is passed enables multiple files to be selected.

.LINK
    Place link here.
.EXAMPLE
   Invoke-UdfOpenFileDialog "C:\" "All files (*.*)| *.*"  -multifile
.EXAMPLE
    Invoke-UdfOpenFileDialog "C:\" "All files (*.*)| *.*"
#>

[CmdletBinding()]
Param(
  [Parameter(Mandatory=$True,Position=0)]
   [string]$initdir,

   [Parameter(Mandatory=$True,Position=1)]
   [string]$filter,

   [switch]$multifile
)
```

```
  [System.Reflection.Assembly]::LoadWithPartialName("System.windows.forms") | Out-Null
  $OpenFileDialog = New-Object System.Windows.Forms.OpenFileDialog
  $OpenFileDialog.initialDirectory = $initdir
  $OpenFileDialog.filter = $filter

  If ($multifile)
  {
    $OpenFileDialog.Multiselect = $true
  }

  $OpenFileDialog.ShowDialog() | Out-Null
  $OpenFileDialog.filename

}
```

The statement below will call `Get-Help` for our function.

```
Get-Help  Invoke-UdfOpenFileDialog-full
```

`Get-Help` displays the comments from the function. A partial Listing of the output is shown below.

```
NAME
     Invoke-UdfOpenFileDialog

SYNOPSIS
    Opens a Windows Open File Common Dialog.

SYNTAX
    Invoke-UdfOpenFileDialog[-initdir] <String> [-filter] <String> [-multifile]
[<CommonParameters>]

DESCRIPTION
    Use this function when you need to provide a selection of files to open.

PARAMETERS
    -initdir <String>
        The directory to be displayed when the dialog opens.

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

Professional features like this are critical to separating a simple scripting language from a professional development language.

There are a lot of reserved documentation tags. See this link for full coverage:

http://technet.microsoft.com/en-us/library/hh847834.aspx

It's All about Objects

As we've seen, PowerShell is object based, so it makes sense that we can create our own custom objects with properties and methods. If you are used to an object-orientated programming language like C#, be aware that the way you accomplish this in PowerShell is different. There is no support for creating classes, nor are there any statements for having source code inherit methods and properties from other source code. However, PowerShell does recognize the .Net hierarchy of objects you create and exposes their methods and properties. PowerShell provides us with an object type called `psobject` that is like a template to which we can add properties and methods.

Since this is a book about using PowerShell for database development, let's create a useful example of a common task—looking up values based on a key. In this case, the example creates a state object that allows us to translate the state code into the state name and the state name into the state code. It also has a Show method that will display the state data in a grid view. The nice thing about this is that once we have the function written, we can just call it when we need it. Let's look at Listing 3-3.

Listing 3-3. Function New-UdfStateObject to look up state codes

```
function New-UdfStateObject
{
 [CmdletBinding()]
        param (
              [ref]$stateobject
          )

   $instates = Import-CSV -PATH `
               (($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\") + "state_table.csv")

   # Properties...

    $stateobject.value  | Add-Member -MemberType noteproperty `
                                  -Name statedata `
                                  -Value $instates `
                                  -Passthru

   [hashtable] $stateht = @{}
   foreach ($item in $instates) {$stateht[$item.abbreviation] = $item.name}

   $stateobject.value  | Add-Member -MemberType noteproperty `
                                 -Name Code `
                                 -Value $stateht  `
                                 -Passthru

   [hashtable] $statenameht = @{}
   foreach ($item in $instates) {$statenameht[$item.name] = $item.abbreviation}

   $stateobject.value  | Add-Member -MemberType noteproperty `
                                 -Name Name `
                                 -Value $statenameht `
                                 -Passthru
```

```
  #  Methods...

    $bshowdata = @'

     $this.statedata | Out-GridView

'@

    $sshowdata = [scriptblock]::create($bshowdata)

    $stateobject.value | Add-Member -MemberType scriptmethod `
                                    -Name Show `
                                    -Value $sshowdata `
                                    -Passthru
}
```

Let's look at the parameter-binding code from Listing 3-3, copied here:

```
[CmdletBinding()]
param (
    [ref]$stateobject
)
```

New-UdfStateObject declares a parameter named $stateobject that is a type REF, which is a reference to the object. This allows the code to modify the object passed in—i.e., we can add properties and methods to it. When we call this function, we will pass in a psobject, and the function will add useful properties and methods to it.

The first thing the function does is load the state data from a csv file. Thank you to Dave Ross, a Boston-area web developer, for his free state data at http://statetable.com/. Please go to this site and download the file to your Documents folder as state_table.csv. The code below, from New-UdfStateObject, loads this data and adds it as a property to our object, $stateobject:

```
$instates = Import-CSV -PATH `
            (($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\") + "state_table.csv")

$stateobject.value  | Add-Member      -MemberType noteproperty `
                                      -Name statedata `
                                      -Value $instates `
                                      -Passthru
```

In Listing 3-3, we use the built-in cmdlet Import-CSV to load the CSV file into a format that we can directly add as an object property. This is not a hash table, but rather is a collection of rows with each column name exposed as a property. We can add methods that can use this property to do things like display the data in a grid, which we do in the Show method. Hash tables are great, but they are limited to two columns. By loading the entire set of rows into a property, we can pull out any columns we need later.

To modify the object passed into the function, the code must set the referenced object's Value property. This is important as you cannot directly update the object; i.e., $stateobject.value = 1 will assign a value of 1 to the object. The cmdlet Add-Member is used to add a new property named statedata to the referenced object and assigns the imported CSV data to the property. The Passthru switch tells PowerShell to pipe this property to cmdlets that support piping. We must pipe the psobject variable that was passed in as a parameter into the Add-Member cmdlet to add properties or methods. Note the tick marks, `, which are used as a line continuation to keep the code readable.

The function adds two additional properties that themselves are objects—i.e., hash tables. By adding these hash tables to a psobject, we can use the hash table properties as methods that allow us to look up a value from a key. We have state names and state codes, so we add the hash table property twice, reversing the name and code on each. This lets us look up the name for a code or the code for a name.

Here is how we load the first hash table...

```
[hashtable] $stateht = @{}
foreach ($item in $instates) {$stateht[$item.abbreviation] = $item.name}

$stateobject.value  | Add-Member -MemberType noteproperty `
                                 -Name Code `
                                 -Value $stateht  `
                                 -Passthru
```

Above, we create an empty hash table type with the statement 'hashtable] $stateht = @{}'. Then we loop through each row in the state data collection using the foreach statement. $item will contain the specific row on each iteration, so we use that to get the abbreviation and name and assign them respectively to the key and value properties of the hash table. By syntax convention, what is on the left side is the key and what is on the right side is the value; i.e., what will be returned is based on the key given—format is $key = $value.

Let's look at how we attach a script to a psobject shown below.

```
    $bshowdata = @'

     $this.statedata | Out-GridView

'@

    $sshowdata = [scriptblock]::create($bshowdata)

    $stateobject.value | Add-Member -MemberType scriptmethod `
                                    -Name Show `
                                    -Value $sshowdata `
                                    -Passthru
```

In New-UdfStateObject, the code block is assigned to variable $bshowdata. Then we cast $bshowdata to a script:block type because that is what the MemberType scriptmethod expects. A script method is really just another property, but it contains code that can be executed. $stateobject.value is piped into the Add-Member cmdlet, where the scriptmethod MemberType named Show is added.

Once we have executed the function in Listing 3-3 to define it in memory, we can use it with the lines below.

```
[psobject] $states1 = New-Object psobject
New-UdfStateObject ([ref]$states1)
```

The first line above creates a new psobject variable named $states1. Then we call the function New-UdfStateObject, passing a reference to $states1. The function will add the properties and method to $states1.

After running the above statements, enter the statement below:

```
$states1.Code["RI"]
```

You should get the following output:

```
PS C:\Windows\system32> $states1.Code["RI"]
Rhode Island

$states1.Name["Vermont"]
```

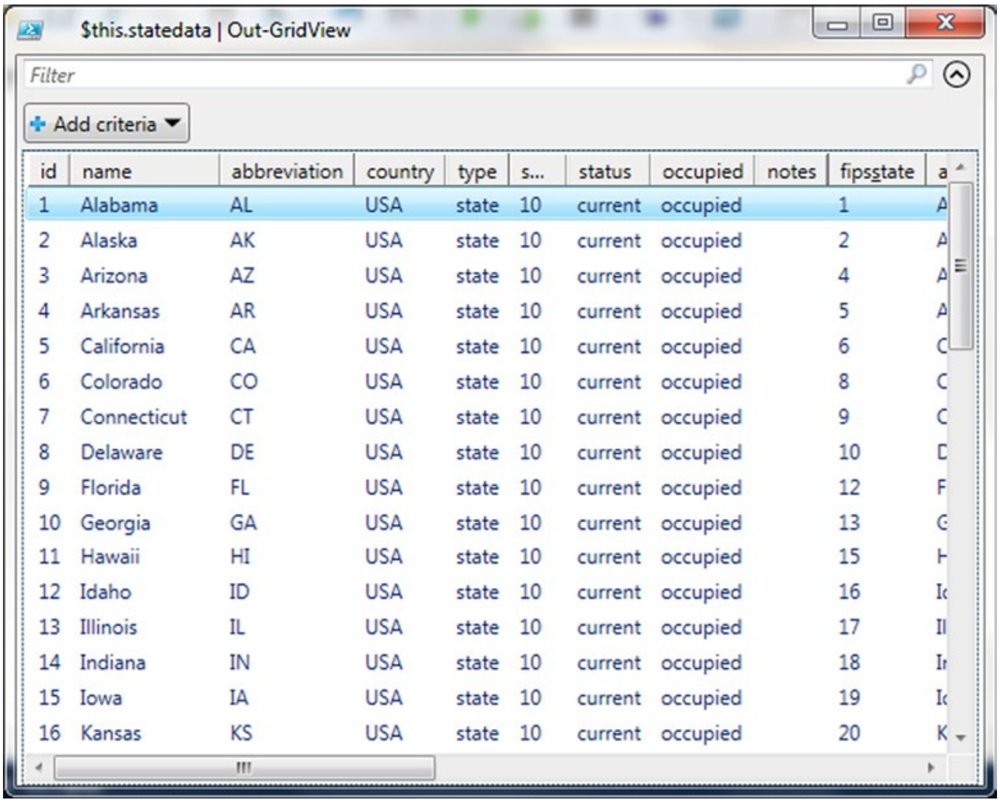
Which displays the abbreviation for the state as shown below.

```
VT
```

To demonstrate that we can add methods to the object, we added a `scriptproperty` that displays the state data in a grid via `Out-GridView`. Enter the following statement to see how this works:

```
$states1.Show()
```

We get the data presented in a grid, as shown in Figure 3-2.



id	name	abbreviation	country	type	s...	status	occupied	notes	fipsstate	a
1	Alabama	AL	USA	state	10	current	occupied		1	A
2	Alaska	AK	USA	state	10	current	occupied		2	A
3	Arizona	AZ	USA	state	10	current	occupied		4	A
4	Arkansas	AR	USA	state	10	current	occupied		5	A
5	California	CA	USA	state	10	current	occupied		6	C
6	Colorado	CO	USA	state	10	current	occupied		8	C
7	Connecticut	CT	USA	state	10	current	occupied		9	C
8	Delaware	DE	USA	state	10	current	occupied		10	D
9	Florida	FL	USA	state	10	current	occupied		12	F
10	Georgia	GA	USA	state	10	current	occupied		13	G
11	Hawaii	HI	USA	state	10	current	occupied		15	H
12	Idaho	ID	USA	state	10	current	occupied		16	Id
13	Illinois	IL	USA	state	10	current	occupied		17	Il
14	Indiana	IN	USA	state	10	current	occupied		18	In
15	Iowa	IA	USA	state	10	current	occupied		19	Io
16	Kansas	KS	USA	state	10	current	occupied		20	K

Figure 3-2. *Using Out-GridView from a function*

A Real-World Use of PowerShell Objects for ETL

Since the goal of this book is to provide real-world examples, I want to present a fairly extensive function that really shows you how to get a bang out of the psobject. This is coded as a function so it can easily be called wherever it is needed. In my ETL work I find I often have to do look ups related to geographic data—i.e., look up a territory for a region, convert currency, and so on. The function creates a psobject with the following properties and methods:

Properties

Territory – a collection of sales territories

Currency – list of currency codes and names

CurrencyConversion – conversion rate by currency

Methods

GetCountryForTerritory – returns the country a territory is in

ConvertCurrency – converts amount passed from US dollars to the currency specified

Listing 3-4 shows the code. It is a long listing, but we will go over it in detail after you have reviewed it. Note: You must have the SQLPS module installed for the code to work. This installs automatically when SQL Server client tools are installed. For information on installing the SQLPS module, see the link below.

https://msdn.microsoft.com/en-us/library/hh231683.aspx

Listing 3-4. Function New-UdfGeoObject supports code translations

```
function New-UdfGeoObject ()
{
 [CmdletBinding()]
        param (
              [ref]$geoobject
          )

 #  Check if the SQL module is loaded and, if not, load it.
     if(-not(Get-Module -name "sqlps"))
        {
              Import-Module "sqlps"
        }

  set-location SQLSERVER:\SQL\BryanCafferkyPC\DEFAULT\Databases\Adventureworks\Tables

   # Load Territory

   $territoryrs = Invoke-Sqlcmd -Query "SELECT distinct [Name],[CountryRegionCode] FROM
   [AdventureWorks].[Sales].[SalesTerritory];" -QueryTimeout 3

   $territoryhash = @{}
   foreach ($item in $territoryrs) {$territoryhash[$item.Name] = $item.CountryRegionCode}
   $geoobject.value | Add-Member -MemberType noteproperty `
                         -Name Territory `
                         -Value $territoryhash

   #  Load Currency

   $currencyrs = Invoke-Sqlcmd -Query "SELECT [CurrencyCode], [Name] FROM [AdventureWorks].
   [Sales].[Currency];" -QueryTimeout 3

   $currencyhash = @{}
   foreach ($item in $currencyrs) {$currencyhash[$item.CurrencyCode] = $item.Name}


   $geoobject.value | Add-Member -MemberType noteproperty `
                         -Name Currency `
                         -Value $currencyhash

    # Load CurrencyConversion

    $sql = "SELECT [FromCurrencyCode], [ToCurrencyCode],[AverageRate], [EndOfDayRate],
    cast([CurrencyRateDate] as date) as CurrencyRateDate
            FROM [AdventureWorks].[Sales].[CurrencyRate]
            where [CurrencyRateDate] = (select max([CurrencyRateDate]) from
            [AdventureWorks].[Sales].[CurrencyRate]);"

    $convrate = @{}

    $convrate = Invoke-Sqlcmd -Query $sql -QueryTimeout 3

    $geoobject.value | Add-Member -MemberType noteproperty `
                         -Name CurrencyConversion `
                         -Value $convrate

    #  Define Methods...

    #  Territory - Look Up Country

    $bterritorytocountry = @'
    param([string] $territory)

    RETURN $this.Territory["$territory"]
'@

    $sterritorytocountry = [scriptblock]::create($bterritorytocountry)

    $geoobject.value | Add-Member      -MemberType scriptmethod `
                                       -Name GetCountryForTerritory `
                                       -Value $sterritorytocountry `
                                       -Passthru
    $convertcurrency = @'
    param([string] $targetcurrency,[decimal] $amount)

     $row = $this.CurrencyConversion | where-object {$_["ToCurrencyCode"] -eq
     "$targetcurrency" }
     $result = $row["AverageRate"] * $amount
     RETURN $result
'@

    $sconvertcurrency = [scriptblock]::create($convertcurrency)

    $geoobject.value | Add-Member -MemberType scriptmethod `
                            -Name ConvertCurrency `
                            -Value $sconvertcurrency `
                            -Passthru
}
```

The function in Listing 3-4 is similar to `New-UdfStateObject` we saw earlier, but this one adds a little complexity to show how much can be done using the `psobject`.

Loading the SQLPS module can take a couple of minutes, so to avoid reloading it with every execution of the function during testing, we can tell PowerShell to load it only if it is not loaded already by using this code:

```
if(-not(Get-Module -name "sqlps"))
{
    Import-Module "sqlps"
}
```

The SQLPS module provides easy access to SQL Server functionality. We will talk about this module in detail later, but to understand the code above, we need to review the SQL Server–related code. Consider this statement:

```
Set-Location SQLSERVER:\SQL\BryanCafferkyPC\DEFAULT\Databases\Adventureworks\Tables
```

PowerShell provides an abstraction that allows us to navigate SQL Server like we do a file system. The `Set-Location` cmdlet sets your context to where you specify. Note: The setting above must be modified to fit your environment and you must have the AdventureWorks database installed. The format of the cmdlet is:

```
set-location SQLSERVER:\SQL\[Machine Name]\[Instance]\Databases\[DatabaseName]\[ObjectType]
```

Where:

`[MachineName]`	=	The server name.
`[Instance]`	=	The SQL Server Instance Name, if there is more than one
`[DatabaseName]`	=	The database you want to access
`[ObjectType]`	=	The type of object you want to access, i.e., tables, views, etc.

Once the context is set, we can run multiple SQL queries, and they will all use that context, as the statement that follows does:

```
$territoryrs = Invoke-Sqlcmd -Query "SELECT distinct [Name],[CountryRegionCode] FROM
[AdventureWorks].[Sales].[SalesTerritory];" -QueryTimeout 3
```

The statement above submits the SQL statement we pass in the `Query` parameter and returns the results to `$territoryrs`. The results are in a format that can easily be used by PowerShell. For example, we could pipe the results to `Out-GridView` or to a file.

To get a quick sense of what this `psobject` offers, let's review Listing 3-5, which exercises its features. You will need to run Listing 3-4 which defines the function `New-UdfGeoObject` before running the code in Listing 3-5. Reminder: The function uses the SQLPS module so that module must be installed to run the code in Listing 3-5.

Listing 3-5. Example of using the geo object

```
# Create the geo object...
[psobject]$mygeo = New-Object PSObject
New-UdfGeoObject ([ref]$mygeo)

#  Since this is a hash table, let's look up a territory by country...
Write-Host "`r`nTranslate Territory directly from Global Object not fully Qualified..."
$mygeo.Territory["Southeast"]

Write-Host "`r`nTranslate Territory using a custom method..."
$mygeo.GetCountryForTerritory("Southeast")

write-host "`r`nCurrency Keys..."
write-host $mygeo.currency.Keys

Write-Host "`r`nCurrency Values..."
write-host $mygeo.currency.Values

Write-Host "`r`nTranslate currency code to currency name..."
$mygeo.currency["FRF"]

Write-Host "`r`nLook up currency conversion row for USD to Japanese Yen..."
$mygeo.CurrencyConversion  | where-object {$_["ToCurrencyCode"] -eq "JPY" }

Write-Host "`r`nConvert currency from USD to Japanese Yen..."
$mygeo.ConvertCurrency("JPY", 150.00)
```

Listing 3-5 starts by creating a new `psobject` variable to hold the geo object. We then call `New-UdfGeoObject`, passing the variable by reference, i.e., using the `REF` type. Passing by reference allows the function to modify the object passed in. Notice the escape characters in the `Write-Host` statements above—"`` `r`n ``". These send commands to the formatter; the `` `r `` is a carriage return and `` `n `` is a line feed. Embedding them like this gives us a quick way to get extra spacing rather than adding extra `Write-Host` statements.

Using objects as containers for methods and properties is powerful, but we can also extend the objects. In C#, classes can inherit other classes thereby gaining the methods and properties of those classes. However, that is done at the source-code level, and as of version 4.0, PowerShell does not support inheritance or the ability to define classes. *Note: PowerShell 5.0 will implement support for inheritance and* *the creation of classes.* Instead it is object based, and we can achieve much of the same functionality using objects. For example, we just created a useful geo object. Let's extend that object by adding more properties and methods using the function created in Listing 3-6.

Listing 3-6. Extending the geo object with the function New-UdfZipCodeObject.

```
function New-UdfZipCodeObject{

 [CmdletBinding()]
        param (
              [ref]$zipobject
          )

    #  Load Zip Codes

    $zipcodes = Import-CSV `
                ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\" + "free-zipcode-database.csv")


     $zipobject.value | Add-Member -MemberType noteproperty `
                           -Name ZipCodeData `
                           -Value $zipcodes

  #  Find Zip Codes...

     $bgetdataforzip = @'
     param([string] $zip)

     RETURN $this.ZipCodeData | where-object {$_.Zipcode -eq "$zip" }
'@

    $sgetdataforzip = [scriptblock]::create($bgetdataforzip)

    $zipobject.value | Add-Member -MemberType scriptmethod `
                            -Name GetZipDetails `
                            -Value $sgetdataforzip `
                            -Passthru
}
```

We call this function passing in the geo object we created earlier. Therefore, you must run Listings 3-4 and 3-5 before you run the code below. The idea is that we are extending the previously created object.

```
New-UdfZipCodeObject([ref]$mygeo)
```

Note that we do not declare the `$mygeo` object variable because it already exists. We pass it into the function to add zip code functionality, using the `REF` type again so the function can extend the object.

Let’s test the features added using the following code:

```
Write-Host "`r`nVerify a Zip Code is valid..."
$mygeo.ZipCodeData.Zipcode.Contains("02886")

Write-Host "`r`nGet the details for a zip code..."
$mygeo.GetZipDetails("02886")

Write-Host "`r`nGet the city or cities for a zip..."
$mygeo.GetZipDetails("02886").City
```

Creating custom objects in PowerShell using `psobject` may seem unintuitive at first. The `REF` type creates a wrapper around the object that it uses to access it. The way to modify the object is to assign values to the `Value` property. The `REF` type is an object is its own right, but its purpose is to allow you to access the object it references. This is to avoid the risk that code might corrupt the object.

Remember the Pipeline

PowerShell is designed around a concept called the pipeline; i.e., the output of one command becomes the input to the next. Many cmdlets support this, which is why you can do so much with so few lines of code. To really leverage PowerShell, you need to know how to use the pipeline in your code. It really is easy to do, and if you come from a development background, it should seem fairly intuitive. For example, if you were to write a SQL stored procedure that used a cursor to read a table and step through each row, then you have the concept of programming for the pipeline already. In the stored procedure you would 1) have a start block of code that opens the cursor and perhaps gets the first row; 2) have a loop to process the rows; and 3) have an ending block that closes the cursor and cleans up after itself. In PowerShell, writing a function to process a pipeline consists of the same basic parts, which are listed here:

`Begin`	Runs only once upon initial call of the function and does not have access to the pipeline.
`Process`	Runs once for each item in the pipeline. This is where you process the pipeline.
`End`	Runs only after the pipeline has been processed. Good place to clean things up.

Extract, Transform and Load (*ETL*) work is all about sources, transformations, and destinations, and you can think of the pipeline as the equivalent of a *SQL Server Integration Services* (SSIS) data flow. A difference, however, is that unlike SSIS, the pipeline is seamlessly integrated into the program flow.

Listing 3-7 is a simple example of a function that processes the pipeline.

Listing 3-7. A function that processes the pipeline

```
function Invoke-UdfPipeProcess( [Parameter(ValueFromPipeline=$True)]$mypipe = "default")

{

    begin {
            Write-Host "Getting ready to process the pipe. `r`n"
          }
    process {  Write-Host "Processing : $mypipe"
            }

    end {Write-Host "`r`nWhew!  That was a lot of work." }

}

# Code to call the function...
Set-Location  ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\")
Get-ChildItem | Invoke-UdfPipeProcess
```

Let's look at the function declaration in Listing 3-7. The key thing is the `ValueFromPipeline` attribute. This tells PowerShell to expect this parameter through the pipeline. The `begin` block is just going to print a message once before the pipeline gets passed. Then the `process` will print out the items in the pipeline, in this case a list of file names. Finally, the end block prints a message after the pipeline has been processed. The output on my machine is shown below.

```
Getting ready to process the pipe.

Processing : Program Files
Processing : Program Files (x86)
Processing : SQL Server 2008 R2
Processing : Temp
Processing : Users
Processing : Windows
Processing : header.txt
Processing : set_config.bat

Whew!  That was a lot of work.

PS C:\>
```

Pipelining Example: SQL Server Deployment

A common problem in database development is managing the database deployment process—i.e., maintaining the scripts to create and alter the database objects and establishing a process to easily deploy these objects and run the scripts. In one place I worked, they developed a custom in-house application called a kit manager to do this. The SQLPS module allows us to easily execute SQL scripts, so I thought an example of a function that uses the pipeline to process SQL scripts would be a great way to explain this topic. Let's start by reviewing the steps involved.

Steps:

`Begin`	Connect to the target server and database. Write out log file header rows.
`Process`	Run each script file passed in the pipeline and write a row to the log file.
`End`	Clean things up and write ending log row entries.

Listing 3-8 shows a function that implements these steps to run all the scripts passed in the pipeline.

Listing 3-8. A function, Invoke-UdfSQLScript, that uses the pipeline to execute SQL scripts

```
function Invoke-UdfSQLScript
([string] $p_server, [string] $p_dbname, [string] $logpath)

{

 begin
 {
   if(-not(Get-Module -name "sqlps")) { Import-Module "sqlps" }

   set-location "SQLSERVER:\SQL\$p_server\DEFAULT\Databases\$p_dbname\"
   write-host $p_server $p_dbname
   "Deployment Log:  Server: $p_server Target Datbase: $p_dbname Date/Time: " `
    + (Get-Date)  + "`r`n" > $logpath
    # > means create/overwrite and >> means append.
   }

  process
  {
   Try
     {
      $filepath = $_.fullname   # grab this so we can write it out in the Catch block.
      $script = Get-Content  $_.fullname       # Load the script into a variable.
      write-host $filepath
      Invoke-Sqlcmd -Query  "$script ;" -QueryTimeout 3
      "Script: $filepath successfully processed. " >> $logpath
      }
      Catch
      {
        write-host "Error processing script: $filepath . Deployment aborted."
        Continue
      }
   }

   end
   {
     "`r`nEnd of Deployment Log:  Server: $p_server Target Datbase: $p_dbname Finished
     Date/Time: " + (Get-Date)  + "`r`n" >> $logpath
   }
}

# Lines to call the function. Passing paramters positionally...

$datapath = $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\"
$scriptpath = $datapath
$logpath    = $datapath + "deploy1_log.txt"
Get-ChildItem -Path $scriptpath | Invoke-UdfSQLScript "(local)" "AdventureWorks" $logpath
```

When we run the code above to call the function `Invoke-UdfSQLScript`, we should see lines similar to the output below.

```
(local) AdventureWorks
C:\Users\BryanCafferky\Documents\create_table_1.sql
WARNING: Using provider context. Server = (local), Database = AdventureWorks.
```

The script `create_table_1.sql` was provided with the book files and should be copied to your Documents folder prior to running the code. Make sure there are no other files with an extension of `.sql` in your Documents folder as this function will attempt to execute them. This script was executed by the function `Invoke-UdfSQLScript`. The function will execute any files in the Documents folder with the `sql` file extension so if you have any SQL scripts already in Documents, they will execute also. You may see `Verbose` and `Debug` messages display depending on the setting of the PowerShell `$DebugPreference` variable. Preference variables will be discussed in Chapter 8.

In Listing 3-8, we create `$datapath` to hold the path to our files using the environment variables `$env:HomeDrive` and `$env:HomePath`, which always point to the current user's home drive and path, i.e., `C:\user1\`. Then we just concatenate the `Documents` folder to this. This is to make it easier to run the sample. In reviewing Listing 3-8, notice that functions can get the pipeline as a parameter while still supporting non-pipeline parameters. This is important because we may need to provide information to the function on how we want the pipeline processed. In this case, the server name, database name, and log file path are passed as regular parameters. Notice that we do not declare the pipeline as a parameter. You *can* do that, but in this case, the pipeline is passed to the function automatically. This is why we use the default name `$_` to access items in the pipeline. In general, it is better to declare the pipeline as a parameter, as it makes the function's intention clearer. See the code below.

```
function Invoke-UdfSQLScript
([string] $p_server, [string] $p_dbname, [string] $logpath)
```

The `begin` block is for initialization purposes as it only gets called once, before the pipeline is passed to the function. This means the `begin` block cannot access the pipeline. However, the non-pipeline parameters are available to the `begin` block. Above, the `begin` block is used to load SQLPS, if needed, and to point SQLPS to the server and database that we want the scripts to target. We also write out the header rows of our log file. Writing lines to a file is so simple that it can be overlooked. To write a line to a file, use one greater than sign, `>`, to replace the contents of the file or create it if it does not exist. To append a line to a file, use two greater than signs together, `>>`. Let's look at the `begin` code block below.

```
begin
 {
     if(-not(Get-Module -name "sqlps")) { Import-Module "sqlps" }

   set-location "SQLSERVER:\SQL\$p_server\DEFAULT\Databases\$p_dbname\"
   write-host $p_server $p_dbname
   "Deployment Log:  Server: $p_server Target Datbase: $p_dbname Date/Time: " `
   + (Get-Date)  + "`r`n" > $logpath
   # > means create/overwrite and >> means append.

 }
```

The process block is the engine of the function—i.e., it processes each item in the pipeline. So, how did we get away with not declaring the pipeline as a parameter? PowerShell automatically tracks the current item in the pipeline with a name of $_. Note: The entire pipeline is available via the default pipeline object variable $input.

```
process
{
Try
  {
    $filepath = $_.fullname   # grab this so we can write it out in the Catch block.
    $script = Get-Content  $_.fullname       # Load the script into a variable.
    write-host $filepath

    Invoke-SQLcmd -Query  "$script ;" -QueryTimeout 3
    "Script: $filepath successfully processed. " >> $logpath
  }
    Catch
    {
      write-host "Error processing script: $filepath . Deployment aborted."
      Continue
    }

}
```

When we pipe the output of Get-ChildItem as we do above, we get a number of attributes. Notice how we use the Fullname property of $_ to get the full path with file name for each script. Why are we storing the file path in the $filepath variable? As the Catch block cannot access the pipeline, we store the file path in a variable so we can display it if an error occurs. After each script has executed, we write a line to the log file.

We could write this function using the Invoke-SQLCmd parameter InputFile to directly execute the script from the file. However, if an error occurs, the error messages are written to the console and cannot be captured. There is the option to suppress error messages by passing the parameter OutputSQLErrors $false, but then the Catch block would not be fired and the error would go unnoticed. Using the Query parameter allows us to trap an error if one occurs. Now let's look at the end block below.

```
end
{
    "`r`nEnd of Deployment Log:  Server: $p_server Target Datbase: $p_dbname
    Finished Date/Time: " + (Get-Date)  + "`r`n" >> $logpath }
}
```

The end block is simple. This is where we wrap things up. There is a nice discussion of pipeline processing at this link:

https://www.simple-talk.com/dotnet/.net-tools/down-the-rabbit-hole--a-study-in-powershell-pipelines,-functions,-and-parameters/

Formatting Output

Out-GridView offers a nice presentation and user experience for out-of-the-box output viewing, but there are times when that will not suit our needs. For example, we may want to email the output. A handy cmdlet, ConvertTo-HTML, converts the output to HTML that we can then display.

Let's display the state_table.csv file we downloaded. Displaying output in HTML format is not a one-line command, because there is no built-in Out-HTML cmdlet. However, with a few lines, we can do it with the code here:

```
# First, we need to clear the SQL Provider by setting the
# location to a system folder...
Set-Location ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents\")

# Now we can run our code...
$datapath = $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\"
$instates = Import-CSV ($datapath + “state_table.csv")
$instates | ConvertTo-HTML | Out-File MyBasicReport.HTML
Invoke-Item  MyBasicReport.HTML
```

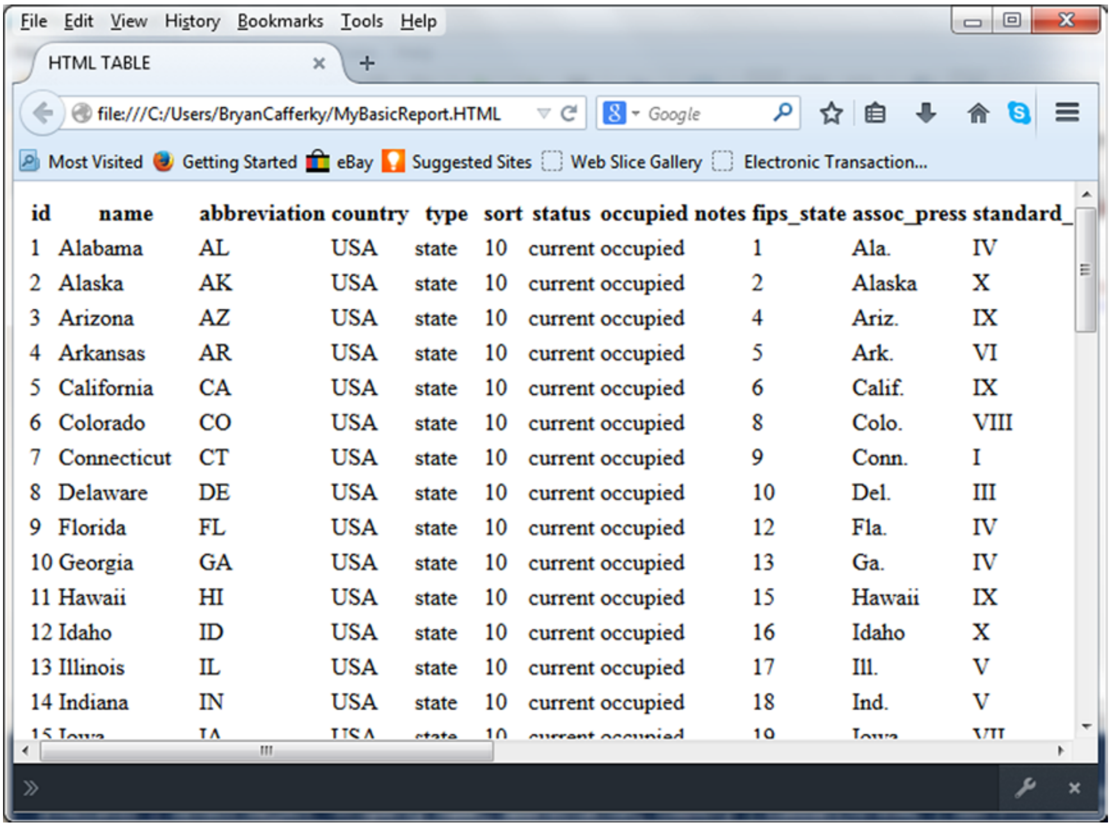
Providers like the one created by the SQLPS module can interfere with other PowerShell code. To reset the provider context back to the file system, the first line above uses Set-Location to change the current location to the user's Documents folder. The line after that sets $datapath to the user's Documents folder. Then, we use Import-CSV to load the file into the variable $instates. The path parameter to the Import-CSV cmdlet is enclosed in parentheses to force PowerShell to resolve the value before passing it to the cmdlet. The line below converts the data in $instates to HTML and saves to file MyBasicReport.html.

```
$instates | ConvertTo-HTML | Out-File MyBasicReport.html
```

We need the next line to get the files displayed:

```
Invoke-Item  MyBasicReport.HTML
```

Invoke-Item can execute a command or open a file, if the file type is registered on the machine—i.e., it displays the file with the program associated for the file type. This is very useful, as we can use it with Office documents and any other file types your machine knows about. In this case, it will use the default browser to display the HTML file, which should look something like Figure 3-3.


File Edit View History Bookmarks Tools Help
HTML TABLE
file:///C:/Users/BryanCafferky/MyBasicReport.HTML
Google
Most Visited
Getting Started
eBay
Suggested Sites
Web Slice Gallery
Electronic Transaction...

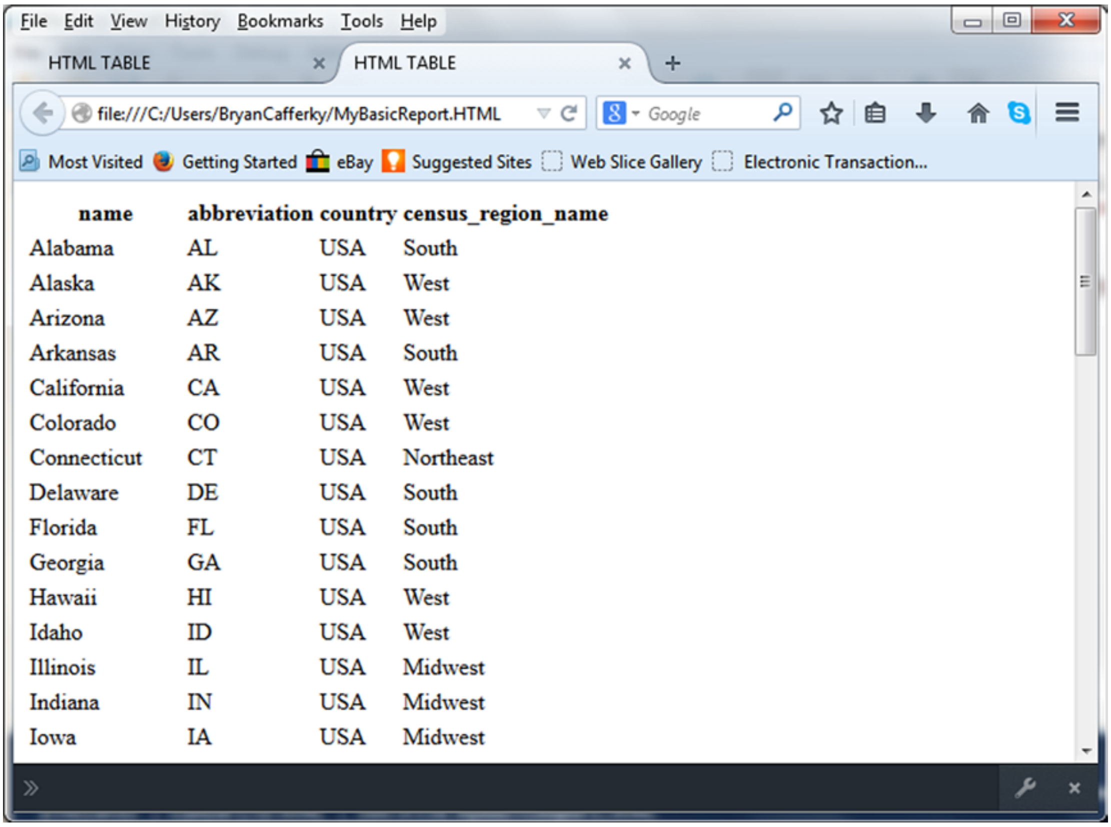
id	name	abbreviation	country	type	sort	status	occupied	notes	fips_state	assoc_press	standard_
1	Alabama	AL	USA	state	10	current	occupied		1	Ala.	IV
2	Alaska	AK	USA	state	10	current	occupied		2	Alaska	X
3	Arizona	AZ	USA	state	10	current	occupied		4	Ariz.	IX
4	Arkansas	AR	USA	state	10	current	occupied		5	Ark.	VI
5	California	CA	USA	state	10	current	occupied		6	Calif.	IX
6	Colorado	CO	USA	state	10	current	occupied		8	Colo.	VIII
7	Connecticut	CT	USA	state	10	current	occupied		9	Conn.	I
8	Delaware	DE	USA	state	10	current	occupied		10	Del.	III
9	Florida	FL	USA	state	10	current	occupied		12	Fla.	IV
10	Georgia	GA	USA	state	10	current	occupied		13	Ga.	IV
11	Hawaii	HI	USA	state	10	current	occupied		15	Hawaii	IX
12	Idaho	ID	USA	state	10	current	occupied		16	Idaho	X
13	Illinois	IL	USA	state	10	current	occupied		17	Ill.	V
14	Indiana	IN	USA	state	10	current	occupied		18	Ind.	V
15	Iowa	IA	USA	state	10	current	occupied		19	Iowa	VII

Figure 3-3. *Presenting output in HTML*

It seems a bit cluttered, so let's eliminate columns we don't need via the `Select-Object` cmdlet, shown below in bold.

```
$instates = Import-CSV ($datapath + "state_table.csv")
$instates | Select-Object -Property name, abbreviation, country, census_region_name |
ConvertTo-HTML | Out-File MyBasicReport.HTML
Invoke-Item  MyBasicReport.HTML
```

This code pipes the data through the `Select-Object` cmdlet, which allows us to pick the properties we want to keep in the pipeline using the `Property` parameter. Now we will just get the columns we asked for, as shown in Figure 3-4.


File Edit View History Bookmarks Tools Help
HTML TABLE
HTML TABLE
file:///C:/Users/BryanCafferky/MyBasicReport.HTML
Google
Most Visited
Getting Started
eBay
Suggested Sites
Web Slice Gallery
Electronic Transaction...

name	abbreviation	country	census_region_name
Alabama	AL	USA	South
Alaska	AK	USA	West
Arizona	AZ	USA	West
Arkansas	AR	USA	South
California	CA	USA	West
Colorado	CO	USA	West
Connecticut	CT	USA	Northeast
Delaware	DE	USA	South
Florida	FL	USA	South
Georgia	GA	USA	South
Hawaii	HI	USA	West
Idaho	ID	USA	West
Illinois	IL	USA	Midwest
Indiana	IN	USA	Midwest
Iowa	IA	USA	Midwest

Figure 3-4. *HTML output of selected columns*

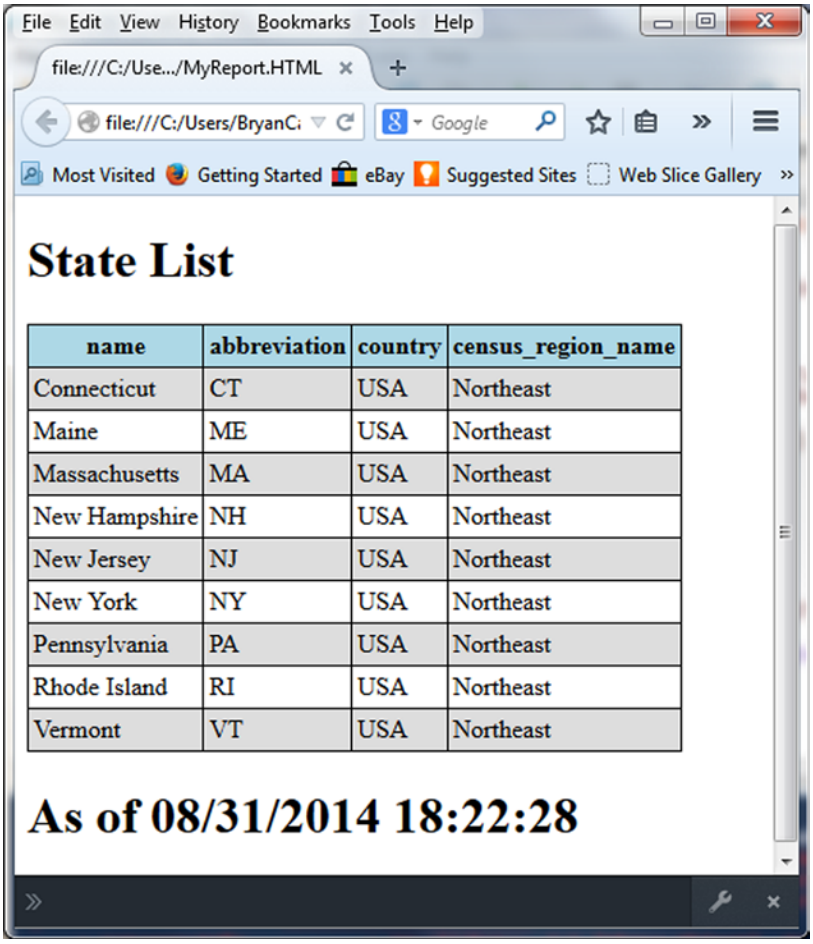
We can add more enhancements to the output. The `ConvertTo-HTML` supports some parameters that allow us to customize the format. Rather than do this as a one-time, hard-coded script, let's do this with a reusable function, as shown in Listing 3-9.

Listing 3-9. A function, Out-UdfHTML, that converts output to HTML

```
function Out-UdfHTML ([string] $p_headingbackcolor, [switch] $AlternateRows)
{
  if ($AlternateRows) {$tr_alt = "TR:Nth-Child(Even) {Background-Color: #dddddd;}"}

  $format = @"
  <style>
  TABLE {border-width: 1px;border-style: solid;border-color: black;border-collapse:
  collapse;}
  TH {border-width: 1px;padding: 3px;border-style: solid;border-color: black;
  background-color:
  $p_headingbackcolor;}
  $tr_alt
  TD {border-width: 1px;padding: 3px;border-style: solid;border-color: black;}
  </style>
"@
```

```
  RETURN $format
}

# Code to call the function...
$datapath = $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\"

$instates = Import-CSV ($datapath + “state_table.csv")

$instates | Select-Object -Property name, abbreviation, country, census_region_name |
where-object -Property census_region_name -eq "Northeast" |
ConvertTo-HTML -Head (Out-UdfHTML "lightblue" -AlternateRows) -Pre "<h1>State List</h1>"
-Post ("<h1>As of " + (Get-Date) + "</h1>") |
Out-File MyReport.HTML

Invoke-Item  MyReport.HTML
```

The new thing in the code above is the use of the custom function `Out-UdfHTML` to set some custom HTML styles, which are then passed to `ConvertTo-HTML's –Head` parameter. In the line `ConvertTo-HTML -Head (Out-UdfHTML "lightblue")`, the function call is enclosed in parentheses, which causes the function to be executed before being passed as the `Head` parameter. In other words, the function call itself is treated as a string, since that is what it passes back. There are two parameters to `Out-UdfHTML`; the background color of the table heading and a switch to get the table rows to alternately toggle the background color, which adds a nice effect. To turn this feature off, we would omit the switch on the function call. The small tweak of adding the function call to define and pass custom styling produces some big benefits. So as to be complete in my coverage of `ConvertTo-HTML`, we added the `Pre` and `Post` options. In the interest of keeping the screen shot smaller so you could see the entire output, we added the `Where-Object` cmdlet line to filter down the rows. The output now should look like Figure 3-5.



State List

name	abbreviation	country	census_region_name
Connecticut	CT	USA	Northeast
Maine	ME	USA	Northeast
Massachusetts	MA	USA	Northeast
New Hampshire	NH	USA	Northeast
New Jersey	NJ	USA	Northeast
New York	NY	USA	Northeast
Pennsylvania	PA	USA	Northeast
Rhode Island	RI	USA	Northeast
Vermont	VT	USA	Northeast

As of 08/31/2014 18:22:28

Figure 3-5. *Customized HMTL to display the state list*

In the interest of simplicity, the options that the function `Out-UdfHTML` supports are limited. However, more parameters can easily be added—for example, to allow the table row's toggle background color to be customizable. Note: The HTML color values can be set using names, a hexadecimal value, or by setting the three properties' `RGB`; for instance, `RGB (255, 0, 255)`. For coverage of this, see the link: `http://www.w3schools.com/cssref/pr_background-color.asp`

The use of HTML for PowerShell output was inspired by the links that follow:

`https://technet.microsoft.com/en-us/library/ff730936.aspx`
`https://www.penflip.com/powershellorg/creating-html-reports-in-powershell/blob/master/building-the-html.txt`

Creating and Using COM Objects

Creating instances of COM objects is easy in PowerShell. Just use the New-Object cmdlet to create an instance of the object and point a variable to it as demonstrated:

```
$ie = New-Object -ComObject InternetExplorer.Application
$ie.visible = $true
$ie.silent = $true
$ie.Navigate( "www.google.com" )
```

When we run the code above, we should see an Internet Explorer window open up with the Google page displayed.

Once we have the object reference, we can access all the methods and properties it supports. From here we can navigate to sites, enter data, and so on, all via code.

Office applications such as Excel expose COM objects that can be accessed from PowerShell. The code in Listing 3-10 stores query results from SQL Server in a variable, opens Excel, and loads the data into a worksheet. Then we call the export method to write the data to a CSV file. Doing this without the Excel object would be a lot more work. Note: You must have Excel installed and change the Set-Location to point to a valid SQL Server AdventureWorks database for your environment for the script to work. This was tested with Office 365.

Listing 3-10. Loading an Excel worksheet from a SQL query

```
#  Check if the SQL module is loaded and, if not, load it.
if(-not(Get-Module -name "sqlps"))
{
  Import-Module "sqlps"
}

set-location SQLSERVER:\SQL\BryanCafferkyPC\DEFAULT\Databases\Adventureworks\Tables

$territoryrs = Invoke-Sqlcmd -Query "SELECT [StateProvinceID],[TaxType],[TaxRate],[Name]
FROM [AdventureWorks].[Sales].[SalesTaxRate];" -QueryTimeout 3

$excel = New-Object -ComObject Excel.Application

$excel.Visible = $true  # Show us what’s happening

$workbook = $excel.Workbooks.Add()  # Create a new worksheet

$sheet = $workbook.ActiveSheet # and make it the active worksheet.

$counter = 0

# Load the worksheet
foreach ($item in $territoryrs) {

    $counter++

    $sheet.cells.Item($counter,1) = $item.StateProvinceID
    $sheet.cells.Item($counter,2) = $item.TaxType
    $sheet.cells.Item($counter,3) = $item.TaxRate
    $sheet.cells.Item($counter,4) = $item.Name
}

#  Exporting Excel data is as easy as...
$sheet.SaveAs("taxrates.csv", 6)
```

Suppose we wanted to use Word to print address labels using addresses in our SQL Server database. We can use the Office Word application object to do this, as shown in Listing 3-11. Note: You must have Word installed on the machine for the code to work.

Listing 3-11. Automating Word in a PowerShell script

```
Import-Module "sqlps"

$word = New-Object -ComObject Word.Application
$doc = $word.Documents.Add()

$stores = Invoke-Sqlcmd -Query "select top 10 * from
[AdventureWorks].Sales.vStoreWithAddresses;" -QueryTimeout 3

$outdoc = ""

foreach ($item in $stores) {
    $outdoc = $outdoc + $item.Name + "`r`n"
    $outdoc = $outdoc + $item.AddressLine1 + "`r`n"
    $outdoc = $outdoc + $item.City + ", " + $item.StateProvinceName + " " + $item.PostalCode
    + "`r`n"
    $outdoc = $outdoc + "`f"  # Does a page break
}

$outtext = $doc.Content.Paragraphs.Add()
$outtext.Range.Text = $outdoc
$word.visible = $true
```

This code is simplified to keep it concise, but the principles are all there. We are getting the data from SQL Server and loading it into a new Word document. We are using the escape sequences tab (`` `t ``), line feed (`` `n ``), and form feed for a page break (`` `f ``). The point here is that we can easily leverage Office applications like Excel, Word, Access, and PowerPoint.

Creating and Using .Net Objects

Accessing COM objects is easy. Creating and using .Net objects is even easier. This is because PowerShell is integrated with the .Net framework, so much so that each release of the .Net framework usually comes with a new version of PowerShell.

Since a key thing in database development is accessing databases, let's try a different way to get to SQL Server. After all, we may not always have the SQLPS module installed. Let's use the .Net library directly, as shown in Listing 3-12.

Listing 3-12. Querying SQL Server using the .Net library

```
$conn = New-Object System.Data.SqlClient.SqlConnection ("Server=(local);
DataBase=AdventureWorks;Integrated Security=SSPI")
$conn.Open()  | out-null

$cmd = new-Object System.Data.SqlClient.SqlCommand("select top 10 FirstName, LastName from
Person.Person", $conn)

$rdr = $cmd.ExecuteReader()

While($rdr.Read()){
    Write-Host "Name: " $rdr['FirstName'] $rdr['LastName']
}

$conn.Close()
$rdr.Close()
```

The nice thing about the code in Listing 3-12 is that it should always work and it has less overhead than the SQLPS module. Since we don't want a COM object, we omit the `COM` parameter and request a specific .Net object, i.e., `System.Data.SqlClient.SqlConnection`. When actually using this code, a good idea would be to create a function with parameters for the server and database name. Note: Although the line below wraps, it must be all on one line.

```
$conn = new-Object System.Data.SqlClient.
SqlConnection("Server=(local);DataBase=AdventureWorks;Integrated Security=SSPI")
```

This code creates the connection object, which is our hook into the database. We open the connection with this statement:

```
$conn.Open()  | out-null
```

This line opens the database connection, but to do something with it, we need to create a command object, as shown here:

```
$cmd = new-Object `
System.Data.SqlClient.SqlCommand("select top 10 FirstName, LastName from Person.Person",
$conn)
```

Above, we pass the query string in as the first parameter and the connection object as the second, and we get the command object returned. We could do any type of SQL command including calling stored procedures with parameters, which we will cover later. The line below will execute the query using `ExecuteReader`.

```
$rdr = $cmd.ExecuteReader()
```

The line above calls the command object `ExecuteReader` method, which runs the query and returns a reader object. We can use the `Read` method to get the query results, as shown here:

```
While($rdr.Read())
{
    Write-Host "Name: " $rdr['FirstName'] $rdr['LastName']
}
```

The While loop above iterates over the results returned by the ExecuteReader writing the first and last name columns to the console. The loop automatically ends after all the rows have been processed. Then we just close things up with these statements:

```
$conn.Close()
$rdr.Close()
```

Remember to close your connection and reader objects. Holding database connections indefinitely ties up valuable resources.

Waiting for a File

Sometimes we need to take an action when an event occurs, such as a new file arriving on the server. There is a .Net object we can use to do this called the File System Watcher. To use it, we register a subscription to the event we want monitored and specify the code to execute when the event occurs. In the interest of making the examples reusable, the code in Listing 3-13 is formatted as a function.

Listing 3-13. Function Register-UdfFileCreateEvent to trap file events

```
function Register-UdfFileCreateEvent([string] $source, [string] $filter)
{
   try
   {
     Write-Host "Watching $source for new files..."

     $fsw = New-Object IO.FileSystemWatcher $source, $filter -Property
     @{IncludeSubdirectories = $false; NotifyFilter = [IO.NotifyFilters]'FileName,
     LastWrite'}

     Register-ObjectEvent $fsw Created -SourceIdentifier FileCreated -Action {
     write-host "Your file has arrived."   }

    }
   catch
    {
      "Error registering file create event."
    }
}

$datapath = $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\"
Register-UdfFileCreateEvent $datapath "*.txt"
```

The Try/Catch command used in Listing 3-13 is a nice feature supported by PowerShell. PowerShell will try to execute the statements between the braces after the Try statement. If it fails, the statements in the Catch block are executed. This is an elegant way to trap errors. The line copied below creates a FileSystemWatcher object.

```
$fsw = New-Object IO.FileSystemWatcher $source, $filter -Property @{IncludeSubdirectories =
$false; NotifyFilter = [IO.NotifyFilters]'FileName, LastWrite'}
```

The statement above creates a `FileSystemWatcher` object where `$source` is the folder to be monitored and `$filter` is the file pattern to be looked for. In this case, we want to fire the event for any file ending in `.txt`. Subdirectories will not be monitored due to the parameter `IncludeSubdirectories = $false`. Then we need to register the event we want to monitor, which the following code does:

```
Register-ObjectEvent $fsw Created -SourceIdentifier FileCreated -Action {
     write-host "Your file has arrived."   }
```

The line above is subscribing to the `FileCreated` event, and the code enclosed in braces after the `Action` parameter is the code that will be executed when the event occurs; i.e., the line "`Your file has arrived.`" will be displayed.

We are pushing the work of waiting for the file to arrive to Windows. We can also subscribe to `FileDeleted` and `FileChanged` events. However, you can only subscribe to one type of `FileWatcher` event at a time—i.e., once you've subscribed to the `FileCreated` event, you cannot add another subscription to that event. To remove a subscription from an event, we use the `Unregister-Event` cmdlet, passing to the event type as shown here:

```
Unregister-Event FileDeleted
Unregister-Event FileCreated
Unregister-Event FileChanged
```

Summary

The goal of this chapter is to demonstrate the wide-ranging capabilities of PowerShell as a development platform. To this end, we discussed a wide variety of advanced programming techniques. We started by discussing the use of parameters to support code reusability. This led to a discussion of the `CmdletBinding` attribute, which causes PowerShell to add a number of powerful features to our code, including parameter validation, parameter sets, and support for PowerShell's common parameters. Since maximum reusability can only be accomplished with functions, we discussed how to develop functions using examples. From here we moved on to a discussion of creating custom objects in PowerShell complete with properties and methods. To fully leverage PowerShell, we need to understand how to use the pipeline. We discussed how to write functions that employ the special code blocks `begin`, `process`, and `end` to use the pipeline. Then we discussed how we can create highly customized output using a function that generates HTML. Windows has built-in support for application automation, so we reviewed an example of leveraging this to load an Excel spreadsheet with data from SQL Server and an example of Word automation. Although we can use the SQLPS module to access SQL Server, this adds unnecessary overhead, so we demonstrated how we can write code to directly query SQL Server tables using the .Net library. There are times when we need to know when something happens to files on the network. For example, when a file is created, we may need to run an ETL job to load it. We discussed trapping such events to execute custom code using the .Net `FileSystemWatcher` object. Hopefully, the limitless capabilities of PowerShell as a development language have been successfully demonstrated.